Python для профи

Интерпретаторы, эмуляторы, графика и машинное обучение

Дэвид Копек



Дэвид Копек

Python для профи

COMPUTER SCIENCE FROM SCRATCH

Building Interpreters, Art, Emulators, and ML in Python

by David Kopec



no starch press®
San Francisco

PYTHON ДЛЯ ПРОФИ

Интерпретаторы, эмуляторы, графика и машинное обучение

Дэвид Копек

BOOKS.KZ
2026

УДК 004+002
ББК 16
К60

Копек Д.
К60 Python для профи: интерпретаторы, эмуляторы, графика и машинное обучение / пер. с англ. В. И. Бахура. – Астана: Books.kz, 2026. – 288 с.: ил.

ISBN 978-6-01140-652-9

Благодаря семи проектам на Python, представленным в этой книге, вы сможете получить представление о некоторых фундаментальных принципах в области информатики. В каждой главе автор пошагово описывает завершенный проект, раскрывающий один из конкретных аспектов программирования, не прибегая к сложной математике и запутанному коду.

Издание адресовано программистам на языке Python среднего и продвинутого уровней, желающим закрепить знания и заполнить возможные пробелы в образовании.

УДК 004+002
ББК 16

Title of English-language original: Computer Science from Scratch: Building Interpreters, Art, Emulators, and ML in Python, ISBN 9781718504301, published by No Starch Press Inc. 245 8th Street, San Francisco, California United States 94103. The Russian-language 1st edition Copyright © 2026 by Books.kz LLC under license by No Starch Press Inc. All rights reserved.

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

ISBN 978-1-7185-0430-1 (англ.)
ISBN 978-6-01140-652-9 (казах.)

Copyright © 2025 by David Kopec
© Перевод, оформление, издание, Books.kz, 2026

Посвящается моей маме, Сильвии,
которая радовалась каждой моей победе
и помогала мне переживать каждое поражение

СОДЕРЖАНИЕ

От издательства ... 10
Благодарности ... 12
Введение ... 14

Часть I ИНТЕРПРЕТАТОРЫ ... 19

Глава 1. Минимально возможный язык программирования ... 20

Что такое Brainfuck? ... 20
Что определяет полноту языка по Тьюрингу? ... 21
Как работает Brainfuck ... 22
Структура интерпретатора ... 27
Реализация Brainfuck на Python ... 28
Получение исходного файла ... 29
Создание интерпретатора ... 29
Запуск интерпретатора ... 33
Тестирование интерпретатора ... 34
Практические приложения ... 37
Упражнения ... 38

Глава 2. Создание интерпретатора языка BASIC ... 39

Основы NanoBASIC ... 40
История BASIC ... 40
Парадигма, синтаксис и семантика NanoBASIC ... 41
Операторы GOTO, GOSUB и RETURN ... 45
Стиль и особенности NanoBASIC ... 46
Пример программы NanoBASIC ... 47
Формализация синтаксиса NanoBASIC ... 48
Реализация NanoBASIC ... 53
Токенизатор ... 54

Узлы ... 58
Ошибки ... 61
Парсер ... 63
Среда выполнения ... 72
Запуск программы ... 77
Тестирование NanoBASIC ... 77
Практические приложения ... 82
Упражнения ... 83

Часть II ИСКУССТВО И ВЫЧИСЛЕНИЯ ... 84

Глава 3. Ретрообработка изображений ... 85

Что такое дизеринг? ... 85
Начало работы ... 88
Алгоритм дизеринга ... 90
Файловый формат MacPaint ... 95
Преобразование байтов в биты ... 96
Реализация кодирования по длине последовательности ... 98
Тестирование кодирования по длине последовательности ... 103
Преобразование в MacBinary ... 104
Сведение всего воедино ... 108
Результаты ... 109
Практические приложения ... 112
Упражнения ... 113

Глава 4. Стохастический алгоритм живописи ... 114

Как это работает ... 114
Опции командной строки ... 120
Формат SVG ... 121
Алгоритм ... 124
Основная реализация ... 125
Настройка ... 125
Вспомогательные методы ... 128
Пробы ... 129
Вывод изображения ... 132
Результаты ... 134
Практические приложения ... 138
Упражнения ... 139

Часть III ЭМУЛЯТОРЫ ... 141

Глава 5. Создание виртуальной машины CHIP-8 ... 142

Виртуальные машины ... 142
Виртуальная машина CHIP-8 ... 144

Регистры и память 145
Инструкции 146
Реализация 150
Цикл выполнения 151
Аргументы командной строки 154
Настройка VM и вспомогательные функции 154
Графика 157
Выполнение инструкций 160
Тестирование VM 166
Запуск игр 167
Практические приложения 169
Упражнения 170

Глава 6. Эмуляция игровой консоли NES 172
О платформе NES 173
Аппаратная платформа 174
Программное обеспечение 176
Создание эмулятора 177
Планирование структуры 177
Создание главного цикла 178
Эмуляция картриджа 181
Эмуляция центрального процессора 185
Принципы работы PPU 212
Реализация PPU 222
Тестирование эмулятора 233
Игровой процесс 236
Практические приложения 240
Упражнения 241

Часть IV ОЧЕНЬ ПРОСТОЕ МАШИННОЕ ОБУЧЕНИЕ 242

Глава 7. Классификация с помощью k-ближайших соседей 243
Становление машинного обучения 243
Как работает KNN 245
Реализация классификации с помощью KNN 247
Классификация рыб 250
Классификация рукописных цифр 254
Практические приложения 258
Упражнения 259

Глава 8. Регрессия с помощью k ближайших соседей 260
Как работает регрессия KNN 260
Реализация регрессии с помощью KNN 262
Прогнозирование веса рыб 263

Прогнозирование недостающей части рукописной цифры 264
Практические приложения 268
Упражнения 269

Послесловие 270

Что мы сделали и что будет дальше 270
Об изучении компьютерных технологий 271
Интерпретаторы 273
Компьютерное изобразительное искусство 273
Эмуляторы 274
Машинное обучение 274

Приложение. Побитовые операции 276

Обзор двоичной системы счисления 276
Распространенные побитовые операции 278
Сдвиг влево (<<) 278
Сдвиг вправо (>>) 279
ИЛИ (|) 280
И (&) 281
XOR (^) 282
Дополнение (~) 283

Предметный указатель 284

ОТ ИЗДАТЕЛЬСТВА

Отзывы и пожелания

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

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

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

Список опечаток

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

Нарушение авторских прав

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

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

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

Об авторе

Дэвид Копек (David Kopec) – доцент кафедры информатики в колледже Олбрайт. Он присоединился к академическому сообществу в 2016 году после работы в качестве разработчика программного обеспечения со специализацией в области создания приложений для iOS. Является автором четырех технических книг, в том числе «*Классические проблемы информатики на Python*» (*Classic Computer Science Problems in Python*), которая была переведена на восемь языков и опубликована по всему миру. Копек – активный разработчик приложений и подкастер. Живет с женой и тремя детьми в городе Вайомиссинг, штат Пенсильвания, США.

О техническом рецензенте

Майкл Кеннеди (Michael Kennedy) хорошо известен среди специалистов по Python благодаря своей работе над подкастами *Talk Python to Me* и *Python Bytes*, в которых на протяжении почти 10 лет освещаются важные темы и новости сообщества Python. Является основателем курсов Talk Python Training, где предлагается множество онлайн-курсов для разработчиков, а также членом Python Software Foundation. Кеннеди живет в Портленде, штат Орегон, США. Когда он не занимается программированием и не проводит время с семьей, его можно увидеть путешествующим на мотоцикле по местным горам.

БЛАГОДАРНОСТИ

Прежде всего я хотел бы поблагодарить вас, читатель, за приобретение этой книги. Вероятно, вы могли бы найти в интернете учебные материалы по большинству тем, затронутых в этой книге, но вряд ли они будут настолько продуманы, согласованы и тщательно оформлены. По крайней мере на мой взгляд, и именно поэтому я написал эту книгу. Приобретая ее, вы поддержали не только ее создание, но и дальнейшее существование всей отрасли, выпускающей подобные книги.

Далее я хотел бы поблагодарить колледж Шамплейн, который предоставил мне творческий отпуск после семи лет работы. Осталось не так много отраслей, где сотрудникам дают почти год отпуска, чтобы они могли заниматься своими профессиональными делами, и при этом еще и платят им за это. Иногда из этого времени и пространства рождается что-то замечательное. Написание данной книги было моим проектом на период этого творческого отпуска.

Я хотел бы поблагодарить свою семью, которая поддерживала меня в работе над этим проектом, особенно мою маму Сильвию и мою жену Ребекку. Мои дети, Дэниел, Вера и Люсиль, еще слишком малы, чтобы по-настоящему осознавать, что значит писать техническую книгу, но я не смог бы ее написать, если бы Ребекка не заботилась о них, поэтому вместо того, чтобы благодарить их, я поблагодарю ее во второй раз.

Я хотел бы поблагодарить издательство No Starch Press за то, что они поверили в эту книгу. Я пришел к ним с уже полностью написанным черновым вариантом рукописи, что является немного необычным для технической книги, и они поверили в ее содержание настолько, что вложили ресурсы в ее доработку ради вашего удобства. В частности, я хотел бы поблагодарить Натана Хайдельбергера (Nathan Heidelberger), моего технического редактора, который нашел время, чтобы сопереживать читателям и найти как масштабные, так и небольшие возможности для улучшения книги. И конечно же, я хотел бы поблагодарить остальных членов команды No Starch Press, которые помогли

в подготовке и издании книги. Я также хотел бы поблагодарить моего технического рецензента Майкла Кеннеди (Michael Kennedy) за его ценные предложения.

И наконец, я хочу поблагодарить всех, кто оставил публичные отзывы о моих предыдущих книгах. Если бы вы не отреагировали положительно на мои предыдущие книги и не нашли время, чтобы это зафиксировать, эта книга никогда бы не получила такого импульса для старта. Если вы читаете это, пожалуйста, найдите время, чтобы оставить отзыв и об этой книге!

ВВЕДЕНИЕ



Как устроен язык программирования? Как устроен простой компьютер? Я отношусь к тому типу людей, которые любят изучать новые предметы, начиная с самых основ, на практике. Мне нужно больше, чем просто общий обзор. Если и вы относитесь к такому типу людей, то вы нашли нужный ресурс. Благодаря семи проектам на Python, представленным в этой книге, вы сможете получить представление о некоторых фундаментальных принципах в области компьютерных наук.

Для кого эта книга

Эта книга предназначена для программистов на языке Python среднего и продвинутого уровней. Если вы начинающий программист, вам, вероятно, стоит вернуться к этой книге позже. На протяжении всего текста подразумевается, что читатель знаком с синтаксисом и семантикой Python, умеет писать программы средней сложности, знает, как устанавливать библиотеки Python, и разбирается в базовых структурах данных, таких как списки, наборы и словари.

Несмотря на предположение о наличии у читателей некоторого опыта программирования, я не рассчитываю на их знания в области компьютерных технологий или высшей математики. Эта книга предназначена для тех, кто не имеет профильного образования в области вычислительной техники или хочет восполнить некоторые пробелы в своих знаниях. Например, если вы заинтересованы в написании собственного языка программирования, но никогда не посещали курсы по компиляторам, эта книга будет отличной отправной точкой. Если вы хотите написать эмулятор игровой консоли, эта книга подскажет вам, как это сделать. В ней также есть вполне понятное введение в тему машинного обучения.

Конкретные проекты, описанные в данной книге, могут не стать вашей конечной целью, но это и не требуется. Рассматривайте их как

инструмент для получения более глубоких знаний об алгоритмическом мышлении и принципах работы программного обеспечения, а также в качестве стартового этапа для ваших собственных исследований.

О чем эта книга

Каждая глава представляет собой полноценный законченный проект, за исключением глав 7 и 8, которые вместе составляют один проект. Сложность представленных в книге семи проектов варьируется от легкой (интерпретатор Brainfuck в главе 1) до сложной (эмулятор NES в главе 6), но благодаря наличию исходного кода вы никогда не окажетесь в тупике и всегда сможете продолжить работу.

Каждый проект начинается с изложения определенной теоретической базы – достаточной для понимания того, что именно будет реализовано, без углубления в детали, а затем следует подробное описание кода. Главы также включают описание моих личных причин интереса к данной теме, обсуждение того, как реализуемые алгоритмы или вычислительные методы используются в реальной жизни, а также задачи для читателя с целью дальнейшего использования предоставленного кода.

Книга разделена на четыре части. Часть I посвящена изучению мира интерпретаторов путем написания реализаций двух простых языков программирования.

Глава 1 «Минимально возможный язык программирования». Brainfuck – это минималистичный язык программирования, который часто используется в образовательных целях из-за его простоты: весь язык состоит всего из восьми символов. Мы познакомимся с принципом работы очень простого интерпретатора, создадим свой собственный, который сможет запускать любую программу на Brainfuck. Мы также разберемся со значением термина «язык, полный по Тьюрингу».

Глава 2 «Создание интерпретатора языка BASIC». Язык программирования BASIC и его упрощенный вариант Tiny BASIC были популярны во времена компьютерной революции конца 1970-х годов. Мы разработаем интерпретатор для несколько упрощенного варианта Tiny BASIC под названием NanoBASIC. Это даст нам представление о компонентах более сложных интерпретаторов, в том числе о токенизаторе, парсере и среде выполнения.

В части II нам предстоит погрузиться в яркий мир компьютерного изобразительного искусства.

Глава 3 «Ретрообработка изображений». Когда технологии отображения были более простыми, для адаптации изображений к устройствам с ограниченной цветовой палитрой требовались алгоритмы дизеринга, то есть размытия (сглаживания). Мы разработаем

алгоритм дизеринга, способный отображать современные цветные фотографии на черно-белом экране оригинального Macintosh. Затем конвертируем полученные изображения в формат, совместимый с классическим приложением MacPaint, с использованием алгоритма сжатия с кодированием длины последовательности. Полученные изображения можно выводить на экран реальной системы Macintosh 1980-х годов.

Глава 4 «Алгоритм стохастического раскрашивания». Может ли сравнительно простой алгоритм помочь в создании сложного абстрактного искусства? Мы будем использовать стохастическую технику для генерации «впечатлений» из уже существующих изображений путем сопоставления случайных форм с исходным изображением. Затем разберемся с оптимизацией результатов с помощью алгоритма поиска максимума.

Часть III посвящена эмуляторам – программам, которые позволяют одним компьютерным системам имитировать работу других.

Глава 5 «Создание виртуальной машины CHIP-8». CHIP-8 – это спецификация виртуальной машины (Virtual Machine, VM), которая первоначально использовалась для разработки видеоигр в 1970-х годах. Создание виртуальной машины CHIP-8 обычно считается наиболее подходящим вариантом для первого знакомства с миром эмуляции: это относительно просто, но при этом включает в себя все этапы, необходимые для создания эмулятора. Наша виртуальная машина CHIP-8 будет способна запускать все игры CHIP-8, которые работали на системах в 1970-х годах.

Глава 6 «Эмуляция игровой консоли NES». NES была одной из самых продаваемых игровых консолей всех времен. Мы создадим эмулятор, который сможет запускать настоящие игры для NES. Он будет без звука, будет довольно медленным и не будет полностью точным или универсально совместимым, но все же станет отличным способом изучить не только эмуляторы, но и то, как работают компьютеры на низком уровне.

Наконец, часть IV представляет собой очень доступное введение в мир машинного обучения с использованием алгоритма k-ближайших соседей (k-nearest neighbors, KNN).

Глава 7 «Классификация с помощью k-ближайших соседей». Мы изучим KNN, который является, пожалуй, самым простым алгоритмом в машинном обучении (Machine Learning, ML), и задействуем его в качестве отправной точки для ознакомления с некоторыми вводными тезисами ML. Мы будем использовать KNN для классификации рыб, а также изображений рукописных цифр. Удивительно, но он справляется с последней задачей с точностью 98 %.

Глава 8 «Регрессия с помощью k ближайших соседей». Мы перейдем на следующий уровень KNN, используя его не только для классификации элементов по категориям, но и для прогнозирования неизвестных атрибутов точек данных. В конце главы с его помощью мы

будем предугадывать отсутствующие пиксели на изображении цифры, нарисованной пользователем.

В дополнение к основным главам в послесловии представлены некоторые рекомендуемые ресурсы для более глубокого изучения тем, затронутых в этой книге, а в приложении освещаются основы низкоуровневого оперирования битами в Python, что необходимо для реализации ряда проектов.

Подход, принятый в этой книге

Я всегда стараюсь, чтобы мои книги были максимально лаконичными. Я ценю ваше время. Я использую формат, похожий на самоучитель, с упором на код, и по возможности даю коду возможность говорить самому за себя.

Это не учебник. Вы найдете в нем немного теории, особенно в начале каждой главы, но она никогда не затягивается, и вскоре мы переходим к коду. В книге содержится ровно столько информации, сколько нужно для понимания того, как работает каждый из проектов, и достаточно подсказок, чтобы вы были в курсе, где искать дополнительную информацию в случае возникновения желания углубиться в какую-либо из затронутых тем.

Я не утверждаю, что являюсь экспертом в области интерпретаторов, компьютерного изобразительного искусства, эмуляторов или машинного обучения. Это может показаться странным, когда это говорит автор книги по данным темам, но это действительно так. Я не эксперт, я учитель. Я работал разработчиком программного обеспечения и преподавателем по компьютерным технологиям в педагогическом колледже. Я утверждаю, что умею писать чистый код и объяснять его вам исключительно понятным образом. А поскольку я не эксперт, я не буду смотреть на вас свысока. Я буду относиться к вам как к равному, так как мы пройдем этот путь вместе. Это руководство, которое я хотел бы иметь под рукой в те времена, когда начинал самостоятельную работу над проектами в данных областях.

О коде

Весь исходный код, используемый в этой книге, доступен в сопутствующем репозитории GitHub по адресу *https://github.com/davecom/ComputerScienceFromScratch*. Код был создан и протестирован с Python версий 3.12 и 3.13. Поскольку здесь используются некоторые функции Python 3.12, связанные с подсказками типов, часть кода не будет работать с более ранними версиями Python (но, вероятно, будет работать с любой новой версией Python для обозримого будущего). Тем не менее если удалить подсказки типов, большая часть кода будет работать с Python версии 3.10 и более поздними.

Я использовал подсказки типов Python (или «аннотации типов») во всем исходном коде, поскольку считаю, что они повышают удобочитаемость, указывая типы параметров и возвращаемых значений функции, без необходимости тщательного изучения кода или комментариев. Если они вам не нравятся, вы можете их игнорировать; они никак не влияют на работу кода. Я старался не злоупотреблять подсказками типов, так как некоторые считают их слишком многословными. Например, я редко использую их в телах функций, но использую в каждой сигнатуре функции. Я проверил весь исходный код на соответствие современной версии Pyright на момент написания книги.

В нескольких проектах этой книги используются внешние библиотеки. Вам необходимо установить Pygame, NumPy и Pillow в виртуальной среде, созданной для исходного кода книги, или в системном интерпретаторе Python. Для большинства читателей их установка не составит труда: достаточно выполнить команду `pip install pygame`, `numpy`, `pillow`. Файл *requirements.txt*, который может использовать менеджер пакетов pip, включен в репозиторий исходного кода книги.

Исправления и комментарии

Репозиторий книги на GitHub – это отличное место, где можно оставить сообщение в случае обнаружения ошибки. Вы также можете связаться со мной по электронной почте *csfromscratch@oaksnow.com* или через X *@davekopec*. Я буду рад вашим отзывам, как положительным, так и отрицательным. Если вам понравилась книга, пожалуйста, оставьте отзыв на Amazon или в том месте, где вы ее приобрели.

ЧАСТЬ I

ИНТЕРПРЕТАТОРЫ

1

МИНИМАЛЬНО ВОЗМОЖНЫЙ ЯЗЫК ПРОГРАММИРОВАНИЯ



Каким может быть минимально возможный язык программирования, который все еще был бы пригоден для решения реальных задач? Одним из кандидатов на это звание, безусловно, является Brainfuck – уникальный язык программирования, который был разработан Урбаном Мюллером (Urban Müller) в 1993 году. В этой главе мы создадим интерпретатор Brainfuck. Возможно, это самый простой язык программирования, для которого можно написать интерпретатор – вы будете удивлены его лаконичности. К концу главы вы узнаете не только о принципах работы Brainfuck, но и об основных принципах работы любого интерпретатора.

Что такое Brainfuck?

В языке Brainfuck используется всего восемь команд (+, -, ., ,, >, <, [,]), и каждая команда обозначается одним символом. Вот пример программы «Hello World!» на языке Brainfuck[1]:

```
++++++++[>++++[>++>+++>+++>+<<<<-]>+>+>->>+[<]<-]>>.>---.+++++++..+++.>>.<-.<.
+++.------.-----\---.>>+.>++.
```

[1] "Brainfuck", Esolangs.org, просмотрено 22 мая 2024 г., *https://esolangs.org/wiki/Brainfuck#Hello.2C_World.21*.

Это может выглядеть весьма необычно, но это реальная программа. Экзотический синтаксис и минимальный набор функций Brainfuck делают его непригодным для каких-либо практических целей. Скорее, это игрушка, которая может быть полезна в качестве образовательной модели. Но это игрушка, полная по Тьюрингу!

Что определяет полноту языка по Тьюрингу?

Язык программирования считается *полным по Тьюрингу* (Turing-complete) в том случае, если он может имитировать машину Тьюринга – абстрактную модель, способную реализовывать любой компьютерный алгоритм[1]. Чтобы получить представление о машине Тьюринга, вообразите ленту неограниченной длины с разбивкой на ячейки, которые либо пусты, либо содержат символ. Затем представьте головку, которая может считывать или записывать символ в ячейке, в том числе и удалять его. Представьте, что головка может перемещаться влево или вправо по одной ячейке за один раз. Наконец, представьте, что она может записывать или перемещаться в зависимости от прочитанного значения, то есть что она может *ветвиться*. Другими словами, головка следует некоторым простым правилам, которые можно рассматривать как программу, в которой, по сути, говорится: «Если прочитано это значение, запиши другое значение. Если прочитано определенное значение, нужно переместиться влево на одну ячейку».

Вот и все. Этого достаточно, чтобы реализовать любой вычислительный алгоритм. Как следствие любой язык программирования, который может имитировать эту функциональность – даже такой простой язык, как Brainfuck, может использоваться для решения реальных задач. На рис. 1.1 показана гипотетическая машина Тьюринга.

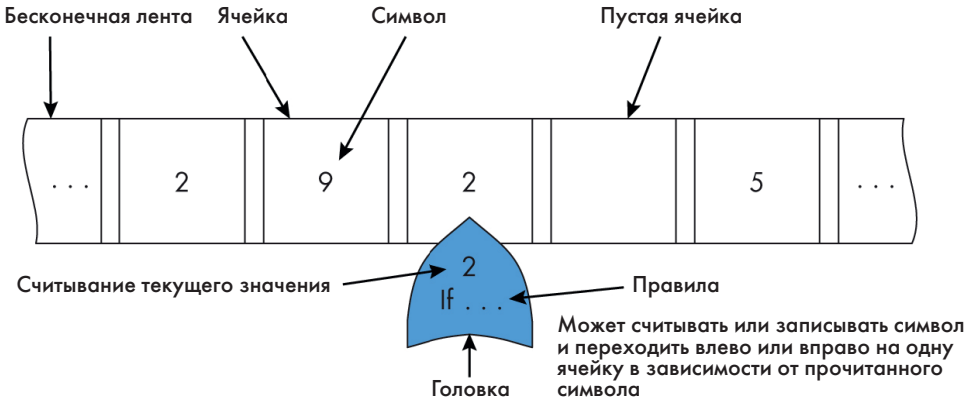


Рис. 1.1. Гипотетическая машина Тьюринга с бесконечной лентой, ячейками и следующей правилам головкой

[1] Terrence W. Pratt and Marvin V. Zelkowitz, *Programming Languages: Design and Implementation*, 3rd ed. (Prentice Hall, 1996), 409.

Какими характеристиками должен обладать язык программирования, чтобы быть полным по Тьюрингу? Как объясняют Аллен Такер (Allen Tucker) и Роберт Нунан (Robert Noonan) в своей книге «*Языки программирования: принципы и парадигмы*», для этого многого не требуется:

> Язык программирования считается полным по Тьюрингу в том случае, если он содержит целочисленные переменные, значения и операции, а также имеет операторы присваивания и конструкции управления последовательностью операторов, условные операторы и операторы ветвления. Все другие формы операторов (циклы while и for, выбор по условию, объявления и вызовы процедур и т. д.) и типы данных (строки, значения с плавающей запятой и т. д.) предоставляются в современных языках только для облегчения программирования различных сложных приложений[1].

Разрешите мне упростить это описание и пересказать его более простым языком. Чтобы быть полным по Тьюрингу, язык программирования должен иметь целочисленные переменные, способ изменения значений, связанных с этими переменными, что-то вроде оператора `if` и что-то вроде оператора `goto` (то есть «перехода»). Это не так уж и много. И если подумать, можно представить себе, как эти простые структуры могут соотноситься с элементами машины Тьюринга. Целочисленные переменные – это символы в ячейках, изменение переменных – это запись, сделанная головкой в ячейках, а операторы `if` и `goto` обозначают ветвление головки.

Любой язык программирования, который является полным по Тьюрингу, способен обеспечить реализацию тех же алгоритмов, что и любой другой язык программирования, который является полным по Тьюрингу. Вы можете реализовать Quicksort на C, но вы также можете реализовать Quicksort на Brainfuck. Вы можете написать JSON-парсер на Python, но вы также можете написать JSON-парсер на Brainfuck. В этом смысле, хотя Brainfuck и является «игрушечным» языком программирования, он также относится к «настоящим» языкам программирования.

Как работает Brainfuck

Основным состоянием в программе Brainfuck является массив целых чисел. Каждый из слотов в массиве называется *ячейкой* (cell). Массив ячеек можно рассматривать как аналог ленты в машине Тьюринга. Вместо символов, которые читаются или записываются в ячейку,

---

[1] Allen B. Tucker and Robert E. Noonan, Programming Languages: Principles and Paradigms, 1st ed. (McGraw-Hill, 2002), 84.

используются целые числа. Команды Brainfuck позволяют программисту перемещаться вперед на одну ячейку (`>`), перемещаться назад на одну ячейку (`<`), увеличивать значение ячейки (`+`), уменьшать значение ячейки (`-`), выводить ячейку (`.`), вводить данные в ячейку (`,`) и выполнять цикл, пока значение определенной ячейки отлично от нуля (`[and]`). Некоторые из этих операций напрямую соответствуют операциям машины Тьюринга, поэтому Brainfuck является полным по Тьюрингу.

В Python нам понадобятся две переменные для хранения состояний ячеек: список всех ячеек (`cells`) и целое число, представляющее индекс текущей ячейки (`cell_index`). Кроме того, нам необходимо отслеживать, где мы находимся в исходном файле Brainfuck. Для этого мы будем использовать еще одно целое число, называемое `instruction_index`.

C в сравнении с Python

В реализации интерпретатора Brainfuck на языке C ячейки могут управляться с помощью одного указателя на массив целых чисел. Указатель может использоваться для инициализации памяти для ячеек и перехода от ячейки к ячейке, а также может быть разыменован для изменения значения ячейки.

Некоторые из вас, возможно, не знают C, но я включаю этот момент для иллюстрации различий между языками программирования. В языке C благодаря возможностям указателей у нас есть одна переменная, тогда как в нашем интерпретаторе Brainfuck на Python для получения той же функциональности требуется несколько переменных.

Указатели C очень эффективны, но они также небезопасны и сложны для понимания начинающими программистами. Разумеется, разные языки программирования предлагают разные компромиссы при реализации интерпретатора. Интерпретатор Brainfuck на C также будет работать намного быстрее, чем интерпретатор Brainfuck на Python. Основной интерпретатор для самого Python, CPython, написан на C.

В табл. 1.1 приведены данные из доклада Мюллера 2017 года, где описаны команды с обозначением каждого символа[1]. Я перевел описания на C на Python с использованием только что описанных имен переменных.

[1] Urban Müller, «Brainfuck, or How I Learned to Change the Problem», лекция на Tamedia TX 2017, Цюрих, Швейцария, 13 июня 2017 г., посещение 10 июня 2022 г., YouTube, 3:50:11, *https://youtu.be/gjm9irBs96U?t=8610*.

Таблица 1.1 Команды Brainfuck

Команда	Эквивалент на Python	Описание
>	`cell_index += 1`	Перемещение на одну ячейку вправо
<	`cell_index -= 1`	Перемещение на одну ячейку влево
+	`cells[cell_index] += 1`	Увеличение текущей ячейки
-	`cells[cell_index] -= 1`	Уменьшение текущей ячейки
.	`print(chr(cells[cell_index]), end='', flush=True)`	Вывод ASCII-значения текущей ячейки
,	`cells[cell_index] = int(input())`	Считывание значения текущей ячейки
[	`if cells[cell_index] == 0: instruction_index = self.find_bracket_match(instruction_index, True)`	Если ячейка равна нулю, переход к соответствующей закрывающей скобке
]	`if cells[cell_index] != 0: instruction_index = self.find_bracket_match(instruction_index, False)`	Если ячейка не равна нулю, переход к соответствующей открывающей скобке

Благодаря табл. 1.1 у нас есть достаточно информации, чтобы пройтись по программе Brainfuck и понять все ее действия. Рассмотрим простую программу, которая выводит один заданный пользователем символ заданное пользователем количество раз. Вот полная программа, в которой каждая команда помечена индексом, чтобы мы могли сослаться на нее позже (вы можете найти эту программу в репозитории исходного кода книги, в *Brainfuck/Examples/repeat.bf*):

```
,>,[<.>-]
012345678
```

Давайте рассмотрим работу этой программы последовательно по каждой команде. Для этого мы опишем процесс работы каждой команды и покажем ее влияние на состояние программы с помощью таблицы. Программе требуется всего две ячейки, плюс индекс ячейки и индекс инструкции. Изначально таблица выглядит следующим образом:

Таблица 1.2 Работа программы Brainfuck

Cell 0	Cell 1	Индекс ячейки	Индекс инструкции
0	0	0	0

Вот так выглядит первая команда (для ясности мы будем предварять каждую команду индексом инструкции и двоеточием):

```
0: ,
```

Ввод пользователя извлекается и сохраняется в ячейке `0`, поскольку индекс ячейки изначально равен `0`. Для нашего примера представим, что пользователь ввел `88` (ASCII-код заглавной буквы *X*). После выполнения команды индекс инструкции увеличивается.

Таблица 1.3 Работа программы Brainfuck

Cell 0	Cell 1	Индекс ячейки	Индекс инструкции
88	0	0	1

```
1: >
```

Индекс ячейки увеличивается, а затем увеличивается индекс инструкции.

Таблица 1.4 Работа программы Brainfuck

Cell 0	Cell 1	Индекс ячейки	Индекс инструкции
88	0	1	2

```
2: ,
```

Пользователь вводит данные в ячейку 1. Допустим, пользователь ввел 10. Тогда индекс инструкции увеличивается.

Таблица 1.5 Работа программы Brainfuck

Cell 0	Cell 1	Индекс ячейки	Индекс инструкции
88	10	1	3

```
3: [
```

Потенциально возможен запуск цикла. Поскольку значение текущего индекса ячейки не равно 0 (оно равно 10), вместо перехода к соответствующей закрывающей скобке мы просто увеличиваем индекс инструкции на 1 для перехода к следующей команде.

Таблица 1.6 Работа программы Brainfuck

Cell 0	Cell 1	Индекс ячейки	Индекс инструкции
88	10	1	4

```
4: <
```

Индекс ячейки уменьшается. Затем увеличивается индекс инструкции.

Таблица 1.7 Работа программы Brainfuck

Cell 0	Cell 1	Индекс ячейки	Индекс инструкции
88	10	0	5

```
5: .
```

ASCII-значение ячейки с текущим индексом выводится на консоль – в данном случае это *X*. Затем индекс инструкции увеличивается.

Таблица 1.8 Работа программы Brainfuck

Cell 0	Cell 1	Индекс ячейки	Индекс инструкции
88	10	0	6

```
6: >
```

Индекс ячейки увеличивается. Затем увеличивается индекс инструкции.

Таблица 1.9 Работа программы Brainfuck

Cell 0	Cell 1	Индекс ячейки	Индекс инструкции
88	10	1	7

```
7: -
```

Значение текущего индекса ячейки уменьшается. Индекс инструкции увеличивается.

Таблица 1.10 Работа программы Brainfuck

Cell 0	Cell 1	Индекс ячейки	Индекс инструкции
88	9	1	8

```
8:]
```

Если значение в текущем индексе ячейки отлично от нуля, мы переходим к соответствующей открывающей скобке. В данном случае ячейка 1 равна 9, поэтому происходит переход, то есть индекс инструкции становится равным 3.

Таблица 1.11 Работа программы Brainfuck

Cell 0	Cell 1	Индекс ячейки	Индекс инструкции
88	9	1	3

Сейчас мы прошли первую итерацию цикла, который будет повторяться девять раз, чтобы вывести в общей сложности 10 *X*. Инструкции с 3 по 8 будут повторяться девять раз, пока ячейка 1 не станет равной 0, и в этом случае проверка закрывающей скобки (индекс 8) завершит повторения, и программа закончится.

Чтобы завершить эту программу, перейдем дальше и запустим ее через наш интерпретатор:

```
% python3 -m Brainfuck Brainfuck/Examples/repeat.bf
88
10
XXXXXXXXXX
```

Наш интерпретатор Brainfuck принимает в качестве входных данных только целые числа. Эти целые числа могут соответствовать кодам символов ASCII, а 88 – это код символа ASCII для *X*. Поэтому ожидаемый результат – 10 символов *X*. После завершения этой главы вы сможете запускать эту и любую другую программу Brainfuck самостоятельно.

Структура интерпретатора

Интерпретаторы обычно состоят как минимум из трех частей:

- *токенизатор* (tokenizer), иногда известный как *лексер*, или *лексический анализатор* (lexer), который принимает исходный код и делит его на мельчайшие распознаваемые конструкции, допускаемые в этом языке программирования. Эти конструкции называются *токенами* (tokens). Для кода **a + 2** токенами могут быть **a**, **+** и **2**;
- *синтаксический анализатор*, или *парсер* (parser), который принимает токены, расположенные рядом друг с другом, и вычисляет их значение (то есть выражения или операторы, которые они образуют). Парсеры обычно создают дерево узлов, представляющее относительные отношения между выражениями, операторами и литеральными значениями. Это дерево называется *деревом абстрактного синтаксического анализа* (abstract syntax tree, AST). Например, если интерпретатор Python увидел токен **a**, за которым следует токен **+**, а за ним токен **2**, он может построить узел арифметического выражения и соединить его с узлами для **a** и **2**;
- *среда выполнения* (runtime environment), которая проходит по узлам AST и запускает соответствующие операции для выполнения заложенного в них смыслового содержания. Для нашего арифметического выражения **a + 2** это означает поиск значения, представленного **a**, и добавление к нему **2**.

Прелесть Brainfuck заключается в том, что каждое выражение представляет собой всего один символ, поэтому для получения токена нам нужно всего лишь прочитать один символ из исходного файла. И каждый из этих токенов сам по себе уже представляет собой узел

значения. Это делает написание интерпретатора для Brainfuck более простым, нежели написание интерпретатора для практически любого другого языка программирования. Мы можем объединить токенизатор, парсер и среду выполнения в один цикл, который объединяет эти три концепции.

Мы вернемся к идее отдельного токенизатора, парсера и среды выполнения в главе 2, где займемся созданием интерпретатора для немного более сложного языка под названием NanoBASIC. Будет интересно посмотреть, как элементы, объединенные в нашем интерпретаторе Brainfuck, разбиваются на части для нашего интерпретатора NanoBASIC.

Интерпретаторы Brainfuck не только просты в создании, но и очень компактны. Вдохновленный другим экзотическим языком под названием FALSE, Мюллер поставил перед собой цель создать минималистичный язык, и ему это определенно удалось. Его первоначальный интерпретатор для языка, состоящий всего из восьми команд, занимал лишь 240 байт. Еще более впечатляющим является интерпретатор Brainfuck, написанный на ассемблере x86, который занимает всего 69 байт[1]. Мы не добьемся 69 байт, но ядро нашего интерпретатора Brainfuck на Python будет состоять лишь из 25 строк кода и пары вспомогательных функций. И эти 25 строк будут способны запускать любую программу Brainfuck, а значит, любой компьютерный алгоритм.

Первоначальная реализация Brainfuck Мюллера имела некоторые ограничения, которые мы повторим в нашем интерпретаторе. Вместо неограниченной ленты, как в машине Тьюринга, исходный Brainfuck был ограничен количеством 30 000 ячеек. И каждая из этих ячеек могла содержать только 8-битное целое число без знака.

Реализация Brainfuck на Python

Прежде чем приступить к основной реализации интерпретатора, давайте немного разберемся с основными моментами. Каждый проект, который мы будем выполнять в этой книге, будет структурирован как пакет Python. Каждый пакет будет находиться в своем каталоге с файлом *__main__.py*, который инициирует выполнение при запуске проекта из командной строки. Весь код можно найти в репозитории GitHub книги по адресу *https://github.com/davecom/ComputerScienceFromScratch*. Кроме того, весь необходимый код вы также найдете непосредственно в этой книге, если не указано иное. Каждый листинг кода сопровождается именем связанного с ним файла Python, чтобы вам было проще найти код в репозитории книги.

---

[1] «bf core», Esolangs.org, посещение 22 мая 2024 г., *https://esolangs.org/wiki/Bf_core*.

Получение исходного файла

Наш файл *__main__.py* отвечает за прием аргумента командной строки, содержащего путь к исходному файлу Brainfuck, и передачу его основному интерпретатору:

```
# Brainfuck/__main__.py
from argparse import ArgumentParser
from Brainfuck.brainfuck import Brainfuck

if __name__ == "__main__":
    # Анализ аргумента файла
    file_parser = ArgumentParser("Brainfuck")
    file_parser.add_argument("brainfuck_file",
                             help="A file containing Brainfuck source code.")
    arguments = file_parser.parse_args()
    Brainfuck(arguments.brainfuck_file).execute()
```

Стандартный класс библиотеки `ArgumentParser` упрощает обработку аргументов командной строки. Мы будем использовать его во всех проектах этой книги. В этом фрагменте мы создаем один аргумент командной строки, `brainfuck_file`, который указывает путь к файлу, из которого мы хотим загрузить исходный код Brainfuck. Тип аргумента по умолчанию – строка, поэтому в конечном итоге мы будем передавать строку пути в наш класс `Brainfuck`, который будет отвечать за чтение его содержимого.

Примечание *Чтобы узнать больше об* `ArgumentParser`*, см. официальную документацию по* `argparse` *по адресу https://docs.python.org/3/library/argparse.html.*

Создание интерпретатора

Наш интерпретатор отвечает за обслуживание состояния Brainfuck (`cells`, `cell_index` и `instruction_index`). Он также отвечает за чтение каждой действительной команды Brainfuck в исходном файле и изменение состояния или выполнение операции ввода/вывода на основе этой команды. Поскольку команды Brainfuck состоят всего из одного символа, их чтение не представляет сложности. Фактические действия, которые необходимо выполнить с каждым символом, практически идентичны приведенным в табл. 1.1. В результате мы получаем всего одну функцию (`execute()`) из 25 строк кода.

```
# Brainfuck/brainfuck.py
```

```
from pathlib import Path

class Brainfuck:
    def __init__(self, file_name: str | Path):
        # Открыть текстовый файл и сохранить в переменной экземпляра
        with open(file_name, "r") as text_file:
            self.source_code: str = text_file.read()

    def execute(self):
        # Состояние установки
        cells: list[int] = [0] * 30000
        cell_index = 0
        instruction_index = 0
        # Продолжать, пока остаются потенциальные инструкции
        while instruction_index < len(self.source_code):
            instruction = self.source_code[instruction_index]
            match instruction:
                case ">":
                    cell_index += 1
                case "<":
                    cell_index -= 1
                case "+":
                    cells[cell_index] = clamp0_255_wraparound(cells[cell_index] + 1)
                case "-":
                    cells[cell_index] = clamp0_255_wraparound(cells[cell_index] - 1)
                case ".":
                    print(chr(cells[cell_index]), end='', flush=True)
                case ",":
                    cells[cell_index] = clamp0_255_wraparound(int(input()))
                case "[":
                    if cells[cell_index] == 0:
                        instruction_index = self.find_bracket_match(instruction_index, True)
                case "]":
                    if cells[cell_index] != 0:
                        instruction_index = self.find_bracket_match(instruction_index, False)
            instruction_index += 1
```

Выполнение каждой команды происходит в соответствии с правилами, приведенными в табл. 1.1, и состоит из простых действий с тремя переменными состояния. Для тех, кто не следит за последними версиями Python: оператор `match` был добавлен в Python 3.10 и может рассматриваться как эффективная версия оператора `switch` из других языков. Он запускает участок кода (или `case`), соответствующий значению в переменной, с которой производится сопоставление. В нашей программе `case` соответствует возможным значениям инструкции.

Подсказки по типам

Вероятно, вы уже обратили внимание на то, что я использую подсказки типов для некоторых локальных переменных. Я продолжу делать это на страницах данной книги в тех случаях, когда считаю, что это добавляет ясности, но не собираюсь следовать данному правилу слишком строго. Например, я считаю, что уточнение того, что `cells` является `list[int]`, имеет смысл в силу того, что некоторые люди могут не помнить используемый мной синтаксис инициализации списка. Но из контекста очевидно, что `instruction` будет `str`, поэтому я и не привел для него подсказку типа. Еще одно преимущество подсказок типов заключается в том, что они позволяют запускать проверку статических типов, чтобы облегчить проверку правильности всего кода в книге. Я всегда нахожу это очень полезным.

Краткое примечание о синтаксисе подсказки типа `file_name: str | Path`, используемом в сигнатуре `__init__()`: этот синтаксис означает, что предоставляемый аргумент должен быть либо типа `str`, либо типа `Path`. Оба типа допустимы. Наш `ArgumentParser` предоставляет пути к файлам в виде строк, а наши модульные тесты предоставляют их в виде объектов `Path`. Функция `open()`, которая использует путь в `__init__()`, может принимать любой из них.

В табл. 1.1 отсутствуют две вспомогательные функции: `find_bracket_match()` и `clamp0_255_wraparound()`. Для начала рассмотрим `find_bracket_match()`, которая помогает перейти от одной команды `if`-типа к ее партнеру. Эта функция реализована как метод класса `Brainfuck`, поскольку ей требуется доступ к `self.source_code`:

```
    # Поиск местоположения скобки, соответствующей скобке в *start*.
    # Если *forward* равно true, перейти вправо в поисках соответствующей скобки "]".
    # В противном случае сделать обратное.
    def find_bracket_match(self, start: int, forward: bool) -> int:
        in_between_brackets = 0
      ❶ direction = 1 if forward else -1
        location = start + direction
        start_bracket = "[" if forward else "]"
        end_bracket = "]" if forward else "["
        while 0 <= location < len(self.source_code):
          ❷ if self.source_code[location] == end_bracket:
                if in_between_brackets == 0:
                    return location
                in_between_brackets -= 1
          ❸ elif self.source_code[location] == start_bracket:
                in_between_brackets += 1
```

```
            location += direction
        # Совпадение не найдено
        print(f"Error: could not find match for {start_bracket} at {start}.")
        return start
```

Чтобы найти подходящую скобку, мы выполняем линейный поиск по исходному коду Brainfuck, просматривая каждый последующий символ по очереди. Мы ищем вправо, если `forward` имеет значение `True`, или влево, если `forward` имеет значение `False`. Переменная `direction` становится прокси для `forward`, увеличивая или уменьшая местоположение для перемещения вправо или влево ❶.

Сложность при поиске подходящей скобки заключается в *промежуточных скобках*, наборах скобок, которые встречаются между начальной скобкой и искомой скобкой. Например, предположим, что мы ищем подходящую скобку для первой скобки в этом фрагменте Brainfuck (для наглядности я пронумеровал символы):

```
[++[--]<<]
0123456789
```

Соответствием открывающей скобки с индексом 0 является закрывающая скобка с индексом 9. Однако если мы наивно примем первую найденную закрывающую скобку, наш поиск придет к выводу, что соответствием индексу 0 является закрывающая скобка с индексом 6. На самом деле эта закрывающая скобка соответствует открывающей скобке с индексом 3.

Решение заключается в том, чтобы просто пересчитывать промежуточные скобки. Каждый раз, когда мы встречаем начало пары промежуточных скобок, мы увеличиваем счетчик `in_between_brackets` ❸. Каждый раз, когда мы встречаем конец пары промежуточных скобок, мы уменьшаем счетчик `in_between_brackets`, если только `in_between_brackets` не равен `0`, что означает, что промежуточных скобок больше нет и конечная скобка найдена ❷.

Альтернатива поиску скобок

Другим способом решения проблемы с промежуточными скобками является использование стека. Каждый раз, когда встречается начальная скобка, ее положение помещается в стек. Каждый раз, когда встречается конечная скобка, стек стирается. Два полученных положения скобок (положение встреченной конечной скобки и стертой начальной скобки) составляют пару.

С помощью этого метода можно пройти весь исходный код за один раз и без труда найти все пары скобок. Затем местополо-

жения пар скобок можно кешировать с целью повышения производительности интерпретатора. Вместо линейного поиска, как в `find_bracket_match()` при каждом необходимом переходе, поиск другой скобки (местоположения перехода) сводится к простому поиску в кеше.

Еще одна вспомогательная функция, `clamp0_255_wraparound()`, имитирует исходный Brainfuck, ограничивая значения ячеек 8-битными целыми числами без знака. Эта функция нам понадобится по той причине, что тип int в Python имеет произвольную точность, то есть он может вмещать целые числа любого размера без переполнения (вместо этого при необходимости захватывается больше байтов). Настоящее 8-битное целое число без знака будет обнуляться, как только превысит 255 на 1, и будет возвращаться к 255, если оно было равно 0 и было уменьшено на 1. Мы имитируем это поведение в `clamp0_255_wraparound()` с помощью нескольких простых условий:

```
# Имитация 1-байтового целого числа без знака
def clamp0_255_wraparound(num: int) -> int:
    if num > 255:
        return 0
    elif num < 0:
        return 255
    else:
        return num
```

Поскольку Brainfuck не может изменять ячейку более чем на 1 за один раз, нам не нужно беспокоиться о случаях, когда мы добавляем более 1 к ячейке, равной 255, или вычитаем более 1 из ячейки, равной 0. Поэтому тестов `num > 255` и `num < 0` будет вполне достаточно.

С этими двумя вспомогательными функциями мы завершили разработку интерпретатора Brainfuck. На самом деле для реализации языка, полного по Тьюрингу, не требуется много усилий.

Запуск интерпретатора

Давайте попробуем запустить какой-нибудь код Brainfuck. В каталоге *Brainfuck* в репозитории этой книги есть вложенный каталог *Examples* с несколькими примерами программ для интерпретации, в том числе *fibonacci.bf* для генерации первых нескольких членов последовательности Фибоначчи и *hello_world_verbose.bf*, содержащий представленную ранее в этой главе программу «Hello World!». В данном случае я запускаю эти программы из главного каталога репозитория:

```
% python3 -m Brainfuck Brainfuck/Examples/fibonacci.bf
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89
% python3 -m Brainfuck Brainfuck/Examples/hello_world_verbose.bf
Hello World!
```

Эти команды необходимо выполнять с опцией `-m`, которая указывает, что Brainfuck следует рассматривать в качестве модуля. В противном случае вы получите ошибки при импорте. Обратите внимание, что способ доступа к Python из оболочки зависит от операционной системы и типа установки Python. В моей системе интерпретатор Python имеет псевдоним `python3`, а для путей используется косая черта. В вашей системе может использоваться `python` и обратные косые черты (в стиле Windows).

Похоже, наш интерпретатор работает, но для уверенности стоит создать несколько тестов.

Тестирование интерпретатора

Давайте составим несколько тестов для проверки правильности работы нашего интерпретатора. Для начала мы могли бы написать несколько *модульных тестов* (unit tests), чтобы убедиться, что каждая отдельная команда интерпретатора работает должным образом. Правильно ли работает `+`? Правильно ли работает `.`? Однако для краткости (и в силу простоты интерпретатора) мы вместо этого напишем несколько *интеграционных тестов* (integration tests). Эти тесты проверяют корректность работы всей программы Brainfuck с интерпретатором и получение ожидаемого результата.

Чтобы упростить настройку непрерывной интеграции, тесты для всей книги хранятся в собственном каталоге в корне основного репозитория, который называется *tests*. Наши тесты для Brainfuck запускают целые программы Brainfuck с помощью интерпретатора, захватывают их текстовый вывод и сравнивают его с заранее известным ожидаемым выводом.

tests/test_brainfuck.py

```
import unittest
import sys
from pathlib import Path
from io import StringIO
from Brainfuck.brainfuck import Brainfuck

# Токенизация, парсинг и интерпретация программы
# Brainfuck; сохранение результата в строке и его возврат
def run(file_name: str | Path) -> str:
    output_holder = StringIO()
    sys.stdout = output_holder
```

```
    Brainfuck(file_name).execute()
    return output_holder.getvalue()
```

Функция `run()` инициализирует класс `Brainfuck` с помощью файла, расположенного в `file_name`. Она также использует `output_holder` для захвата и возврата `stdout`, то есть вместо того, чтобы вывод программы run поступал в консоль, он будет присвоен переменной. Это дает нам возможность программного сравнения фактического вывода с ожидаемым выводом после вызова `run()` в каждом из наших тестов:

```
class BrainfuckTestCase(unittest.TestCase):
    def setUp(self) -> None:
        self.example_folder = (Path(__file__).resolve().parent.parent
                               / 'Brainfuck' / 'Examples')

    def test_hello_world(self):
        program_output = run(self.example_folder / "hello_world_verbose.bf")
        expected = "Hello World!\n"
        self.assertEqual(program_output, expected)

    def test_fibonacci(self):
        program_output = run(self.example_folder / "fibonacci.bf")
        expected = "1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89"
        self.assertEqual(program_output, expected)

    def test_cell_size(self):
        program_output = run(self.example_folder / "cell_size.bf")
        expected = "8 bit cells\n"
        self.assertEqual(program_output, expected)

    def test_beer(self):
        program_output = run(self.example_folder / "beer.bf")
        with open(self.example_folder / "beer.out", "r") as text_file:
            expected = text_file.read()
            self.assertEqual(program_output, expected)

if __name__ == "__main__":
    unittest.main()
```

Каждый тест берет программу Brainfuck из каталога *Examples*, использует `run()` для ее выполнения и сравнивает конечный результат с ожидаемым результатом с помощью `assertEqual()`. Попробуем запустить все тесты из главного каталога репозитория:

```
% python3 -m tests.test_brainfuck
....
----------------------------------------------------------------------
Ran 4 tests in 0.689s
OK
```

Если наш интерпретатор Brainfuck сможет успешно запустить четыре программы, которые значительно отличаются друг от друга, то есть большая вероятность, что он работает. В онлайн-репозитории этой книги я настроил непрерывную интеграцию, чтобы эти тесты автоматически запускались при каждом изменении кода. В большинстве глав данной книги также есть модульные или интеграционные тесты, которые запускаются автоматически.

Код в реальной жизни

Однажды давным-давно я впервые услышал о Brainfuck как о чем-то курьезном, но по-настоящему заинтересовался им в 2018 году, когда готовился к преподаванию курса «Новые языки» в колледже Champlain. Этот курс посвящен теории программирования с одной особенностью: мы используем языки, которые в 2018 году только становились актуальными в отрасли, а именно Go, Swift и Clojure. Они применялись для наглядной демонстрации программных идей. Я разработал курс вместе с коллегой по имени Джош Ауэрбах (Josh Auerbach). Я создал части курса, посвященные Go и Swift, а Джош разработал часть, посвященную Clojure.

Нам обоим понравилась идея выполнить какое-нибудь задание на Brainfuck, потому что это отличный образовательный инструмент для понимания работы простого интерпретатора. Джош предложил использовать часть задания по Brainfuck для обучения макросам Clojure. Мы использовали макрос Clojure для объяснения идеи *гомоиконичности* (homoiconicity) в Lisp (Clojure – это диалект Lisp), то есть изложения концепции «код – это данные». В макросе Clojure можно работать с кодом до его запуска, рассматривая его как любые другие данные в программе Clojure. Разработанный студентами в задании Джоша макрос позволяет писать код Brainfuck непосредственно в Clojure и запускать его на исполнение так, будто он принадлежит этой программе. Вы можете просто написать в середине своей программы Clojure что-то вроде этого:

```
(bf +++++++..+++.>>.<-.<.+++.------.--------.>>+.>++.)
```

Я до сих пор использую это задание в процессе преподавания (Джош ушел из академической среды), но, признаюсь, иногда мне трудно вспомнить синтаксис для написания макроса Clojure. Написать интерпретатор Brainfuck даже проще, чем макрос. Вот почему Brainfuck является замечательным инструментом для образовательных целей.

Практические приложения

Каждая глава книги заканчивается примерами практического применения, но, к сожалению, практического применения Brainfuck не существует. Этот язык вызывает любопытство и полезен для изучения некоторых фундаментальных идей в области информатики. Поэтому, возможно, можно сказать, что практическое применение Brainfuck сводится к образовательным целям.

Интерпретаторы в более широком смысле являются критически важной вычислительной инфраструктурой со множеством областей применения в реальной жизни. Как вы, вероятно, знаете, Python сам по себе является интерпретируемым языком. Существует много способов преобразования языков программирования из текстовых файлов в машинный код, но в целом мы можем разделить большинство реализаций языков программирования на интерпретируемые, компилируемые заранее (ранние компиляторы) или компилируемые в режиме реального времени. Для некоторых языков программирования существуют даже реализации всех трех категорий. Например, есть интерпретаторы Java, ранние компиляторы Java, а наиболее популярные реализации Java компилируются в режиме реального времени.

Как правило, интерпретируемые реализации языков программирования работают медленнее, чем скомпилированные. В связи с этим может возникнуть вопрос, почему же языки программирования, которые используются в реальной жизни, реализуются в виде интерпретаторов. Возьмем, к примеру, Python: почему он реализован с использованием интерпретатора, а не компилятора? Все мы знаем, что Python – относительно медленный язык, и, безусловно, он работал бы быстрее, если бы был скомпилирован.

Ответ заключается в том, что многие динамические функции Python не были бы возможны или по крайней мере были бы очень сложны для реализации в чем-либо, кроме интерпретатора. Есть попытки сделать это (например, PyPy), но они гораздо сложнее в плане реализации.

Кроме того, интерпретаторы гораздо проще в реализации, нежели компиляторы, поскольку в них отсутствует вся бэкенд-фаза компилятора, отвечающая за генерацию машинного кода. По этой причине многие языки программирования изначально являются интерпретируемыми, так как это самый быстрый способ запустить их в работу. Например, первая версия Java была интерпретируемой, и понадобилось несколько лет, прежде чем появилась версия с компиляцией «just-in-time».

Таким образом, интерпретаторы существуют, потому что их проще реализовать, чем компиляторы, и потому что они обеспечивают определенные эффективные динамические функции выполнения.

Если вы думаете о реализации нового языка программирования, особенно динамического, проще всего будет начать с интерпретатора.

Упражнения

1. Напишите *транспайлер* (или транспилятор – transpiler) с Brainfuck на Python. Транспайлер подобен компилятору, но вместо преобразования исходного кода, написанного на языке высокого уровня, в машинный код он преобразует исходный код с одного языка высокого уровня на другой язык высокого уровня. Вы можете повторно использовать большую часть структуры интерпретатора Brainfuck. Вместо выполнения каждой команды Brainfuck вы можете вывести эквивалентный код Python в список строк. Конечным результатом вашей программы должен быть сохраненный в файле эквивалентный код Python. Самая сложная часть будет заключаться в том, чтобы понять, что делать со скобками.
2. Добавьте в интерпретатор режим отладки, который позволит вам проходить программу Brainfuck по одной команде за раз. После каждой команды на консоль выводится таблица, содержащая все состояния интерпретатора Brainfuck, подобно таблицам в начале этой главы для прохождения программы «repeat».
3. Напишите программу Brainfuck, которая считывает два числа, сравнивает их и выводит большее число. Напишите тест на Python, который подтверждает, что программа работает правильно со случайно сгенерированными числами. Совет: возможно, вам понадобится изменить `sys.stdin` по аналогии с тем, как мы изменили `sys.stdout` в функции `run()` для тестов.

2

СОЗДАНИЕ ИНТЕРПРЕТАТОРА ЯЗЫКА BASIC



В главе 1 мы создали простой интерпретатор для минималистичного экзотического языка Brainfuck. Однако Brainfuck – это всего лишь игрушка; хотя мы и *могли бы* решать реальные задачи с его помощью, на практике это *вряд ли* было бы целесообразно. Существует множество других языков программирования, которые по сложности не сильно отличаются от Brainfuck, но при этом являются «реальными» в том смысле, что обычные программисты используют (или использовали) их в своей повседневной работе. В этой главе мы создадим интерпретатор для одного из таких языков, NanoBASIC, и в процессе более детально познакомимся с принципами работы интерпретаторов.

В то время как Brainfuck имеет всего восемь команд, NanoBASIC, упрощенный диалект BASIC, имеет всего шесть типов операторов. Справедливости ради, каждый из этих операторов обладает большей функциональностью, по сравнению с командой Brainfuck, но все же это не так много. Он достаточно сложен для того, чтобы исследовать несколько аспектов интерпретатора, которые были объединены в нашей реализации Brainfuck. В частности, мы создадим отдельный токенизатор, парсер и среду выполнения, в то время как наш интерпретатор Brainfuck обрабатывал все три задачи одновременно.

Мы будем использовать масштабируемый подход для каждого компонента, а это значит, что наши действия могут быть расширены для работы с более серьезными языками.

Основы NanoBASIC

Благодаря простому синтаксису и повсеместному распространению язык BASIC (универсальный символьный код инструкций для начинающих – Beginner's All-purpose Symbolic Instruction Code) способствовал демократизации компьютерного мира и стал де-факто стандартным языком революции в области персональных компьютеров. NanoBASIC – это версия языка программирования BASIC, созданная на основе популярного диалекта для микрокомпьютеров 1970-х годов, известного как Tiny BASIC. Версия NanoBASIC еще проще (или меньше, если хотите), чем Tiny BASIC, отсюда и название Nano.

NanoBASIC почти полностью идентичен Tiny BASIC, но я внес несколько изменений: в нем отсутствует пара операторов, а также есть некоторые незначительные различия в именах переменных и ширине целых чисел. По мере изучения языка вы, возможно, удивитесь некоторым его необычным синтаксическим особенностям или ограничениям. Эти особенности являются результатом намеренного решения, поскольку NanoBASIC в целом должен быть совместим с Tiny BASIC. В конце главы вы сможете взять реальные программы на Tiny BASIC из интернета и запустить их в нашем интерпретаторе NanoBASIC. Таким образом, вы сможете применять язык, который действительно использовался в реальной жизни.

Все, что вам нужно знать о NanoBASIC, можно выучить всего за несколько минут. Помните, сколько времени у вас ушло на изучение Python? К концу этого раздела вы будете полностью готовы к написанию программ на NanoBASIC.

История BASIC

Изначально язык BASIC был разработан в 1964 году Джоном Кемени (John Kemeny) и Томасом Курцем (Thomas Kurtz) в колледже Дартмута с целью сделать компьютеры более доступными, в том числе для студентов, не изучающих естественные науки или математику[1]. Фактически именно студенты бакалавриата вместе с Кемени и Курцем разработали первую реализацию BASIC. Когда в середине 1970-х годов началась революция в области персональных компьютеров, BASIC естественным образом подошел для любителей и других «обычных» пользователей, которые купили свои первые машины. В результате он был самым популярным языком программирования высокого уровня для персональных компьютеров с середины 1970-х до середины 1980-х годов. Обычные компьютеры той эпохи, такие как Commodore 64 и Apple II, поставлялись со встроенными интерпретаторами BASIC. Таким образом, BASIC был средством взаимодействия многих

[1] «BASIC Begins at Dartmouth», BASIC at 50, 2014, доступ 22 мая 2024 г., *https://www.dartmouth.edu/basicfifty/basic.html*.

людей с персональными компьютерами раннего периода. Например, BASIC был первым языком программирования Линуса Торвальдса (Linus Torvalds) на Commodore VIC-20 в 1981 году[1].

Интересен тот факт, что компания Microsoft начала свою деятельность в 1975 году, когда Билл Гейтс (Bill Gates) и Пол Аллен (Paul Allen) разработали интерпретатор BASIC для одного из первых персональных компьютеров Altair 8800[2]. Успех их компании пришел после того, как они портировали свой интерпретатор на другие машины конца 1970-х годов. Microsoft BASIC поставлялся со многими персональными компьютерами и стал де-факто стандартным диалектом BASIC. В конце концов, в 1981 году Microsoft вошла в бизнес операционных систем с DOS на оригинальном IBM PC, но именно BASIC был стартом компании. Теперь и вы будете разрабатывать интерпретатор BASIC!

В свою очередь, язык Tiny BASIC, одним из вариантов которого является NanoBASIC, появился благодаря компании Microsoft. Многие из тех, кто был вовлечен в процесс ранней разработки Tiny BASIC, отчасти исходили из соображений высокой стоимости интерпретаторов Microsoft[3]. Некоторые из них также считали, что люди должны иметь право свободно делиться программным обеспечением по своему усмотрению. Это была ранняя форма движения за бесплатное программное обеспечение. Помимо желания обойти высокие расценки Microsoft, разработчики также хотели создать достаточно компактный язык, который бы помещался в крайне ограниченной памяти микрокомпьютеров того времени (часто всего 4 килобайта [КБ]) и был достаточно портируемым для запуска программ на различных типах машин. В конечном итоге Tiny BASIC был портирован на широкий спектр различных персональных компьютеров с разными микропроцессорными архитектурами и получил широкое распространение.

Парадигма, синтаксис и семантика NanoBASIC

Как вы уже успели прочитать, BASIC был разработан с целью облегчить его использование для людей, не имеющих технического образования, и многие версии BASIC были созданы для работы в условиях ограниченного объема памяти. Поэтому BASIC принято считать упрощенным языком с относительно небольшим набором функций даже по сравнению с другими языками того времени. Создаваемый нами диалект NanoBASIC является императивным, но его едва ли можно назвать процедурным. Давайте разберемся, что означают эти термины.

---

1 Linus Torvalds and David Diamond, Just for Fun (HarperCollins, 2001), 7–8.

2 James Wallace and Jim Erickson, Hard Drive: *Bill Gates and the Making of the Microsoft Empire* (HarperBusiness, 1993).

3 Tom Pittman, «Itty Bitty Computers & Tiny Basic», Itty Bitty Computers, 2004, последнее изменение 10 июля 2017 г., доступ 22 мая 2024 г., *http://www.ittybittycomputers.com/IttyBitty/TinyBasic*.

Императивным (imperative) называют язык, где вы даете компьютеру подробные инструкции о действиях по выполнению задачи. Это контрастирует с *декларативными* (declarative) языками, которые сосредоточены на вопросе «что» вы хотите сделать, а не «как» вы хотите это сделать. Допустим, я хочу, чтобы вы нарисовали квадрат в центре листа миллиметровой бумаги. Императивный способ сделать это – сказать вам: «Начните с точки (4, 4) и проведите линию вверх на пять единиц. Затем проведите линию вправо на пять единиц. Потом проведите линию вниз на пять единиц. Затем проведите линию влево на пять единиц». Декларативный способ сделать это – сказать: «Нарисуйте квадрат 5×5 в середине листа». В декларативном стиле я заявляю, что я хочу, и позволяю вам (или компьютеру) разобраться в деталях того, как это сделать.

Современные императивные языки программирования обычно делятся на два основных подвида: процедурные и объектно-ориентированные. *Процедурные* (procedural) языки программирования используют подпрограммы/процедуры/функции (эти термины часто, но не всегда, используются как взаимозаменяемые) в качестве основных элементов абстракции. Код разбивается на несколько функций, каждая из которых имеет определенное назначение и работает согласованно с другими, образуя целостную программу. *Объектно-ориентированное* (object-oriented) программирование подразумевает использование объектов в качестве основного средства абстракции, и, поскольку вы являетесь программистом Python среднего или продвинутого уровня, я полагаю, что вы знаете, что это означает. Конечно же, на Python можно программировать в любом из этих стилей.

Примечание *Наиболее популярными подвидами декларативного программирования являются функциональное (functional) и логическое (logic). Подробное рассмотрение этих подвидов выходит за рамки данной главы.*

NanoBASIC явно не является декларативным языком программирования. Он определенно относится к императивному лагерю. Он также определенно не является объектно-ориентированным. Но является ли он процедурным? Хотя технически он имеет способ вызова подпрограммы с помощью оператора `GOSUB`, который вскоре будет описан, в нем нет ничего, напоминающего современную функцию в плане наличия параметров и возвращаемых значений. Вот почему я написал, что мы «едва ли можем назвать его процедурным».

Некоторые версии BASIC, такие как Tiny BASIC и NanoBASIC, не только не используют функции, но и не имеют циклов или других современных структур управления. Вместо этого все управление осуществляется с помощью `GOTO` и `GOSUB`, которые вызывают прямой переход к определенному номеру строки программы. В сочетании с операторами `if` это единственный способ управления программой

Tiny BASIC или NanoBASIC. Ранние версии BASIC были известны тем, что поощряли образование так называемого «спагетти-кода» из-за этих явных переходов из одной части программы в другую, а также из-за неэффективных механизмов организации. Эта критика вполне справедлива. Без функций или объектов в качестве механизмов организации императивный язык неизбежно превращается в спагетти-код. Не удивляйтесь, увидев немного спагетти в процессе приготовления NanoBASIC!

Теперь давайте рассмотрим синтаксис NanoBASIC и принципы работы его шести операторов.

Комментарии и номера строк

Комментарии в NanoBASIC начинаются с обозначения REM и могут заканчиваться любой строкой. Комментарии не обрабатываются интерпретатором.

Любая строка в NanoBASIC, не являющаяся комментарием, начинается с номера строки, за которым следует оператор. Программист может выбирать любые номера строк, лишь бы они были расположены в порядке возрастания от начала до конца исходного файла. Например, следующие номера строк являются допустимыми:

```
10 PRINT "Hello"
REM Это комментарий
20 PRINT "Goodbye"
30 PRINT "WOW"
```

Напротив, эти номера строк являются недопустимыми, и поэтому в данном случае поведение программы не определено:

```
10 PRINT "Hello"
REM Это комментарий
40 PRINT "Goodbye"
30 PRINT "WOW"
```

Программы, содержащие операторы GOTO или GOSUB и несоответствующие номера строк, будут работать некорректно.

В NanoBASIC существует только шесть способов запуска оператора: PRINT, IF, GOTO, GOSUB, RETURN и LET. Если вы знаете эти шесть типов операторов, вы в принципе знаете весь язык. Именно поэтому, если вы уже знаете другой язык программирования, вы сможете освоить NanoBASIC всего за несколько минут.

LET, переменные и математические выражения

Оператор LET связывает значение с переменной. Все переменные представляют собой целые числа. Другие типы переменных отсут-

ствуют. В исходном Tiny BASIC использовались только однобуквенные имена переменных (от A до Z). В NanoBASIC это ограничение снято и можно использовать идентификаторы любой длины, включающие буквы и подчеркивания. Следующий оператор устанавливает значение переменной A равным 5:

```
10 LET A = 5
```

За ключевым словом LET должно следовать имя переменной и знак равенства (=). После этого можно использовать любое математическое выражение. Математические выражения NanoBASIC могут состоять из переменных, целочисленных литералов, операторов сложения (+), вычитания (-), умножения (*) и деления (/), а также скобок ((и)). Кроме того, в NanoBASIC можно отрицать любое математическое значение с помощью знака минус (-). Все математические вычисления выполняются в области целых чисел со знаком. С этого момента мы будем называть математические выражения просто выражениями. Ниже приведены все допустимые варианты использования LET:

```
20 LET B = A
30 LET C = 23 - A
40 LET D = 5 * (24 + 25)
50 LET E = -(24 + 23 - (2 * (5 + 3)))
```

Из-за ограничений вычислительной техники большинство реализаций Tiny BASIC ограничивались 16-битными целыми числами. Наши переменные основываются на целых числах Python и имеют произвольную точность, поэтому они не ограничены 16 битами. Это, а также имена переменных произвольной длины, является одним из немногих преимуществ NanoBASIC по сравнению с Tiny BASIC, вместо того чтобы быть просто одним из его подвидов.

Операторы PRINT

Любой строковый литерал или выражение может быть выведено на консоль с помощью PRINT. Строковые литералы NanoBASIC – это любые символы, заключенные в двойные кавычки ("). К сожалению, в NanoBASIC нет возможности включать в строки настоящие двойные кавычки. Другими словами, нет механизма экранирования. Я оставлю это в конце главы в качестве упражнения. Вот несколько допустимых операторов PRINT со строковыми литералами:

```
10 PRINT "Какая хорошая программа "
20 PRINT "Кто разрешил садиться?"
30 PRINT "6734 означает HELP типа вверх ногами"
```

Как уже упоминалось, PRINT также может выводить результат любого выражения:

```
REM Это было первое, что Пол Аллен запустил на Altair 8800
70 PRINT 2 + 2
```

Вы также можете передать в PRINT список элементов (строковые литералы и выражения), разделенных запятыми. Все выводимые элементы будут разделены символами табуляции, а PRINT всегда завершает вывод вставкой символа новой строки. Например:

```
30 PRINT "2 плюс 2 равно", 2 + 2, "а 3 умножить на 5 равно", 3 * 5
```

В результате будет выведен следующий текст:

```
2 плюс 2 равно 4, а 3 умножить на 5 равно 15
```

Обратите внимание, что пробелы между выражениями обусловлены символами табуляции. Из-за различных настроек консоли на вашем терминале они могут выглядеть иначе.

Условные операторы IF и булевы выражения

Условные операторы IF в NanoBASIC похожи на условные операторы `if` в других языках, но они проще и лаконичнее. Они могут иметь только одно булево выражение (в них нет, например, оператора `and` или `or`), и в них нет спецификатора (клаузулы) `else`. Наконец, они могут исполнить только одно выражение, если они истинны. Выражение, которое должно быть исполнено в случае истинности, всегда предваряется литералом THEN. Например:

```
500 IF N < 10 THEN PRINT "Небольшое число"
700 IF V >= 34 THEN GOTO 20
```

Булевы выражения по большей части включают в себя ожидаемые сравнения, но операторы немного отличаются от стандартных операторов стиля C. Например, *неравенство* в NanoBASIC может быть обозначено как <> или ><, а *равенство* обозначается как =, а не ==.

Операторы GOTO, GOSUB и RETURN

Оператор GOTO напрямую переходит к номеру строки без возможности вернуться назад. Оператор GOSUB переходит к номеру строки, но соответствующий оператор RETURN возвращает программу к строке сразу после того места, где изначально был вызван GOSUB. Вот конкретный пример:

```
10 GOTO 50
20 LET A = 10
40 RETURN
50 LET A = 5
60 GOSUB 20
REM RETURN возвращает сюда; мы ожидаем, что A будет равным 10
70 PRINT A
```

Эта программа в конечном итоге выведет на консоль число `10`.

Стиль и особенности NanoBASIC

Обычно считается хорошим тоном писать ключевые слова BASIC заглавными буквами. Поскольку NanoBASIC не имеет особых средств организации, также рекомендуется включать в программу комментарии, объясняющие происходящее.

К сожалению, код BASIC обычно быстро заполняется операторами `GOTO` и превращается в «спагетти-код». Это нормально, и в NanoBASIC вы мало что можете с этим поделать, если хотите написать программу средней сложности. В конце концов, благодаря недовольству стилем программирования `GOTO` возникло движение за *структурированное программирование*.

Так, к примеру, Эдсгер Дейкстра (Edsger Dijkstra) прославился своим письмом под названием «Оператор Go To считается вредным» (Go To Statement Considered Harmful)[1].

Вот еще несколько особенностей NanoBASIC, о которых вам следует знать:

- NanoBASIC не учитывает регистр символов. Это означает, что `LET A = 5` и `let a = 5` – это одно и то же;
- поведение, не описанное в этой главе, является неопределенным.

В конечном счете NanoBASIC является преднамеренно ущербным продуктом, потому что я хотел сохранить простоту интерпретатора и предоставить реальный аналог – Tiny BASIC, чтобы придать работе в этой главе более «реальный» характер. Я также полагаю, что сам факт создания NanoBASIC на основе реального языка и его способность запускать реальные программы Tiny BASIC из интернета делают работу по созданию интерпретатора более увлекательной. Однако было бы совсем несложно сделать язык более мощным. Оставим это для упражнений.

[1] Edsger Dijkstra, «Letters to the Editor: Go To Statement Considered Harmful», Communications of the ACM 11, № 3 (1968): 147–148, *https://dl.acm.org/doi/10.1145/362929.362947*.

Пример программы NanoBASIC

Несколько примеров программ NanoBASIC включены в каталог *NanoBASIC/Examples* в сопутствующем репозитории. Одна из этих программ выводит все числа из последовательности Фибоначчи, которые меньше 100. Последовательность Фибоначчи – это прогрессивная последовательность чисел, в которой каждое число (кроме двух первых) является суммой двух предыдущих. Она начинается с чисел 0 и 1. Затем следует, что 0 + 1 = 1, поэтому следующее число в последовательности – это 1. Потом 1 + 1 = 2, поэтому следующее число – это 2. Затем идет 3, 5, 8, 13 и так далее.

Вот так выглядит программа *fib.bas*, реализующая последовательность Фибоначчи на NanoBASIC:

NanoBASIC/Examples/fib.bas

```
REM Вывод чисел Фибоначчи меньше 100
REM A – это последнее число
10 LET A = 0
REM B – это следующее число
11 LET B = 1
20 PRINT A
21 PRINT B
REM C – это последнее + следующее
30 LET C = A + B
31 LET A = B
32 LET B = C
40 IF B < 100 THEN GOTO 21
```

В соответствии со стилем Tiny BASIC в качестве имен переменных мы используем только заглавные буквы. Это подчеркивает важность наличия большого количества комментариев. Как уже упоминалось ранее, выбранные значения номеров строк являются произвольными, главное, чтобы они были расположены в порядке возрастания. В строках 10 и 11 мы начинаем последовательность с жестко заданных начальных значений `0` и `1`. В строке 30 мы формируем следующее число в последовательности, C, путем сложения двух предыдущих чисел. Строки с 21 по 40 составляют своего рода цикл с использованием `IF` и `GOTO` в строке 40. Некоторые более поздние версии Tiny BASIC включали в себя настоящие операторы цикла, такие как `FOR`, но самые ранние версии выполняли все циклы с использованием синтаксиса, похожего на этот, во многом напоминающего работу циклов в большинстве языков ассемблера.

Когда вы дочитаете главу, вы сможете запустить эту программу самостоятельно с помощью следующей команды:

```
% python3 -m NanoBASIC NanoBASIC/Examples/fib.bas
0
1
```

```
1
2
3
5
8
13
21
34
55
89
```

Все выглядит правильно.

Мы почти готовы к созданию реализации NanoBASIC, но прежде чем приступить к этому, важно формализовать синтаксис языка. Мы можем напрямую использовать эту спецификацию для написания нашей реализации.

Формализация синтаксиса NanoBASIC

Синтаксис языка программирования формально определяется *грамматикой*. Форма Бэкуса–Наура (Backus–Naur form, BNF) – это типичный способ определения грамматики языка программирования. Существует много расширений и дополнений BNF; но мы будем использовать форму, которая, на мой взгляд, будет наиболее понятна программистам среднего уровня, поскольку она включает в себя синтаксис, похожий на регулярные выражения.

Грамматика состоит из набора *продукционных правил* (production rules), которые определяют допустимый синтаксис языка программирования. Термин «продукционное правило» звучит замысловато, но это просто способ замены одного элемента другим. Допустим, я создаю грамматику для языка, который может состоять только из букв `A` и `B` и цифр `1` и `2`. Его продукционные правила могут выглядеть следующим образом:

```
<expression> ::= (<letter> | <number>)*
<letter> ::= 'A' | 'B'
<number> ::= '1' | '2'
```

Идентификатор, заключенный в угловые скобки, например `<expression>`, является *нетерминальным* (non-terminal) символом. Это элемент грамматики, который при расширении заменяется чем-то другим. То, чем он заменяется, указывается в правой части правила производства. В правиле производства символ `::=` отделяет нетерминальный символ от его замены. Замена может состоять из нетерминальных или терминальных символов.

Терминальный (terminal) символ – это то, что будет появляться в языке в своей непосредственной форме. Он не подлежит дальнейшему расширению. В нашем синтаксисе терминальный символ заключается в одинарные кавычки, например `'A'`. В нашем синтаксисе также используется символ |, означающий *or* («или»). Символ *or* означает, что для этой части правила есть несколько вариантов на выбор. Для группировки мы используем скобки, а символ * означает ноль или больше повторений чего-либо.

С учетом этого мы можем прочитать три правила производства, показанных ранее, как означающие:

1. *Выражение* состоит из нуля или более букв или цифр.
2. *Буква* (letter) – это A или B.
3. *Цифра* (number) – это 1 или 2.

Мы можем использовать грамматику для проверки того, является ли определенная строка текста действительным синтаксисом для языка, просто следуя его правилам производства. Например, наша грамматика определяет, что «AAA21B» является допустимым синтаксисом, а «AB123» – не является: несколько грамматик могут определять один и тот же язык. Имена нетерминальных символов по большей части являются произвольными и должны выбираться таким образом, чтобы иметь максимальный смысл для восприятия человеком. Например, нет никакой причины разделять буквы и цифры в нашей грамматике. Мы могли бы упростить грамматику до следующего вида:

```
<expression> ::= <character>*
<character> ::= 'A' | 'B' | '1' | '2'
```

Мы могли бы даже полностью исключить второе правило производства:

```
<expression> ::= ('A' | 'B' | '1' | '2')*
```

Как правило, не следует использовать ненужные правила производства, поскольку они чрезмерно усложняют грамматику. Тем не менее если дополнительный нетерминальный символ представляет несколько возможных терминальных символов и будет использоваться в другом месте грамматики, имеет смысл присвоить ему собственное правило производства вместо дублирования длинного списка терминальных символов. Это станет более понятным, когда мы столкнемся с более обширными грамматиками с более сложной структурой. Это аналогично программированию, где лучше иметь много небольших функций, которые мы повторно используем, чем несколько крупных.

Рассмотрим другой пример. Допустим, мне нужно определить правила производства для нумерованного списка. Они могут выглядеть следующим образом:

```
<list> ::= <item>*
<item> ::= <number>'.' <text>'\n'
<number> ::= <digit><digit>*
<digit> :: = '0' | '1' | ... | '8' | '9'
<text> ::= .*
```

Теперь мы ввели еще пару специальных форм. Символ … обозначает, что список терминальных символов продолжается подразумеваемым образом (это не очень формально, но экономит место), а . просто означает любой терминальный символ, который может представить себе пользователь, например, в регулярном выражении. Давайте снова сформулируем пять правил этой грамматики в более понятной форме:

1. *Список* (list) состоит из нуля или более элементов.
2. *Элемент* (item) – это число, за которым следует точка, некоторый текст и новая строка.
3. *Число* (number) – это одна или несколько цифр.
4. *Цифра* (digit) – это один из символов от 0 до 9.
5. *Текст* (text) – это любая произвольная строка.

Заметили ли вы проблему в этой грамматике, касающуюся способа обработки чисел? Число с ведущими нулями, например 0020, будет допустимо, но это не будет иметь смысла в нумерованном списке. Как это можно исправить? Я оставлю это в качестве упражнения для читателя. Если вы сможете это исправить, то, вероятно, уже достаточно хорошо понимаете терминальные и нетерминальные символы.

Грамматики, описанные с помощью BNF, называются *контекстно-свободными* (context free), а это значит, что каждое правило производства может самостоятельно обозначать нетерминальный символ, который встречается в более длинной строке. Даже без контекста остальной части строки правило производства все равно может быть расширено для одного конкретного нетерминального символа. Другими словами, в контекстно-свободной грамматике каждый нетерминальный символ не зависит от других нетерминальных символов вокруг него при его расширении.

Грамматика NanoBASIC основана на исходной грамматике Tiny BASIC, опубликованной Деннисом Эллисоном (Dennis Allison), создателем первой реализации Tiny BASIC, в 1976 году[1]. Она выглядит следующим образом:

```
❶ <line> ::= <number> <statement> '\n' | 'REM' .*'\n'

❷ <statement> ::= 'PRINT' <expr-list> |
```

[1] Dennis Allison, «Design Notes for Tiny BASIC», Dr. Dobb's Journal of Computer Calisthenics and Orthodontia 1 (1976): 9, *https://archive.org/details/dr_dobbs_journal_vol_01/page/n9/mode/2up*.

```
  'IF' <boolean-expr> 'THEN' <statement> |
  'GOTO' <expression> |
  'LET' <var> '=' <expression> |
  'GOSUB' <expression> |
  'RETURN'

❸ <expr-list> ::= (<string> | <expression>) (',' (<string> | <expression>))*

❹ <expression> ::= <term> (('+'|'-') <term>)*

❺ <term> ::= <factor> (('*'|'/') <factor>)*

❻ <factor> ::= ('-'|ε) <factor> | <var> | <number> | '('<expression>')'

<var> ::= ('_'|<letter>) ('_'|<letter>)*

<number> ::= <digit> <digit>*

<digit> ::= '0' | '1' | ... | '8' | '9'

<letter> ::= 'a'|'b'| ... |'y'|'z'|'A'|'B'| ... |'Y'|'Z'

<relop> ::= '<' ('>'|'='|ε) | '>' ('<'|'='|ε) | '='

❼ <boolean-expr> ::= <expression> <relop> <expression>

<string> ::= '"' .* '"'
```

Единственным новым синтаксическим элементом в полной грамматике NanoBASIC является символ эпсилон (ε). Он означает, что в месте его появления может не быть ничего («пустота»). Он всегда появляется как часть or, обозначая тем самым, что может быть что-то или может не быть ничего.

Грамматика NanoBASIC выглядит гораздо сложнее, чем в двух предыдущих примерах – в конце концов, это целый язык программирования, но на практике разобраться в ней довольно несложно:

1. *Строка* (line) – это либо число (номер строки), за которым следует оператор, либо комментарий (REM предшествует всем комментариям) ❶.
2. *Оператор* (statement) – это один из шести операторов, которые мы уже узнали (PRINT, IF, GOTO, LET, GOSUB, RETURN) ❷. Здесь мы впервые сталкиваемся с рекурсией: оператор IF содержит другой оператор в своем предложении THEN, поэтому после THEN может появиться любой из шести операторов.
3. *Список выражений* (expression list – expr-list) – это разделенный запятыми список строк или выражений ❸. Как видно из раздела PRINT в предыдущем производственном правиле, списки выражений используются только для операторов PRINT. Это также единственное правило, связанное со строками. Поэтому строка может использоваться только как часть списка выражений опе-

ратора PRINT. Мы называем его списком, но на самом деле список выражений может содержать только одно выражение или строку. Если их больше, мы используем грамматический элемент *, который присоединяется к запятой и следующему выбору выражения или строки.

4. *Выражение* (expression) имеет отношение к арифметике. Это может быть сложение нескольких чисел, умножение нескольких чисел или просто извлечение значения из переменной. Само правило производства выражений включает только возможность сложения или вычитания ❹.
5. Выражение состоит из *термов* (terms). В то время как правило генерации выражений обрабатывает сложение и вычитание, правило генерации термов обрабатывает умножение и деление ❺. Причина этого связана с приоритетом: чем «глубже» мы спускаемся по лестнице нетерминальных символов, тем выше приоритет наших операторов при финальном воплощении этой грамматики в рабочий язык. Вот почему умножение и деление идут после сложения и вычитания.
6. *Приоритет* (precedence) также является причиной того, что скобки встречаются в правиле производства для множителя ❻, но не в правилах производства для выражений или терминов. Скобки имеют самый высокий приоритет среди всех арифметических операторов. Другими элементами, которыми мы можем заменить множитель, являются переменная (которая во время выполнения будет извлекать свое значение), числовой литерал или отрицание (вариант с ведущим знаком -).
7. Правила для *переменных* (variables), *чисел* (numbers), *цифр* (digits), *букв* (letters), *отношений* (relational operators, relops) и *строк* (strings) в основном не требуют пояснений. Обратите внимание на то, как просто расширить список допустимых идентификаторов переменных: мы разрешаем использовать один или несколько символов подчеркивания либо букв, в отличие от оригинального Tiny BASIC, где разрешалось применять только отдельные буквы. Первые ПК действительно имели ограниченный объем памяти, если им приходилось ограничивать нас только 26 однобуквенными именами переменных.
8. *Булево выражение* (Boolean expression) – это просто два числовых выражения с реляционным оператором между ними ❼. Поскольку в NanoBASIC нет операторов and и or, здесь нет необходимости в специальной форме *, как это было принято для арифметических операций, таких как сложение и умножение.

Эта грамматика служит основой для реализации токенизатора и парсера нашего интерпретатора. И если вы вспомните из главы 1, что это за компоненты, то теперь сможете понять то, как терми-

нальные символы в грамматике станут токенами, которые читает наш токенизатор. И вот что еще важнее: правила производства для нетерминальных символов в конечном итоге будут сопоставляться с функциями в нашем рекурсивном нисходящем парсере. Мы вернемся к этому чуть позже. В конечном счете грамматика определяет *синтаксис* языка программирования, но не придает *значение* каждому элементу языка. Это будет магия нашего интерпретатора.

Реализация NanoBASIC

Теперь, после знакомства с NanoBASIC и его синтаксисом, пришло время перейти к разработке его реализации. Как вы помните из главы 1, базовый интерпретатор состоит минимум из трех частей:

- *токенизатор* (tokenizer), иногда также называемый *лексером* (lexer – то есть *лексический анализатор*), который принимает исходный код и разбивает его на мельчайшие распознаваемые элементы, допускаемые в этом языке программирования. Эти элементы называются *токенами* (tokens). Для кода `a + 2` токенами могут быть `a`, `+` и `2`;
- *синтаксический анализатор*, или *парсер* (parser), который принимает расположенные рядом токены и определяет их значение (то есть образуемые ими выражения или операторы). Парсеры обычно создают дерево узлов, представляющее относительные отношения между выражениями, операторами и литеральными значениями. Это дерево называется *деревом абстрактного синтаксического анализа* (Abstract Syntax Tree, AST). Например, если интерпретатор Python увидел токен `a`, за которым следует токен `+`, а за ним токен `2`, он может построить узел арифметического выражения и соединить его с узлами для `a` и `2`;
- *среда выполнения* (runtime environment), которая проходит по узлам AST и выполняет соответствующие операции для реализации заложенного в них значения. Для нашего узла арифметического выражения `a + 2` это будет означать поиск значения, представленного `a`, и добавление к нему `2`.

Мы будем создавать эти три компонента по порядку, но перед тем, как перейти к токенизатору, нам нужно научиться открывать файл кода NanoBASIC:

NanoBASIC/__main__.py

```
from argparse import ArgumentParser
from NanoBASIC.executioner import execute

if __name__ == "__main__":
    # Разбор (парсинг) аргумента файла
    file_parser = ArgumentParser("NanoBASIC")
```

```
    file_parser.add_argument("basic_file",
                             help="A text file containing NanoBASIC code.")
    arguments = file_parser.parse_args()
    execute(arguments.basic_file)
```

Мы загружаем файл исходного кода по аргументу командной строки, так же как мы делали в нашем интерпретаторе Brainfuck, и передаем ему функцию execute(). Эта функция находится в отдельном файле, чтобы ее было проще использовать в наших тестах. Она объединяет компоненты токенизатора, парсера и интерпретатора (среда выполнения). Вывод одного компонента подается в качестве ввода для другого (исходный код ▸ токенизатор ▸ парсер ▸ интерпретатор):

NanoBASIC/executioner.py

```
from pathlib import Path
from NanoBASIC.tokenizer import tokenize
from NanoBASIC.parser import Parser
from NanoBASIC.interpreter import Interpreter

def execute(file_name: str | Path):
    # Загрузка текстового файла из аргумента
    # Токенизация, парсинг и выполнение
    with open(file_name, "r") as text_file:
        tokens = tokenize(text_file)
        ast = Parser(tokens).parse()
        Interpreter(ast).run()
```

Каждая строка кода в `execute()` переводит нас от одного основного компонента интерпретатора к следующему. Результат работы токенизатора поступает в парсер, а результат работы парсера – в среду выполнения. В оставшейся части главы мы будем поочередно создавать каждый из этих компонентов.

Токенизатор

Токенизатор принимает строку исходного кода (содержимое текстового файла) и преобразует ее в токены. Токены представляют собой все минимальные отдельные фрагменты программы, которые поддаются обработке. Допустимые токены в NanoBASIC берутся непосредственно из терминалов описанной в предыдущем разделе грамматики NanoBASIC.

Для поиска токенов используются шаблоны регулярных выражений: каждому типу токена присваивается шаблон регулярного выражения, а затем выполняется поиск по одному типу за один раз. Сложность этой настройки заключается в том, что нам нужно быть осторожными с порядком, в котором происходит поиск. Если два регулярных выражения могут соответствовать одному и тому же токену, то порядок будет важен. Например, в нашем токенизаторе регуляр-

ное выражение для имени переменной может также соответствовать токену PRINT (или любому другому имени оператора), поэтому поиск токена имени переменной должен быть заведомо последним.

Мы начнем создание токенизатора с определения всех различных типов токенов в виде перечисления. Каждый случай будет сопровождаться регулярным выражением для его поиска. Некоторые токены также будут иметь определенные пользователем значения, обозначенные как True или False в конце каждого случая перечисления. Например, токен переменной будет иметь фактическое имя переменной, связанное с ним в качестве ассоциированного значения. Вот как выглядит наше перечисление TokenType:

NanoBASIC/tokenizer.py

```
from enum import Enum
from typing import TextIO
import re
from dataclasses import dataclass

class TokenType(Enum):
    COMMENT = (r'rem.*', False)
    WHITESPACE = (r'[\t\n\r]', False)
    PRINT = (r'print', False)
    IF_T = (r'if', False)
    THEN = (r'then', False)
    LET = (r'let', False)
    GOTO = (r'goto', False)
    GOSUB = (r'gosub', False)
    RETURN_T = (r'return', False)
    COMMA = (r',', False)
    EQUAL = (r'=', False)
    NOT_EQUAL = (r'<>|><', False)
    LESS_EQUAL = (r'<=', False)
    GREATER_EQUAL = (r'>=', False)
    LESS = (r'<', False)
    GREATER = (r'>', False)
    PLUS = (r'\+', False)
    MINUS = (r'-', False)
    MULTIPLY = (r'\*', False)
    DIVIDE = (r'/', False)
    OPEN_PAREN = (r'\(', False)
    CLOSE_PAREN = (r'\)', False)
    VARIABLE = (r'[A-Za-z_]+', True)
    NUMBER = (r'-?[0-9]+', True)
    STRING = (r'".*"', True)

    def __init__(self, pattern: str, has_associated_value: bool):
        self.pattern = pattern
        self.has_associated_value = has_associated_value

    def __repr__(self) -> str:
        return self.name
```

Перечисление `TokenType` просто описывает тип токена. Помимо типа токена, нам также нужно знать, в каком месте исходного кода он появился. Это будет полезно для точного определения места синтаксических ошибок и передачи этой информации программисту. Всю эту информацию мы будем хранить в `Token`, составном типе, который объединяет тип токена, его местоположение и связанное значение, если применимо:

```
@dataclass(frozen=True)
class Token:
    kind: TokenType
    line_num: int
    col_start: int
    col_end: int
    associated_value: str | int | None
```

Тип свойства `associated_value`, `str | int | None`, использует расширенный синтаксис подсказки типа, который был введен в Python 3.10 благодаря PEP 604[1].

Этот вопрос был уже кратко затронут в главе 1, а здесь мы рассмотрим его более подробно. Это способ создания *объединенного типа* (union type). Переменная, объявленная как объединенный тип, может ссылаться на значения любого из типов, составляющих объединение. В версиях Python до 3.10 вам нужно было бы импортировать `Union` из `typing`, и подсказка типа выглядела бы как `Union[str, int, None]`. Новый синтаксис, безусловно, гораздо менее громоздкий. Если говорить кратко, это означает, что `associated_value` может быть строкой, целым числом или `None`.

Мы готовы прочитать файл исходного кода и разбить его на составляющие токены с помощью функции `tokenize()`:

```
def tokenize(text_file: TextIO) -> list[Token]:
    tokens: list[Token] = []
    for line_num, line in enumerate(text_file.readlines(), start=1):
        col_start: int = 1
```

Функция принимает объект `TextIO` – тип объекта, который может выступать в качестве текстового потока. Мы инициализируем список токенов, в котором будем собирать все токены из всего файла. Затем просматриваем каждую строку файла. Поскольку мы хотим сообщить пользователю номера строк и столбцов в том виде, в котором они могут отображаться в текстовом редакторе, мы начинаем с 1. Далее извлекаем все токены:

[1] См. *https://peps.python.org/pep-0604*.

```
        while len(line) > 0:
            found: re.Match | None = None
            for possibility in TokenType:
                # Перебор каждого шаблона с начала, без учета регистра
                # В случае обнаружения сохраняем совпадение в *found*
              ❶ found = re.match(possibility.pattern, line, re.IGNORECASE)
                if found:
                    col_end: int = col_start + found.end() - 1
                    # Сохранение всех токенов, кроме комментариев и пробелов
                  ❷ if (possibility is not TokenType.WHITESPACE
                            and possibility is not TokenType.COMMENT):
                        associated_value: str | int | None = None
                        if possibility.has_associated_value:
                            if possibility is TokenType.NUMBER:
                                associated_value = int(found.group(0))
                            elif possibility is TokenType.VARIABLE:
                                associated_value = found.group()
                            elif possibility is TokenType.STRING:
                                # Удаление символов кавычек
                                associated_value = found.group(0)[1:-1]
                        ❸ tokens.append(Token(possibility, line_num, col_start,
                                              col_end, associated_value))
                    # Продолжение поиска с места в строке после токена
                    line = line[found.end():]
                    col_start = col_end + 1
                    break # повторение в поисках следующего токена
            # Если все токены проверены и ни один из них не совпал,
            # то этот токен является недействительным
          ❹ if not found:
                print(f"Syntax error on line {line_num} column {col_start}")
                break

    ❺ return tokens
```

Мы просматриваем каждую строку файла слева направо в поисках совпадения каждого возможного шаблона токена по порядку ❶. Когда мы находим совпадение, которое не является пробелом или комментарием (они игнорируются) ❷, мы проверяем, является ли это типом токена со связанным значением. Если да, мы сохраняем связанное значение. Мы создаем `Token`, содержащий найденный `TokenType`, место его нахождения и любое связанное значение, и добавляем его в нашу подборку `tokens` ❸. Это действительно очень простой линейный процесс. Если мы находим фрагмент текста, который не соответствует ни одному из известных `TokenType` для NanoBASIC, это считается синтаксической ошибкой, и мы предупреждаем об этом пользователя ❹. Наконец, происходит возврат `tokens` ❺.

Токенизатор – это самая простая часть нашего интерпретатора. Он отвечает за преобразование начального файла исходного кода в подборку действительных токенов языка. Затем эти токены передаются

в парсер. Но прежде чем мы рассмотрим парсер, давайте разберемся с составными элементами, которые будет генерировать парсер для среды выполнения интерпретатора: с узлами.

Узлы

Наш парсер в конечном итоге сгенерирует AST с узлами (nodes), которые будут представлять каждую значимую часть программы. Например, каждое условие `IF` будет узлом, и каждое извлечение значения переменной также будет узлом. Поскольку это дерево, AST связывает все узлы в иерархию отношений. Для иллюстрации этой концепции давайте посмотрим на реальное возможное ветвление (с использованием фактических имен узлов) AST из нашего интерпретатора. Это ветвление будет представлять оператор `IF`: `IF A < 10 THEN GOTO 40`.

Корневым узлом ветви будет `IfStatement`. Этот узел `IfStatement` будет связан с узлом `BooleanExpression` (`A < 10`) и узлом `GoToStatement` (`GOTO 40`). Узел `BooleanExpression` будет иметь внутреннюю переменную для представления `TokenType` своего оператора (`<`), связь с узлом `VarRetrieve` (`A`) и связь с узлом `NumberLiteral` (`10`). Узел `GoToStatement` будет связан с единственным узлом `NumberLiteral` (`40`). Эта структура представлена на рис. 2.1. Обратите внимание, что метки на стрелках представляют собой фактические имена связей между узлами в коде.

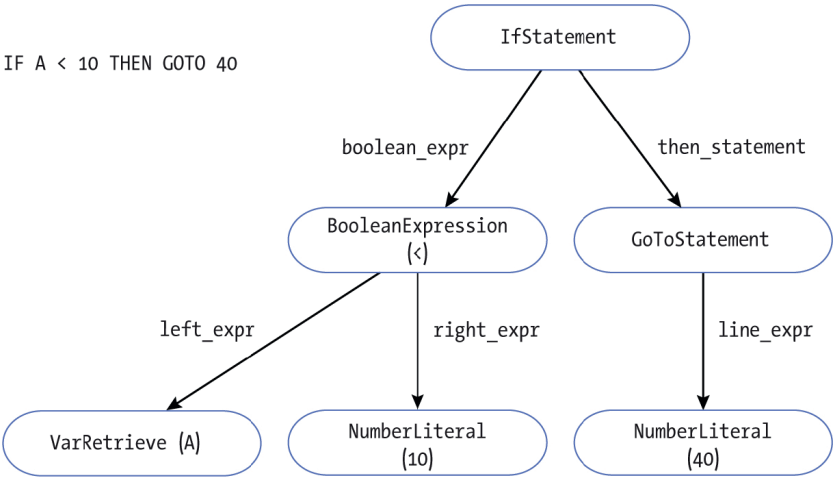


Рис. 2.1. Узлы для `IF A < 10 THEN GOTO 40`

Задача нашего парсера заключается в преобразовании наборов значимых смежных токенов в узлы AST. На заключительном этапе работы нашего интерпретатора узлы AST будут *пройдены*, что включает в себя выполнение всех действий, с которыми связан каждый узел, в нужном порядке. Каждый узел, который может появиться в нашем AST, имеет свой собственный класс в *nodes.py*. Все узлы наследуются

от класса Node. Каждый узел отслеживает свое местоположение в начальном файле исходного кода для целей отладки:

NanoBASIC/nodes.py

```
from dataclasses import dataclass
from NanoBASIC.tokenizer import TokenType

# Для отладки нам нужно знать расположение всех узлов
@dataclass(frozen=True)
class Node:
    line_num: int
    col_start: int
    col_end: int
```

Теперь определим узел `Statement`:

```
# Все операторы в NanoBASIC имеют идентификатор номера строки,
# который программист вставляет перед оператором (*line_id*).
# Это немного сбивает с толку, потому что есть еще и "физический"
# номер строки (*line_num*), который показывает, на какой строке
# в файле находится оператор.
@dataclass(frozen=True)
class Statement(Node):
    line_id: int
```

Каждое выражение в NanoBASIC появляется после определенного пользователем номера строки. Это необходимо для вызовов `GOTO` и `GOSUB`. Не следует путать эти номера строк с `line_num` каждого объекта `Node`, то есть местом, где `Node` появился в файле исходного кода. Для ясности мы называем определенный пользователем номер строки `line_id` в классе `Statement`. Например, если первая строка моего исходного кода – это `23 PRINT "HELLO"`, то `line_id` равен `23`, а `line_num` равен `1`.

`NumericExpression` – это тип `Node`, который при вычислении может дать одно целое число. Это может быть бинарная операция, унарная операция, числовой литерал или поиск переменной, поэтому мы объявим все эти узлы подклассами `NumericExpression`:

```
# Числовое выражение является тем, что можно вычислить в виде числа.
# Это суперкласс литералов, переменных и простых арифметических операций.
@dataclass(frozen=True)
class NumericExpression(Node):
    pass

# Числовое выражение с двумя операндами, такое как 2 + 2 или 8 / 4
@dataclass(frozen=True)
class BinaryOperation(NumericExpression):
    operator: TokenType
```

```
    left_expr: NumericExpression
    right_expr: NumericExpression

    def __repr__(self) -> str:
        return f"{self.left_expr} {self.operator} {self.right_expr}"

# Числовое выражение с одним операндом, такое как -4
@dataclass(frozen=True)
class UnaryOperation(NumericExpression):
    operator: TokenType
    expr: NumericExpression

    def __repr__(self) -> str:
        return f"{self.operator}{self.expr}"

# Целое число, записанное в коде NanoBASIC
@dataclass(frozen=True)
class NumberLiteral(NumericExpression):
    number: int

# Переменная *name*, значение которой должно быть извлечено
@dataclass(frozen=True)
class VarRetrieve(NumericExpression):
    name: str
```

Для выполнения оценки эти различные типы числовых выражений должны содержать определенную информацию. Например, `VarRetrieve` должен иметь имя изменяемой переменной, значение которой ищется. Аналогичным образом `BinaryOperation`, который также можно рассматривать как арифметическую операцию, должен хранить фактическую выполняемую арифметическую операцию (сложение, вычитание, умножение или деление), поэтому мы храним с ним токен оператора.

В то время как `NumericExpression` преобразуется в целое число, `BooleanExpression` предназначен для получения булевых значений. Он принимает два узла `NumericExpression` и сравнивает их с помощью булева оператора (сохраненного в виде токена):

```
# Булево выражение может быть вычислено в виде значения true или false.
# Оно принимает два числовых выражения, *left_expr* и *right_expr*, и сравнивает
# их с помощью булева *operator*.
@dataclass(frozen=True)
class BooleanExpression(Node):
    operator: TokenType
    left_expr: NumericExpression
    right_expr: NumericExpression

    def __repr__(self) -> str:
        return f"{self.left_expr} {self.operator} {self.right_expr}"
```

Остальные узлы предназначены для представления шести типов операторов NanoBASIC:

```
# Представление оператора LET, устанавливающего *name* в *expr*
@dataclass(frozen=True)
class LetStatement(Statement):
    name: str
    expr: NumericExpression

# Представление оператора GOTO, передающего управление *line_expr*
@dataclass(frozen=True)
class GoToStatement(Statement):
    line_expr: NumericExpression

# Представление оператора GOSUB, передающего управление *line_expr*
# Возвращаемый line_id здесь не сохраняется, он будет поддерживаться стеком
@dataclass(frozen=True)
class GoSubStatement(Statement):
    line_expr: NumericExpression

# Представление оператора RETURN, передающего управление строке после
# последнего оператора GOSUB
@dataclass(frozen=True)
class ReturnStatement(Statement):
    pass

# Оператор PRINT со всем, что он должен вывести (с разделением запятыми)
@dataclass(frozen=True)
class PrintStatement(Statement):
    printables: list[str | NumericExpression]

# Условие IF
# *then_statement* - это оператор, который будет выполнен, если
# *boolean_expression* равно true
@dataclass(frozen=True)
class IfStatement(Statement):
    boolean_expr: BooleanExpression
    then_statement: Statement
```

Свойства этих узлов отражают фрагменты данных, которые требуются для каждого типа оператора. Например, поскольку оператор `LET` присваивает значение переменной, узлу `LetStatement` требуется строковое имя переменной и `NumericExpression`, передающее значение.

Ошибки

Немного отвлечемся на обсуждение обработки ошибок в нашем интерпретаторе. Ничто так не раздражает при программировании, как

недостаточные сообщения об ошибках. Когда вы делаете ошибку в коде, вы хотите знать, что произошло и где. Как создатель языка программирования, вы несете ответственность за предоставление пользователю (программисту NanoBASIC) качественных сообщений об ошибках.

NanoBASIC будет сообщать о двух общих типах ошибок: ошибках парсера и ошибках интерпретатора. *Ошибки парсера* можно рассматривать как синтаксические ошибки, например, когда токены расположены в неправильном порядке. Так, после `GOTO` должно быть числовое выражение (представляющее номер строки), но не после оператора `IF`. *Ошибки интерпретатора* – это семантические ошибки. Они возникают в том случае, когда программа пытается сделать что-то нелогичное, например использовать переменную до ее инициализации. Определим классы ошибок для обоих типов ошибок:

NanoBASIC/errors.py

```
from NanoBASIC.tokenizer import Token
from NanoBASIC.nodes import Node

class NanoBASICError(Exception):
    def __init__(self, message: str, line_num: int, column: int):
        super().__init__(message)
        self.message = message
        self.line_num = line_num
        self.column = column

    def __str__(self):
        return (f"{self.message} Occurred at line {self.line_num} "
                f"and column {self.column}")

class ParserError(NanoBASICError):
    def __init__(self, message: str, token: Token):
        super().__init__(message, token.line_num, token.col_start)

class InterpreterError(NanoBASICError):
    def __init__(self, message: str, node: Node):
        super().__init__(message, node.line_num, node.col_start)
```

Как `ParserError`, так и `InterpreterError` являются подклассами `NanoBASICError`. Он, в свою очередь, является подклассом `Exception` – встроенного класса Python, с помощью которого можно создавать собственные исключения для использования в программе. Эти классы выводят сообщение, связанное с ошибкой, и место ее возникновения в исходной программе. Например, предположим, что у нас есть следующая программа:

```
10 PRINT(A)
```

Это приведет к сообщению о следующей ошибке:

```
NanoBASIC.errors.InterpreterError: Var A used before initialized. Occurred at
line 1 and column 10 (переменная A использована до инициализации. Произошло
в строке 1 и столбце 10)
```

Данная ошибка возникает из-за того, что переменная A так и не была инициализирована с помощью оператора LET. В следующих разделах вы увидите много сообщений `ParserError` и `InterpreterError`.

Парсер

Парсер (синтаксический анализатор) получает токены от токенизатора и пытается преобразовать их в структуры, пригодные для интерпретации программы. Синтаксический анализ – это хорошо изученная область компьютерных наук, и существует множество различных алгоритмов синтаксического анализа. Есть даже специальные программы, которые генерируют парсеры. Нет ничего удивительного в том, что они называются *генераторами парсеров* (parser generators). Генератор парсеров может принимать грамматику в форме BNF и выводить парсер.

Конечно, мы могли бы использовать здесь и генератор парсеров, но это не было бы столь поучительно, как самостоятельное написание парсера. И хотя существует множество алгоритмов парсинга, как выяснилось, один из самых простых является также одним из самых эффективных, настраиваемых и широко используемых. Это алгоритм *рекурсивного спуска* (recursive descent), который лежит в основе парсеров C/C++ в двух самых популярных компиляторах в мире, GCC и Clang[1]. Эта техника также использовалась в оригинальной версии Tiny BASIC, созданной Деннисом Эллисоном[2].

При рекурсивном спуске, как правило, каждый определенный в грамматике нетерминальный символ становится функцией. Эта функция отвечает за проверку того, что последовательность анализируемых ею токенов соответствует заданному в грамматике производственному правилу. Парсер проверяет токены путем их последовательного просмотра. Если анализируемый токен, как ожидается, является частью другого производственного правила, парсер рекурсивного спуска просто вызывает функцию, представляющую это другое производственное правило. В случае успеха функции рекурсивного спуска возвращают соответствующие узлы, и это означает, что функция действительно нашла ожидаемые токены.

---

[1] Joseph Sibony, «GCC vs. Clang: Battle of the Behemoths», Incredibuild, 27 мая 2021 г., *https://www.incredibuild.com/blog/gcc-vsclang-battle-of-the-behemoths*.

[2] Tom Pittman, «Tiny Basic Experimenter's Kit», 1977, посещение 4 декабря 2024 г., *http://www.ittybittycomputers.com/IttyBitty/TinyBasic/TBEK.txt*.

Рекурсивный спуск – это метод *нисходящего* парсинга, то есть парсинг начинается с «начала» грамматики (в нашем случае это `<line>`) и «спускается» до достижения необходимой конкретной точки. Это и есть *спуск*, а вот *рекурсивная* часть требует немного большего визуального представления. Представьте, что мы анализируем оператор `IF`. Оператор `IF` – это тип оператора, и каждый нетерминальный символ, включая как «оператор», так и «оператор `IF`», может получить соответствующую функцию в нашем парсере рекурсивного спуска. Оператор `IF` имеет дополнение `THEN`, которое также является оператором. Таким образом, когда мы выполняем парсинг оператора `IF`, мы можем снова вызвать нашу функцию парсинга оператора для дополнения `THEN` – ту же функцию, которая вызвала нашу функцию парсинга оператора `IF`! Это своего рода рекурсия. В итоге мы вызываем функцию, которая вызвала функцию, в которой мы сейчас находимся.

Как *спуск*, так и *рекурсия* станут более понятными в процессе изучения кода класса Parser:

NanoBASIC/parser.py

```
from NanoBASIC.tokenizer import Token
from typing import cast
from NanoBASIC.nodes import *
from NanoBASIC.errors import ParserError

class Parser:
    def __init__(self, tokens: list[Token]):
        self.tokens = tokens
        self.token_index: int = 0

    @property
    def out_of_tokens(self) -> bool:
        return self.token_index >= len(self.tokens)

    @property
    def current(self) -> Token:
        if self.out_of_tokens:
            raise (ParserError(f"No tokens after "
                               f"{self.previous.kind}", self.previous))
        return self.tokens[self.token_index]

    @property
    def previous(self) -> Token:
        return self.tokens[self.token_index - 1]
```

Класс `Parser` получает набор токенов от токенизатора. По мере продвижения разбора внутренний `token_index` отслеживает, на каком токене мы сейчас находимся. Мы также определяем некоторые удобные свойства для извлечения текущего или предыдущего токена.

Вспомогательный метод `consume()` проверяет, является ли текущий токен ожидаемым токеном, увеличивает `token_index` и возвращает

проверенный токен. Если токен не является ожидаемым, мы вызываем `ParserError`:

```
    def consume(self, kind: TokenType) -> Token:
        if self.current.kind is kind:
            self.token_index += 1
            return self.previous
        raise ParserError(f"Expected {kind} after {self.previous}"
                          f"but got {self.current}.", self.current)
```

В парсерах широко используется вспомогательная функция `consume()`, иногда называемая `eat()` или `accept()`, поскольку проверка того, является ли токен ожидаемым, и переход к следующему, если это так, является очень распространенным шаблонным алгоритмом. Если бы у нас не было `consume()`, вы бы увидели много ненужного дублирующего кода.

Цель нашего парсера заключается в создании AST, по которому может пройти среда выполнения для исполнения программы NanoBASIC. Корнем AST будет список операторов. Другой способ представить это – считать, что программа NanoBASIC представляет из себя просто список операторов, написанных в порядке следования сверху вниз в файле исходного кода. В конечном итоге наша среда выполнения будет выполнять эти операторы по одному за раз. Поэтому наш парсер рекурсивного спуска начинает работу с `parse()`, который «спускается» по другим методам парсера и в конце возвращает список операторов:

```
    def parse(self) -> list[Statement]:
        statements: list[Statement] = []
        while not self.out_of_tokens:
            statement = self.parse_line()
            statements.append(statement)
        return statements
```

Каждый оператор должен быть написан в отдельной строке, рядом с идентификатором строки, поэтому первым шагом в спуске является разбор строки:

```
    def parse_line(self) -> Statement:
        number = self.consume(TokenType.NUMBER)
        return self.parse_statement(cast(int, number.associated_value))
```

Мы ожидаем, что идентификатор строки будет находиться в начале строки. Поэтому `parse_line()` начинает с попытки обработать токен `NUMBER`. Если это удается, мы продолжаем разбор самого оператора. Использование `cast()` в данном случае необходимо для проверки типа. Если вы помните из кода токенизатора (вернитесь и посмотрите, если

это поможет), associated_value токена может быть либо целым числом, либо строкой, либо None. Мы знаем, что NUMBER в качестве associated_value всегда представляет целое число, поэтому можно безопасно преобразовать его в int. Такие средства проверки типов, как mypy или Pyright, могут успешно использовать это преобразование.

Обратите внимание, как метод parse_line() соответствует нетерминальному <line> в грамматике. С этого момента многие из наших методов будут прямыми аналогами нетерминальных элементов в грамматике или их соответствующих производственных правил (вернитесь и посмотрите грамматику в качестве руководства). Например, наш следующий метод, parse_statement(), соответствует нетерминальному <statement>:

```
def parse_statement(self, line_id: int) -> Statement:
    match self.current.kind:
        case TokenType.PRINT:
            return self.parse_print(line_id)
        case TokenType.IF_T:
            return self.parse_if(line_id)
        case TokenType.LET:
            return self.parse_let(line_id)
        case TokenType.GOTO:
            return self.parse_goto(line_id)
        case TokenType.GOSUB:
            return self.parse_gosub(line_id)
        case TokenType.RETURN_T:
            return self.parse_return(line_id)
    raise ParserError("Expected to find start of statement.",
                      self.current)
```

Этот метод отвечает за определение того, какой из шести операторов NanoBASIC появляется в следующих нескольких токенах. К счастью, каждый оператор NanoBASIC можно идентифицировать по его единственному начальному токену (PRINT, IF, LET, GOTO, GOSUB или RETURN), поэтому нам нужно просто сопоставить текущий токен с шестью возможными вариантами.

Для удобства я разделил каждый тип оператора на отдельный метод, хотя они не соответствуют непосредственно нетерминальным. Вместо этого можно считать, что каждое из правил производства <statement> получает свой собственный метод. Начнем с оператора PRINT, который является одним из самых сложных для разбора, поскольку в его <expr-list> может быть несколько различных типов, разделенных запятыми.

```
# PRINT "ЗАПЯТАЯ",РАЗДЕЛЕННЫЕ,7154
def parse_print(self, line_id: int) -> PrintStatement:
    print_token = self.consume(TokenType.PRINT)
```

```
        printables: list[str | NumericExpression] = []
        last_col: int = print_token.col_end
        while True: # продолжать поиск объектов для вывода
            if self.current.kind is TokenType.STRING: ❶
                string = self.consume(TokenType.STRING)
                printables.append(cast(str, string.associated_value))
                last_col = string.col_end
            elif (expression := self.parse_numeric_expression()) is not None: ❷
                printables.append(expression)
                last_col = expression.col_end
            else: ❸
                raise ParserError("В списке печати допускаются только строки "
                                  "и числовые выражения.", self.current)
            # Запятая означает, что есть что-то еще для вывода
            если self.out_of_tokens и self.current.kind не равны TokenType.COMMA: ❹
                self.consume(TokenType.COMMA)
                continue
            break
        return PrintStatement(line_id=line_id, line_num=print_token.line_num,
                              col_start=print_token.col_start, col_end=last_col,
                              printables=printables)
```

Мы храним элементы для вывода в списке `printables` Python. Чтобы собрать их, мы продолжаем двигаться вперед (используя цикл), токен за токеном, проверяя наличие строки ❶ или числового выражения ❷. Пока мы находим одно из них, за которым следует запятая ❹, мы продолжаем цикл. Если мы находим что-то, что не является строкой или числовым выражением ❸, мы выводим ParserError. По мере выполнения цикла мы также отслеживаем конец столбца последнего элемента для нужд отладки. Конечный возвращаемый узел `PrintStatement` должен знать, где он начинается и где заканчивается; он начинается там, где начинается токен `PRINT`, и заканчивается в конце последнего столбца последнего элемента в списке выражений.

Далее рассмотрим `parse_if()`, который включает хороший пример использования рекурсивного спуска:

```
# IF BOOLEAN_EXPRESSION THEN STATEMENT
def parse_if(self, line_id: int) -> IfStatement:
    if_token = self.consume(TokenType.IF_T)
    boolean_expression = self.parse_boolean_expression()
    self.consume(TokenType.THEN)
    statement = self.parse_statement(line_id)
    return IfStatement(line_id=line_id, line_num=if_token.line_num,
                       col_start=if_token.col_start, col_end=statement.col_end,
                       boolean_expr=boolean_expression, then_statement=statement)
```

Как уже упоминалось, `THEN` в операторе `IF` является еще одним оператором. Для парсинга `THEN` мы вызываем `parse_statement()` – тот же метод, который выше в цепочке вызовов привел нас к `parse_if()`. Од-

нако сначала мы разберем булево выражение в начале оператора IF. Вскоре мы рассмотрим способ выполнения этой операции.

Во всех рассмотренных до сих пор методах разбора обратите внимание на то, как мы вызываем другие методы разбора и предполагаем, что они работают. Другие методы должны самостоятельно обрабатывать ошибки и постоянно перемещать token_index, обычно вызывая consume(). Этот принцип продолжается в методах для других четырех типов операторов:

```
# LET VARIABLE = VALUE
def parse_let(self, line_id: int) -> LetStatement:
    let_token = self.consume(TokenType.LET)
    variable = self.consume(TokenType.VARIABLE)
    self.consume(TokenType.EQUAL)
    expression = self.parse_numeric_expression()
    return LetStatement(line_id=line_id, line_num=let_token.line_num,
                        col_start=let_token.col_start, col_end=expression.col_end,
                        name=cast(str, variable.associated_value), expr=expression)

# GOTO NUMERIC_EXPRESSION
def parse_goto(self, line_id: int) -> GoToStatement:
    goto_token = self.consume(TokenType.GOTO)
    expression = self.parse_numeric_expression()
    return GoToStatement(line_id=line_id, line_num=goto_token.line_num,
                         col_start=goto_token.col_start, col_end=expression.col_end,
                         line_expr=expression)

# GOSUB NUMERIC_EXPRESSION
def parse_gosub(self, line_id: int) -> GoSubStatement:
    gosub_token = self.consume(TokenType.GOSUB)
    expression = self.parse_numeric_expression()
    return GoSubStatement(line_id=line_id, line_num=gosub_token.line_num,
                          col_start=gosub_token.col_start,
                          col_end=expression.col_end,
                          line_expr=expression)

# RETURN
def parse_return(self, line_id: int) -> ReturnStatement:
    return_token = self.consume(TokenType.RETURN_T)
    return ReturnStatement(line_id=line_id, line_num=return_token.line_num,
                           col_start=return_token.col_start,
                           col_end=return_token.col_end)
```

Эти четыре метода парсинга довольно похожи друг на друга. В каждом случае мы ожидаем определенный начальный токен (например, LET или GOTO), а затем производим парсинг некоторой информации, необходимой для создания узла для этого типа оператора. Например, для оператора LET нам нужны переменная и числовое выражение, а для оператора GOTO нужно только числовое выражение (строка, к ко-

торой нужно перейти). Самым простым для разбора является оператор RETURN, поскольку после RETURN ничего не следует.

Как и обещано, ниже приведен метод parse_boolean_expression():

```
# NUMERIC_EXPRESSION BOOLEAN_OPERATOR NUMERIC_EXPRESSION
def parse_boolean_expression(self) -> BooleanExpression:
    left = self.parse_numeric_expression()
    if self.current.kind in {TokenType.GREATER, TokenType.GREATER_EQUAL, TokenType.EQUAL,
                             TokenType.LESS, TokenType.LESS_EQUAL, TokenType.NOT_EQUAL}:
        operator = self.consume(self.current.kind)
        right = self.parse_numeric_expression()
        return BooleanExpression(line_num=left.line_num,
                                 col_start=left.col_start, col_end=right.col_end,
                                 operator=operator.kind, left_expr=left, right_expr=right)
    raise ParserError(f" Ожидался булев оператор, но найден "
                      f"{self.current.kind}.", self.current)
```

Булево выражение должно содержать два числовых выражения и один из допустимых операторов между ними. Числовое выражение перед оператором называется левым (left), а числовое выражение после оператора называется правым (right). Оператор хранится в узле BooleanExpression, чтобы мы могли выполнить соответствующее сравнение во время выполнения.

Разбор числовых выражений строго следует иерархии нетерминальных символов в грамматике, от <expression> к <term> и далее к <factor>, с методом для каждого из них. В <factor> может быть включен <var> или <number>, но все они обрабатываются непосредственно в parse_factor(), поскольку необходимая информация о них уже содержится в соответствующих токенах:

```
def parse_numeric_expression(self) -> NumericExpression:
    left = self.parse_term()
    # Продолжение парсинга +s и -s до тех пор, пока их не останется
    while True:
        if self.out_of_tokens: # что, если выражение является концом файла?
            return left
        if self.current.kind is TokenType.PLUS:
            self.consume(TokenType.PLUS)
            right = self.parse_term()
            left = BinaryOperation(line_num=left.line_num, col_start=left.col_start,
                                   col_end=right.col_end, operator=TokenType.PLUS,
                                   left_expr=left, right_expr=right)
        elif self.current.kind is TokenType.MINUS:
            self.consume(TokenType.MINUS)
            right = self.parse_term()
            left = BinaryOperation(line_num=left.line_num, col_start=left.col_start,
                                   col_end=right.col_end, operator=TokenType.MINUS,
                                   left_expr=left, right_expr=right)
```

```
            else:
                break # больше нет, видимо, конец выражения
        return left

    def parse_term(self) -> NumericExpression:
        left = self.parse_factor()
        # Продолжение парсинга *s и /s до тех пор, пока их не останется
        while True:
            if self.out_of_tokens: # что, если выражение является концом файла?
                return left
            if self.current.kind is TokenType.MULTIPLY:
                self.consume(TokenType.MULTIPLY)
                right = self.parse_factor()
                left = BinaryOperation(line_num=left.line_num, col_start=left.col_start,
                                       col_end=right.col_end, operator=TokenType.MULTIPLY,
                                       left_expr=left, right_expr=right)
            elif self.current.kind is TokenType.DIVIDE:
                self.consume(TokenType.DIVIDE)
                right = self.parse_factor()
                left = BinaryOperation(line_num=left.line_num, col_start=left.col_start,
                                       col_end=right.col_end, operator=TokenType.DIVIDE,
                                       left_expr=left, right_expr=right)
            else:
                break # больше нет, видимо, конец выражения
        return left

    def parse_factor(self) -> NumericExpression:
        if self.current.kind is TokenType.VARIABLE:
            variable = self.consume(TokenType.VARIABLE)
            return VarRetrieve(line_num=variable.line_num,
                               col_start=variable.col_start, col_end=variable.col_end,
                               name=cast(str, variable.associated_value))
        elif self.current.kind is TokenType.NUMBER:
            number = self.consume(TokenType.NUMBER)
            return NumberLiteral(line_num=number.line_num,
                                 col_start=number.col_start, col_end=number.col_end,
                                 number=int(cast(str, number.associated_value)))
        elif self.current.kind is TokenType.OPEN_PAREN:
            self.consume(TokenType.OPEN_PAREN)
            expression = self.parse_numeric_expression()
            if self.current.kind is not TokenType.CLOSE_PAREN:
                raise ParserError("Ожидаемое соответствие закрывающей скобки.", self.current)
            self.consume(TokenType.CLOSE_PAREN)
            return expression
        elif self.current.kind is TokenType.MINUS:
            minus = self.consume(TokenType.MINUS)
            expression = self.parse_factor()
            return UnaryOperation(line_num=minus.line_num,
                                  col_start=minus.col_start, col_end=expression.col_end,
                                  operator=TokenType.MINUS, expr=expression)
        raise ParserError("Неожиданный токен в числовом выражении.", self.current)
```

Обратите внимание на порядок приоритетов. В арифметике мы ожидаем, что деление будет иметь более высокий приоритет, чем вычитание, а скобки – более высокий приоритет, чем что-либо другое. Как я уже упоминал, когда мы впервые обсуждали грамматику NanoBASIC, это можно смоделировать в рекурсивном спуске по порядку, в котором анализируются нетерминальные символы. Чем дальше вы спускаетесь, тем выше приоритет. В этом случае все, что обрабатывается в `parse_term()`, будет иметь более высокий приоритет, чем все в `parse_numeric_expression()`, а все в `parse_factor()` будет иметь более высокий приоритет, чем все в `parse_numeric_expression()` или `parse_term()`. Это также причина того, что `-` и `+` появляются в одном и том же правиле производства, а `/` и `*` появляются «глубже».

Каждый раз, когда нам нужна левая или правая сторона выражения в `parse_numeric_expression()` или `parse_term()`, мы спускаемся. Например, `parse_numeric_expression()` никогда не вызывает `parse_numeric_expression()`. Вместо этого она вызывает `parse_term()`. Это может показаться нелогичным, поскольку вы можете задаться вопросом, как обрабатываются несколько последовательных суммирований. Все дело в том, что `parse_numeric_expression()` и `parse_term()` используют циклы, подобно тому, как мы делали в `parse_print()` для обработки произвольного количества арифметических операций (таких как многократные операции суммирования). Можно думать об этом так: мы спускаемся и обрабатываем все, что имеет более высокий приоритет, а затем возвращаемся назад, чтобы продолжить цикл в `parse_numeric_expression()`, если еще остались токены сложения или вычитания.

Рассмотрим пример. Допустим, мы анализируем выражение `2 + 3 * 4 + 5`. Инициализация `left` в `parse_numeric_expression()` вызывает `parse_term()`, который вызывает `parse_factor()`, который возвращает `NumberLiteral` для `2`. Затем мы возвращаем `2` обратно в `parse_numeric_expression()`, который сохраняет его в `left`. Далее встречается токен `+` и вызывается `parse_term()`. Он анализирует `3 * 4` с использованием нескольких вызовов `parse_factor()`. Мы возвращаемся в `parse_numeric_expression()` с узлом `BinaryOperation` для `3 * 4`, упомянутым как `right`. Затем `left` и `right` объединяются в новую `BinaryOperation` и связываются с `left` (новая привязка для него). Наконец, встречается последний токен `+`, и `5` анализируется практически так же, как и `2` (спускаясь до `parse_factor()`), и связывается с `right`. Снова `left` и `right` объединяются в `left` для окончательного возвращаемого значения `parse_numeric_expression()`.

Попробуйте проработать несколько собственных арифметических примеров для лучшего понимания того, как операторы на более глубоком уровне спуска имеют более высокий приоритет. Возможно, вам захочется открыть код парсера во время работы. Сочетание циклов и повторное использование переменных для разных узлов мо-

жет быть сложным для понимания, но после проработки нескольких примеров все станет на свои места. Вы также можете попробовать добавить к различным методам несколько вызовов `print()` для наглядного отображения процесса парсинга. Вы можете исключить вызовы для запуска AST через среду выполнения в *executioner.py*, если еще не закончили ввод всей программы.

Примечание *Существуют более эффективные способы парсинга арифметических выражений. Один из популярных методов, открытый Дейкстрой, называется алгоритмом сортировочной станции (shunting yard algorithm). Более эффективный алгоритм, такой как сортировочная станция, иногда сочетается в виде гибридной модели с алгоритмом парсинга рекурсивного спуска для частей арифметического выражения.*

Среда выполнения

Конечным результатом работы нашей программы парсинга будет набор узлов AST `Statement` в виде списка, по которому среда выполнения может проходить и пошагово выполнять команды. Класс, который проходит по AST, я назвал `Interpreter`, хотя понимаю, что это может немного сбивать с толку, поскольку вся эта глава посвящена созданию интерпретатора. Да, токенизатор и парсер являются частями общего интерпретатора, но класс `Interpreter` – это место, где язык фактически интерпретируется в том смысле, что токены, которые стали узлами, превращаются в нечто значимое – программу, которая выполняется с определенным выводом. Независимо от того, хорошее ли это название, класс `Interpreter` предоставляет среду выполнения и понимание того, как изменять это окружение или предоставлять вывод на основе встреченных им узлов операторов и выражений.

Класс `Interpreter` начинается подобно `Parser`:

NanoBASIC/interpreter.py

```
from NanoBASIC.nodes import *
from NanoBASIC.errors import InterpreterError
from collections import deque

class Interpreter:
    def __init__(self, statements: list[Statement]):
        self.statements = statements
        self.variable_table: dict[str, int] = {}
        self.statement_index: int = 0
        self.subroutine_stack: deque[int] = deque()

    @property
    def current(self) -> Statement:
        return self.statements[self.statement_index]
```

Вместо списка токенов, как в классе `Parser`, `Interpreter` получает список операторов. Для удобства также имеется свойство current, позволяющее получить доступ к текущему оператору. Среда выполнения состоит из операторов, `statement_index`, `variable_table` для отслеживания значений каждой переменной и `subroutine_stack`, который поможет нам перейти в нужное место после пары `GOSUB` и `RETURN`.

Далее нам нужно найти способ связать идентификатор строки с индексом оператора. Рассмотрим следующую программу NanoBASIC:

```
27 PRINT "HELLO"
38 GOTO 50
45 PRINT "NEVER"
50 PRINT "BYE"
```

Когда выполняется `GOTO 50`, интерпретатор должен найти оператор, связанный с идентификатором строки `50`, и продолжить выполнение с этого места. В NanoBASIC программист может произвольно выбирать любой идентификатор строки для любой строки, при условии что все идентификаторы строк являются целыми числами в порядке возрастания, так как же найти `50`? Нам нужно найти его в списке операторов. Поскольку строки должны быть упорядочены, наш метод `find_line_index()` может выполнить бинарный поиск:

```
    # Возвращает индекс *line_id* с помощью бинарного поиска,
    # или None, если он не найден; считается, что список операторов отсортирован
    def find_line_index(self, line_id: int) -> int | None:
        low: int = 0
        high: int = len(self.statements) - 1
        while low <= high:
            mid: int = (low + high) // 2
            if self.statements[mid].line_id < line_id:
                low = mid + 1
            elif self.statements[mid].line_id > line_id:
                high = mid - 1
            else:
                return mid
        return None
```

Далее метод `run()` последовательно выполняет операторы в `statements`:

```
    def run(self):
        while self.statement_index < len(self.statements):
            self.interpret(self.current)
```

Обратите внимание, что мы используем цикл `while`, управляемый `statement_index`, а не цикл `for...in`. Это связано с тем, что мы можем

перескакивать, пропускать или повторять некоторые операторы из-за GOTO и GOSUB. Другими словами, statement_index может изменяться по мере интерпретации различных операторов внутри цикла.

Метод interpret() является сердцем интерпретатора. Он интерпретирует узлы Statement и изменяет среду выполнения или создает некоторый вывод в зависимости от значения каждого конкретного оператора:

```
def interpret(self, statement: Statement):
    match statement:
        case LetStatement(name=name, expr=expr):
            value = self.evaluate_numeric(expr)
            self.variable_table[name] = value
            self.statement_index += 1
        case GoToStatement(line_expr=line_expr):
            go_to_line_id = self.evaluate_numeric(line_expr)
            if (line_index := self.find_line_index(go_to_line_id)) is not None:
                self.statement_index = line_index
            else:
                raise InterpreterError("No GOTO line id.", self.current)
        case GoSubStatement(line_expr=line_expr):
            go_sub_line_id = self.evaluate_numeric(line_expr)
            if (line_index := self.find_line_index(go_sub_line_id)) is not None:
                self.subroutine_stack.append(self.statement_index + 1) # настройка для RETURN
                self.statement_index = line_index
            else:
                raise InterpreterError("No GOSUB line id.", self.current)
        case ReturnStatement():
            if not self.subroutine_stack: # check if the stack is empty
                raise InterpreterError("RETURN without GOSUB.", self.current)
            self.statement_index = self.subroutine_stack.pop()
        case PrintStatement(printables=printables):
            accumulated_string: str = ""
            for index, printable in enumerate(printables):
                if index > 0: # put tabs between items in the list
                    accumulated_string += "\t"
                if isinstance(printable, NumericExpression):
                    accumulated_string += str(self.evaluate_numeric(printable))
                else: # otherwise, it's a string
                    accumulated_string += str(printable)
            print(accumulated_string)
            self.statement_index += 1
        case IfStatement(boolean_expr=boolean_expr, then_statement=then_statement):
            if self.evaluate_boolean(boolean_expr):
                self.interpret(then_statement)
            else:
                self.statement_index += 1
        case _:
            raise InterpreterError(f"Unexpected item {self.current} "
                                   f"in statement list.", self.current)
```

Процесс перемещения по AST гораздо проще его построения, поскольку мы можем использовать структурное сопоставление шаблонов, обеспечиваемое в Python оператором `match`. В каждом случае (за исключением `ReturnStatement`, который не имеет свойств) мы захватываем некоторые свойства подкласса `Statement` для сопоставления. Например, строка `case LetStatement(name=name, expr=expr):` говорит о том, что если считать, что `statement` является `LetStatement`, то `statement.name` будет храниться в локальной переменной `name`, а `statement.expr` – в локальной переменной `expr`.

В трех случаях интерпретатор продолжает работу, увеличивая значение `statement_index` после выполнения своих задач. В случаях `GoToStatement`, `GoSubStatement` и `ReturnStatement` этого не происходит, поскольку они перемещаются по коду, напрямую изменяя значение `statement_index`. Каждый раз, когда встречается `GoSubStatement`, нам нужно знать, куда вернуться при следующем выполнении `ReturnStatement` – это, если угодно, своего рода закладка. В этом и заключается назначение `subroutine_stack`. В случае `GoSubStatement` мы сохраняем в стеке значение `statement_index + 1` во избежание бесконечного цикла (возврата к источнику `GOSUB`); затем, в случае `ReturnStatement`, мы извлекаем значение из стека.

Обратите внимание, что если `boolean_expr` узла `IfStatement` оценивается как `True`, то `interpret()` вызывается рекурсивно, и `statement_index` не увеличивается. Это происходит из-за того, что оператор, связанный с пунктом `THEN`, сам изменяет `statement_index`.

Вычисление числовых выражений в основном сводится к выполнению правильного оператора Python, совпадающего с токеном арифметического оператора NanoBASIC, или извлечению переменной из `variable_table`:

```
def evaluate_numeric(self, numeric_expression: NumericExpression) -> int:
      match numeric_expression:
          case NumberLiteral(number=number):
              return number
          case VarRetrieve(name=name):
              if name in self.variable_table:
                  return self.variable_table[name]
              else:
                  raise InterpreterError(f"Var {name} used "
                                         f"before initialized.", numeric_expression)
          case UnaryOperation(operator=operator, expr=expr):
              if operator is TokenType.MINUS:
                  return -self.evaluate_numeric(expr)
              else:
                  raise InterpreterError(f"Expected - "
                                         f"but got {operator}.", numeric_expression)
          case BinaryOperation(operator=operator, left_expr=left, right_expr=right):
              if operator is TokenType.PLUS:
                  return self.evaluate_numeric(left) + self.evaluate_numeric(right)
```

```
            elif operator is TokenType.MINUS:
                return self.evaluate_numeric(left) - self.evaluate_numeric(right)
            elif operator is TokenType.MULTIPLY:
                return self.evaluate_numeric(left) * self.evaluate_numeric(right)
            elif operator is TokenType.DIVIDE:
                return self.evaluate_numeric(left) // self.evaluate_numeric(right)
            else:
                raise InterpreterError(f"Unexpected binary operator "
                                       f"{operator}.", numeric_expression)
        case _:
            raise InterpreterError("Expected numeric expression.",
                                   numeric_expression)
```

Обратите внимание на все рекурсивные вызовы в `evaluate_numeric()`. Когда вы только начинаете учиться программированию на императивном языке, таком как Python, рекурсия может показаться вам чем-то необычным. Но по мере того, как вы достигаете среднего или продвинутого уровня программирования, вы уже можете оценить ее преимущества. Этот проект является хорошей тому иллюстрацией. Мы уже видели в классах `Parser` и `Interpreter`, насколько полезна рекурсия для алгоритмического выражения наших идей.

Хотите верьте, хотите нет, но существуют целые языки программирования (в основном в функциональной парадигме), в которых нет циклов, а есть только рекурсия. Это может показаться экстримом, и использование только рекурсии, безусловно, было бы ужасным способом программирования на Python, поскольку это сделало бы ваш код гораздо менее читабельным для других программистов Python и привело бы к некоторым потерям производительности. Но этот факт лишний раз подтверждает, насколько мощным инструментом может быть рекурсия. Все, что можно сделать с помощью циклов, можно сделать и с помощью рекурсии, но магия заключается в том, что рекурсия действительно помогает лучшему выражению идей, как в случае с этим проектом. В частности, рекурсия может быть очень полезна при работе с иерархическими структурами данных, такими как AST.

Оценка булевых выражений во многом похожа на оценку числовых выражений. Она представляет собой преобразование операторов NanoBASIC в операторы Python:

```
def evaluate_boolean(self, boolean_expression: BooleanExpression) -> bool:
    left = self.evaluate_numeric(boolean_expression.left_expr)
    right = self.evaluate_numeric(boolean_expression.right_expr)
    match boolean_expression.operator:
        case TokenType.LESS:
            return left < right
        case TokenType.LESS_EQUAL:
            return left <= right
        case TokenType.GREATER:
```

```
            return left > right
        case TokenType.GREATER_EQUAL:
            return left >= right
        case TokenType.EQUAL:
            return left == right
        case TokenType.NOT_EQUAL:
            return left != right
        case _:
            raise InterpreterError(f"Unexpected boolean operator "
                                   f"{boolean_expression.operator}.", boolean_expression)
```

И на этом все! Запустить программу NanoBASIC гораздо проще, чем выполнить ее разбор.

Примечание *Если бы это был простой компилятор, а не интерпретатор, то вместо прохождения по AST и выполнения каких-либо действий пришлось бы генерировать машинный код при встрече с каждым узлом.*

Запуск программы

Теперь, после завершения работы над интерпретатором NanoBASIC, мы можем запускать некоторые программы NanoBASIC. В интернете можно найти программы на Tiny BASIC, которые можно запускать и на NanoBASIC (или модифицировать, если они используют INPUT или другую функцию, которой нет в NanoBASIC). Вы также можете написать свои собственные программы на NanoBASIC, и это отличный способ протестировать свой интерпретатор. Как уже упоминалось ранее, я также предоставил несколько простых программ NanoBASIC в каталоге *Examples* этого проекта. Например, в начале главы была представлена программа *fib.bas*. Другой пример, *gcd.bas*, позволяет найти наибольший общий делитель двух чисел, указанных в исходном коде. В данном случае он находит наибольший общий делитель чисел 350 и 539:

```
% python3 -m NanoBASIC NanoBASIC/Examples/gcd.bas
7
```

Как и в главе 1, команда запуска программы предполагает, что вы находитесь в главном каталоге репозитория. Не забудьте запустить программу в качестве модуля при помощи опции `-m`.

Тестирование NanoBASIC

Как и в случае с Brainfuck, полезно провести несколько интеграционных тестов, чтобы убедиться в корректной работе нашего интерпре-

татора. Тесты NanoBASIC очень похожи на тесты Brainfuck. Мы перехватываем стандартный вывод и убеждаемся, что ожидаемый вывод совпадает с фактическим выводом для многих программ в каталоге *Examples*:

tests/test_nano basic.py

```
import unittest
import sys
from pathlib import Path
from io import StringIO
from NanoBASIC.executioner import execute

# Токенизация, парсинг и интерпретация программы на NanoBASIC;
# сохранение результата в строке и его возврат
def run(file_name: str | Path) -> str:
    output_holder = StringIO()
    sys.stdout = output_holder
    execute(file_name)
    return output_holder.getvalue()

class NanoBASICTestCase(unittest.TestCase):
    def setUp(self) -> None:
        self.example_folder = (Path(__file__).resolve().parent.parent
                               / 'NanoBASIC' / 'Examples')

    def test_print1(self):
        program_output = run(self.example_folder / "print1.bas")
        expected = "Hello World\n"
        self.assertEqual(program_output, expected)

    def test_print2(self):
        program_output = run(self.example_folder / "print2.bas")
        expected = "4\n12\n30\n7\n100\t9\n"
        self.assertEqual(program_output, expected)

    def test_print3(self):
        program_output = run(self.example_folder / "print3.bas")
        expected = "E is\t-31\n"
        self.assertEqual(program_output, expected)

    def test_variables(self):
        program_output = run(self.example_folder / "variables.bas")
        expected = "15\n"
        self.assertEqual(program_output, expected)

    def test_goto(self):
        program_output = run(self.example_folder / "goto.bas")
        expected = "Josh\nDave\nNanoBASIC ROCKS\n"
        self.assertEqual(program_output, expected)
```

```
    def test_gosub(self):
        program_output = run(self.example_folder / "gosub.bas")
        expected = "10\n"
        self.assertEqual(program_output, expected)

    def test_if1(self):
        program_output = run(self.example_folder / "if1.bas")
        expected = "10\n40\n50\n60\n70\n100\n"
        self.assertEqual(program_output, expected)

    def test_if2(self):
        program_output = run(self.example_folder / "if2.bas")
        expected = "GOOD\n"
        self.assertEqual(program_output, expected)

    def test_fib(self):
        program_output = run(self.example_folder / "fib.bas")
        expected = "0\n1\n1\n2\n3\n5\n8\n13\n21\n34\n55\n89\n"
        self.assertEqual(program_output, expected)

    def test_factorial(self):
        program_output = run(self.example_folder / "factorial.bas")
        expected = "120\n"
        self.assertEqual(program_output, expected)

    def test_gcd(self):
        program_output = run(self.example_folder / "gcd.bas")
        expected = "7\n"
        self.assertEqual(program_output, expected)

if __name__ == "__main__":
    unittest.main()
```

Еще одной особенностью тестов NanoBASIC является то, что многие из них изолируют отдельный тип операторов. Например, `print2.bas` использует только операторы `PRINT` и числовые выражения, поэтому даже если **`GOTO`** работает некорректно, `test_print2()` все равно может проходить (но, надеюсь, ошибки `GOTO` будут обнаружены другим тестом). Эта методология изоляции обеспечивает более детализированные результаты тестирования, но нам все равно нужны более всесторонние интеграционные тесты, такие как *fib.bas*, чтобы убедиться, что операторы работают правильно и вместе. Более надежный набор модульных тестов также включал бы тесты, которые проверяют токенизатор, парсер и интерпретатор независимо друг от друга.

Вам следует убедиться, что все тесты проходят успешно. Вы также можете попробовать добавить собственную программу на BASIC в качестве дополнительного теста.

Код в реальной жизни

Мой отец, Дэнни Копек (Danny Kopec)[1], который был профессором информатики, изучал программирование в колледже Дартмута, где он прошел курс программирования на языке BASIC у президента этого колледжа Джона Кемени (John Kemeny)[2], одного из создателей данного языка. Когда я в середине 1990-х годов в возрасте около восьми лет впервые начал учиться программированию, он купил мне копию True BASIC[3], «официального» BASIC от компании, основанной Кемени и соавтором BASIC Томасом Курцем[4]. К 1995 году BASIC был уже несколько устаревшим, но я этого не знал. Я провел много времени, создавая игры с помощью True BASIC, хотя, по-моему, так и не научился работать с подпрограммами. Полагаю, я писал спагетти-код.

Как и мой отец, я поступил в университет Дартмута на факультет экономики, а после недолгой работы на Уолл-стрит, которую я ненавидел, подал документы в аспирантуру по информатике. Интерес к программированию, который начался с BASIC, все еще жил во мне. Как человек без диплома по информатике может поступить в аспирантуру по информатике? Я прослушал пять курсов по информатике как студент бакалавриата, большинство из них я прошел хорошо, и в свободное время опубликовал несколько проектов. Этого было достаточно, чтобы поступить на несколько программ. В итоге я вернулся в Дартмут, чтобы получить степень магистра. Для тех, кто раздумывает над таким путем, скажу, что отсутствие полного образования уровня бакалавра в области информатики значительно затруднило переход на магистерскую программу. Я бы не рекомендовал поступать в аспирантуру по совершенно другой специальности без серьезной самоподготовки.

Чтобы еще больше усугубить свои ошибки, в первом семестре магистратуры я решил взять курс по сложной теме компиляторов. В итоге я получил самую низкую оценку за всю свою учебу в магистратуре. Курс включал в себя большой проект по созданию компилятора C на языке C (с некоторыми отсутствующими функциями). У меня был напарник по проекту, и мы разделили работу на отдельные части компилятора, аналогичные рассмотренным в этой главе (токенизатор, парсер, генератор кода и т. д.). Компилятор мог работать только в том случае, если бы работали все его части. К последней неделе семестра я написал несколько тысяч строк кода для своих этапов, а мой партнер,

---

1 См. *https://en.wikipedia.org/wiki/Danny_Kopec*.

2 См. *https://en.wikipedia.org/wiki/John_G._Kemeny*.

3 См. *https://en.wikipedia.org/wiki/True_BASIC*.

4 См. *https://en.wikipedia.org/wiki/Thomas_E._Kurtz*.

несмотря на мои постоянные подталкивания, написал менее 100 строк для своей части. Сегодня я часто вспоминаю этот опыт, когда задаю студентам групповые проекты: порой, когда они обвиняют своего партнера, они говорят правду. Без сомнения, мой код тоже был не идеальным, но по крайней мере я его написал.

Мы работали в безумном темпе, не спали несколько ночей подряд, чтобы хоть что-то заработало, соединяя скудный код моего партнера с моим и добавляя много нового для заполнения всех пробелов. В конце концов, наш компилятор выполнял некоторые базовые вещи правильно, но провалил большинство автоматических тестов профессора.

Как человек, который едва не провалил курс по компиляторам, в итоге написал эту главу об интерпретаторах? Спустя восемь лет и пару лет работы в сфере компьютерного образования я вспомнил свой опыт с BASIC. В детстве я также пользовался детским языком программирования Logo[1], и эти два языка вместе были отличным способом обучения программированию для ребенка. В качестве вызова самому себе я решил создать свой собственный детский язык программирования, который был бы чем-то средним между BASIC и Logo. Это не новая идея – многие люди реализовывали похожие проекты, но мне хотелось создать что-то отточенное и доказать себе, что курс по компиляторам не обязательно должен быть для меня концом пути в разработке языков программирования.

Результатом стал продукт под названием SeaTurtle[2], который был продан тиражом в сотни экземпляров. Сотни, а не тысячи. Это не сделало меня богатым, но доказало, что я могу написать язык программирования, который люди действительно захотят купить. Настоящий язык программирования. Ладно, настоящий «детский» язык программирования.

Этот опыт с SeaTurtle привел меня к созданию NanoBASIC в качестве проекта Swift для моего курса «Новые языки» (упомянутого в блоке «Код в реальной жизни» в главе 1). И этот опыт привел меня к созданию данной главы. NanoBASIC очень прост, но он настоящий, и как только вы создадите что-то настоящее, вы уже недалеко от создания чего-то интересного. В конце концов, опыт, который я получил благодаря всем этим занятиям, включая мучительный курс по компиляторам, сделал меня подходящим человеком для написания этой главы. Теперь, когда вы ее прочитали, надеюсь, вам не придется мучиться так же, как мне на том курсе.

---

1 См. *https://en.wikipedia.org/wiki/Logo_(programming_language)*.

2 См. *https://oaksnow.com/seaturtle*.

Практические приложения

Как уже упоминалось в разделе «История BASIC», в эпоху революции персональных компьютеров язык BASIC был стандартным языком программирования. Миллионы программистов начинали свою карьеру с написания программ на BASIC, а Tiny BASIC был самым распространенным диалектом этого языка. Некоторые из создателей Tiny BASIC стали пионерами движения за свободное программное обеспечение. Благодаря открытой лицензии Tiny BASIC был портирован на множество платформ, где его простота оказалась настоящим преимуществом. Он работал на машинах с таким небольшим объемом памяти, что они не могли запускать языки, более сложные, чем Tiny BASIC.

Это может показаться не таким уж большим достижением, но для пользователей ранних персональных компьютеров, у которых не было других альтернатив (из-за стоимости или доступности и ограниченного объема памяти), Tiny BASIC был огромным продвижением по сравнению с необходимостью писать машинный код. Поскольку он распространялся по свободной лицензии и был портирован на множество платформ, он также обеспечивал некоторую переносимость для написанных на нем программ. Если машина могла запускать Tiny BASIC, то она могла запускать и вашу программу.

Язык Tiny BASIC используется и по сей день. Как и Brainfuck, он служит образовательным инструментом, но также используется и для иных целей. На момент написания этой статьи немецкая компания выпускала микроконтроллеры, работающие под управлением версии Tiny BASIC[1].

Интерпретаторы, которые используются в современных популярных языках программирования, гораздо более сложны, чем созданный нами в этой главе, но они имеют те же основные компоненты: токенизатор, парсер, промежуточное представление, такое как AST, и среду выполнения. Дополнительная сложность, как правило, необходима для поддержки дополнительных функций или повышения производительности, но относительно простых методов, описанных в этой главе, вполне достаточно для создания рабочего прототипа нового языка или даже готового к производству доменно-специфичного языка (domain-specific language, DSL) для вашей работы. Как правило, DSL не требуют высокой производительности.

Многие успешные языки программирования в реальных условиях начинались с довольно простых реализаций и со временем эволюционировали. Например, Ruby изначально был интерпретатором с прохождением AST, подобным NanoBASIC. Позже Ruby компилировался в байт-код, который выполнялся на виртуальной машине (подробнее

[1] См. *https://www.tinybasic.de*.

о виртуальных машинах в главе 5), а более поздние версии Ruby включают в себя компилятор just-in-time (JIT).

Независимо от того, проходит ли реализация языка AST, использует байт-код или имеет компилятор JIT, она нуждается в принципах, рассмотренных в этой главе.

Упражнения

1. Сделайте возможным появление экранированных двойных кавычек в строках NanoBASIC. Для этого потребуется немного разобраться в регулярных выражениях.
2. Превратите NanoBASIC в Tiny BASIC путем реализации операторов `INPUT`, которые дают пользователю возможность вводить числовое значение, сохраняемое в переменной. Этого достаточно для реализации «базового» Tiny BASIC (и для возможности запускать программы Tiny BASIC, найденные в интернете, с помощью вашего интерпретатора), но многие реальные версии также имели дополнительные возможности, такие как способ генерации случайных чисел, называемый `RND()`.
3. Tiny BASIC работает в интерактивном режиме, который поддерживает команды (рассматриваемые как операторы в исходной грамматике) `CLEAR`, `LIST`, `RUN` и `END`. Строки, начинающиеся с номера строки, сохраняются. Эти данные можно очистить, вывести на экран или запустить. Программу также можно завершить. Создайте интерактивный режим (по сути, REPL) для NanoBASIC с этими же четырьмя командами. Выше я уже говорил, что упражнение 2 превратит NanoBASIC в Tiny BASIC, но с этим интерактивным режимом вы сможете реализовать Tiny BASIC в полном объеме.
4. Добавьте поддержку интерполяции строк в NanoBASIC. Когда строка содержит знак доллара перед идентификатором, проверьте, определен ли этот идентификатор в таблице переменных, и выведите его значение вместо него. Например, «`Значение X равно $X`» выведет «`Значение X равно 24`», если переменная `X` имеет значение `24`. Для этого потребуется изменить токенизатор, парсер и среду выполнения.
5. Напишите программу NanoBASIC, которая делает что-то интересное и тестирует каждое выражение в интерпретаторе. Считайте это окончательным интеграционным тестом.

ЧАСТЬ II

ИСКУССТВО И ВЫЧИСЛЕНИЯ

3

РЕТРООБРАБОТКА ИЗОБРАЖЕНИЙ



Что делать, если нужно отобразить изображение на дисплее с меньшим количеством цветов, чем содержится в самом изображении? Решение этой проблемы связано с алгоритмами так называемого *дизеринга* (dithering), которые позволяют стратегически использовать ограниченную цветовую палитру для создания иллюзии большего количества цветов. В этой главе мы напишем программу, которая может взять любую современную фотографию и отобразить ее на экране классического монохромного компьютера Macintosh. Она преобразует фотографию в 1-битную черно-белую версию с дизерингом и экспортирует ее в формат, который может читать ранний Macintosh: MacPaint. По ходу работы мы изучим алгоритм дизеринга, алгоритм сжатия, а также познакомимся с файловыми форматами.

Что такое дизеринг?

Алгоритмы дизеринга позволяют намеренно вносить в изображение специфический шум, благодаря чему картинка выглядит более насыщенной, чем на самом деле. Этот способ обмана человеческого зрения имеет как практическое, так и художественное применение. Если вы когда-нибудь видели полноэкранную графику на игровой консоли или компьютере начала 1990-х годов, то, вероятно, имеете представление о том, как выглядит дизеринг. Эта техника также широко используется в анимированных файлах GIF, поскольку формат GIF поддерживает только 256 цветов (существует хитроумный способ

получить более 256 цветов в файле GIF, но большинство программ экспорта его не поддерживают).

Если вы посмотрите на рис. 3.1, то увидите одно и то же изображение в форматах JPEG и GIF. В JPEG отображается 45 807 цветов, а в GIF – только 256. Благодаря эффекту дизеринга разница не так заметна, как можно было бы ожидать. (Если вы читаете эту книгу в печатном виде, цветные версии изображений можно найти в каталоге *figures* репозитория GitHub нашей книги).



Рис. 3.1. В JPEG 45 807 цветов, в GIF – 256 цветов с эффектом дизеринга

Другим распространенным вариантом использования дизеринга и подобных ему техник является придание черно-белому изображению оттенков серого. Газеты уже давно используют нечто похожее на дизеринг (технику, называемую «*полутоны*») для воспроизведения фотографий на своих сугубо черно-белых печатных машинах. На компьютерных устройствах технология дизеринга позволяет создавать изображения с глубиной даже на 1-битных экранах, которые могут отображать только два цвета (обычно черный и белый). Эта технология актуальна и сегодня. Например, выпущенная в 2022 году игровая консоль Panic Playdate имеет 1-битный черно-белый экран. Большинство устройств Amazon Kindle поддерживают 16 оттенков серого, поэтому многие обложки книг и фотографии, отображаемые на Kindle, должны получить приближенный вид с помощью дизеринга (хотя и не 1-битного).

Оригинальная модель Apple Macintosh 1984 года имела 1-битный черно-белый экран, как и несколько последующих моделей. В действительности Apple продолжала продавать 1-битные Macintosh до тех пор, пока в 1993 году не было прекращено производство Classic II, а компания отказалась от поддержки пользователей Classic II только в 2001 году. За эти 17 лет поддержки сторонние разработчики создали множество интересных графических изображений для монохромных Macintosh с использованием алгоритмов дизеринга.

В нашем проекте мы сосредоточимся на этих классических монохромных компьютерах Macintosh, а также на любимом графическом

редакторе MacPaint, который они запускали. На рис. 3.2 показан конечный результат: изображение из рис. 3.1, отображенное в MacPaint, который запущен на эмуляторе Mac Plus. (Mac Plus 1986 года был небольшим усовершенствованием оригинального Macintosh 1984 года, но имел те же ограничения по разрешению экрана.) Рисунок был создан с помощью программы, построенной в этой главе.

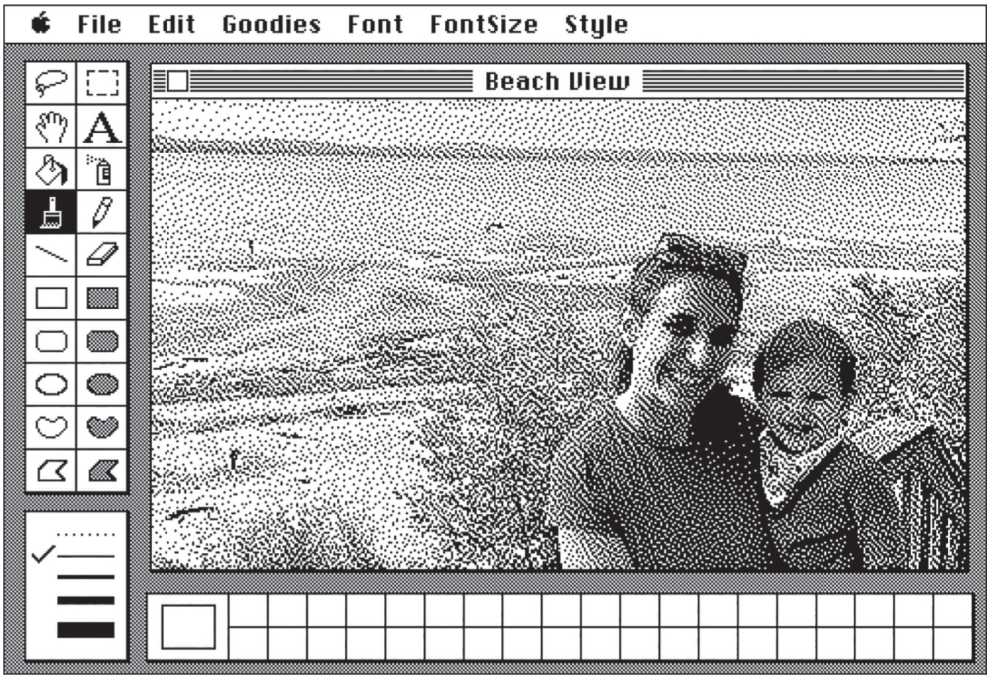


Рис. 3.2. Пляжная сцена из рис. 3.1, преобразованная нашей программой для показа в MacPaint

Вы можете заметить, что пляжная сцена на рис. 3.2 выглядит более «увеличенной», чем на рис. 3.1. Для рис. 3.1 я сначала уменьшил разрешение сцены, чтобы обе версии могли поместиться рядом друг с другом на одном рисунке (я использовал программное обеспечение для подсчета цветов версий с более низким разрешением). Я пропустил исходное изображение с полным разрешением через нашу программу, чтобы преобразовать его в формат MacPaint для рис. 3.2. В MacPaint можно работать только с документами размером до 576 пикселей в ширину и 720 пикселей в высоту, поэтому, как мы увидим, наша программа всегда будет предварительно изменять размер изображений, чтобы они соответствовали этим ограничениям. Однако окно отображения MacPaint на Mac Plus еще меньше, потому что дисплей Mac Plus имеет ширину всего 512 пикселей, а часть этих пикселей занимает панель инструментов.

Поэтому на рис. 3.2 мы видим только примерно 400 из 576 пикселей по ширине. Полная высота изображения также скрыта.

Начало работы

Наш проект будет следовать довольно простому алгоритму:

1. Считывание изображения с диска.
2. Изменение размера и преобразование в оттенки серого.
3. Дизеринг в черно-белое изображение.
4. Запись на диск в формате MacPaint.

Уникальными и наиболее интересными этапами проекта являются шаги 3 и 4, поэтому для выполнения шагов 1 и 2 мы воспользуемся библиотекой. Наиболее популярной библиотекой для работы с изображениями в мире Python, пожалуй, можно считать Pillow. Установить ее очень просто: `pip install pillow`.

С помощью Pillow можно прочитать изображение в любом популярном формате с помощью одной строки кода, а для изменения размера изображения и преобразования его в оттенки серого потребуется всего несколько строк. Мы выполним эту простую подготовительную работу прямо в нашем файле *__main__.py*, вместе с обработкой аргументов командной строки, как мы делали в двух предыдущих проектах. Начнем с кода для изменения размера и преобразования в оттенки серого, который находится в функции с подходящим названием `prepare()`:

```
# RetroDither/__main__.py
from PIL import Image
from argparse import ArgumentParser
from RetroDither.dither import dither
from RetroDither.macpaint import MAX_WIDTH, MAX_HEIGHT, write_macpaint_file

def prepare(file_name: str) -> Image.Image:
    with open(file_name, "rb") as fp:
        image = Image.open(fp)
        # Размер в пределах максимума для MacPaint
        if image.width > MAX_WIDTH or image.height > MAX_HEIGHT:
            desired_ratio = MAX_WIDTH / MAX_HEIGHT
            ratio = image.width / image.height
            if ratio >= desired_ratio:
                new_size = (MAX_WIDTH, int(image.height * (MAX_WIDTH / image.width)))
            else:
                new_size = (int(image.width * (MAX_HEIGHT / image.height)), MAX_HEIGHT)
            image.thumbnail(new_size, Image.Resampling.LANCZOS)
        # Преобразование в оттенки серого
        return image.convert("L")
```

Как отмечалось ранее, изображения MacPaint ограничены разрешением 576 пикселей в ширину и 720 пикселей в высоту. Мы определяем эти значения как константы в *macpaint.py*. В `prepare()`, если изображение слишком большое, мы масштабируем его в пропорциональ-

ном соотношении. Для этого мы вычисляем соотношение ширины изображения к его высоте и сравниваем его с соотношением максимальных размеров MacPaint. Масштабируя одно измерение до максимального допустимого в MacPaint размера, а другое в соответствии с пропорциями, получаем окончательное максимально крупное для MacPaint изображение без необходимости обрезания каких-либо его фрагментов. Чтобы вычислить формулу изменения размера, я провел несколько простых алгебраических вычислений на бумаге. Рекомендую вам сделать то же самое, если любопытно, как это работает.

Как и чтение файла изображения, изменение размера в Pillow – это простая однострочная операция с помощью метода `image.thumbnail()`. Он предлагает несколько встроенных алгоритмов для фактического расчета цветов каждого из пикселей, размер которых был изменен; вероятно, `LANCZOS` – это алгоритм с самым высоким качеством (при некоторой потере производительности). Наконец, `image.convert("L")` преобразует изображение в оттенки серого. Для режима оттенков серого используется сокращение `"L"` от слова *luminance* (яркость), которое в компьютерной графике также зачастую называют просто *luma*.

Теперь давайте разберемся с аргументами командной строки:

```
if __name__ == "__main__":
    argument_parser = ArgumentParser("RetroDither")
    argument_parser.add_argument("image_file", help="Input image file.")
    argument_parser.add_argument("output_file", help="Resulting MacPaint file.")
    argument_parser.add_argument('-g', '--gif', default=False, action='store_true',
                                 help='Create an output gif as well.')
    arguments = argument_parser.parse_args()
    original = prepare(arguments.image_file)
    dithered_data = dither(original)
    if arguments.gif:
        out_image = Image.frombytes('L', original.size, dithered_data.tobytes())
        out_image.save(arguments.output_file + ".gif")
    write_macpaint_file(dithered_data, arguments.output_file, original.width, original.
height)
```

На данный момент, в нашем третьем проекте, мы уже достаточно хорошо познакомились с `ArgumentParser`. Для проекта «Ретрообработка» (Retro Dither) у нас есть еще пара опций командной строки в сравнении с предыдущими проектами. Одна из них, `output_file`, предназначена для того, чтобы пользователь мог указать имя выходного файла и путь для результатов. Дополнительный параметр `-g` или `--gif` предназначен для того, чтобы пользователь мог указать, хочет ли он получить на выходе файл формата GIF в дополнение к файлу формата MacPaint. Если пользователь запрашивает вывод в формате GIF, мы используем Pillow для записи GIF-версии изображения с дизерингом.

Библиотека Pillow является очень мощной и многофункциональной. В этой главе мы не будем использовать многие из ее функций, но если вам понадобится работать с изображениями в Python, то стоит потратить время на ее изучение. Мы также вернемся к ней в следующей главе. Для получения дополнительной информации ознакомьтесь с документацией Pillow по адресу *https://pillow.readthedocs.io*.

Алгоритм дизеринга

Существует много различных алгоритмов дизеринга, из которых наиболее популярными являются алгоритмы *стохастического распределения ошибок*, или *стохастического выравнивания* (error-diffusion). Этот тип алгоритмов берет некоторую величину разницы между конечным и начальным положениями пикселя (*ошибку*) и распределяет ее (*производит диффузию*) между соседними пикселями. Самый популярный алгоритм стохастического распределения ошибок известен под названием дизеринга Флойда–Стейнберга (Floyd-Steinberg dithering) и был разработан Робертом Флойдом (Robert Floyd) и Луисом Стейнбергом (Louis Steinberg) в 1976 году. В Pillow есть встроенная поддержка дизеринга Флойда–Стейнберга.

Просто использовать Pillow для дизеринга было бы неинтересно, и мы бы ничего не узнали в процессе. Вместо этого мы реализуем алгоритм, которого нет в Pillow: алгоритм дизеринга Аткинсона (Atkinson dithering algorithm). Он был создан Биллом Аткинсоном (Bill Atkinson) в 1984 году специально для использования с программным обеспечением на оригинальном Macintosh, таким как MacPaint, автором которого был сам Аткинсон. Поэтому неудивительно, что дизеринг Аткинсона был популярной техникой на монохромных Mac. Он придаст нашим результатам более аутентичный вид. Как и алгоритм Флойда–Стейнберга, дизеринг Аткинсона является алгоритмом рассеивания ошибок. Как будет показано далее, для перехода от одного алгоритма рассеивания ошибок к другому требуется всего несколько изменений.

Прежде чем углубиться в особенности дизеринга Аткинсона, давайте поговорим об алгоритмах дизеринга с рассеиванием ошибок в целом. Большинство этих алгоритмов рассматривают пиксели изображения по одному, начиная с левого верхнего угла и заканчивая правым нижним. Движение происходит слева направо по каждой строке, а затем вниз по одной строке после завершения работы с каждой строкой. Для каждого обработанного пикселя выполняются следующие шаги:

1) определение цвета, которому он наиболее близок (при дизеринге выходные цвета задаются заранее, например черный и белый);
2) определение разницы между выходным цветом и исходным цветом пикселя;

3) добавление этой разницы к некоторым пикселям справа и ниже текущего пикселя.

Давайте воспользуемся этими общими шагами и применим их к нашему конкретному сценарию, где пиксели преобразуются из оттенков серого в черно-белые с помощью дизеринга Аткинсона:

1) определение того, что ближе к серому цвету в пикселе: черный или белый. Это должно основываться на некотором пороговом значении. Например, если оттенки серого хранятся в виде 8-битных целых чисел без знака, может быть 256 оттенков серого с номерами от 0 до 255, и наше пороговое значение может быть 127. Любой пиксель, превышающий 127, может быть помечен как белый (255 в этой схеме), а любой пиксель, меньший или равный 127, может быть помечен как черный (0);
2) вычитание разницы между новым цветом пикселя и его исходным цветом. Представьте, что исходный серый цвет был равен 204. Разница составит 255 – 204 = 51. Это и будет нашей погрешностью;
3) добавление одной восьмой ошибки к шести конкретным пикселям, близким к исходному. Это этап диффузии. В данном случае 51 // 8 = 6 (необходимо выполнить целочисленное деление). Корректируемые пиксели: один справа, два справа, один по диагонали вниз и влево, один по диагонали вниз и вправо, один прямо внизу и один на два ниже.

Единственное отличие между дизерингом Флойда–Стейнберга и дизерингом Аткинсона заключается в шаге 3. Обратите внимание, что поскольку мы распределяем одну восьмую ошибки между шестью пикселями, мы распределяем только шесть восьмых, то есть три четверти от общей ошибки. В дизеринге Флойда–Стейнберга *вся* ошибка распределяется между четырьмя соседними пикселями. Это различие в распределении ошибки придает дизерингу Аткинсона вид, который, по всей видимости, подчеркивает изменения контраста, в то время как дизеринг Флойда–Стейнберга может иметь более плавный эффект. На рис. 3.3 показаны исходное изображение (первая панель), дизеринг Аткинсона (вторая панель) и дизеринг Флойда–Стейнберга (третья панель).

В табл. 3.1 и 3.2 приведены матрицы, представляющие распределение ошибок в дизеринге Аткинсона и дизеринге Флойда–Стейнберга соответственно. Символ *X* обозначает исходное положение пикселя, а заголовки столбцов и строк обозначают смещение в столбцах или строках от исходного пикселя.

Δ	**–1**	**0**	**+1**	**+2**
0		X	1/8	1/8
+1	1/8	1/8	1/8	
+2		1/8		

Таблица 3.1 Рассеивание ошибок при дизеринге Аткинсона



Рис. 3.3. Бабушка автора на портрете, где хорошо виден контраст между различными цветами (вид краев при дизеринге)

Δ	–1	0	+1
0		X	7/16
+1	3/16	5/16	1/16

Таблица 3.2 Рассеивание ошибок при дизеринге Флойда–Стейнберга

Хотя мы будем реализовывать дизеринг Аткинсона, матрицу мы оставим отдельной, чтобы вам было проще подключить другую матрицу. Например, вы без труда сможете изменить код на дизеринг Флойда–Стейнберга или другой вариант диффузии ошибок либо попробовать свой собственный метод. Фактически это одно из упражнений в конце главы.

Наш код дизеринга начинается с определения некоторых констант:

RetroDither/dither.py

```
from PIL import Image
from array import array
from typing import NamedTuple

THRESHOLD = 127

class PatternPart(NamedTuple):
    dc: int # изменение в столбце
    dr: int # изменение в ряду
    numerator: int
    denominator: int

ATKINSON = [PatternPart(1, 0, 1, 8), PatternPart(2, 0, 1, 8),
            PatternPart(-1, 1, 1, 8), PatternPart(0, 1, 1, 8),
            PatternPart(1, 1, 1, 8), PatternPart(0, 2, 1, 8)]
```

Мы устанавливаем THRESHOLD равным 127 (примерно середина между 0 и 255), как описано в нашем алгоритме. В ATKINSON мы сглаживаем матрицу дизеринга Аткинсона в шесть кортежей PatternPart, каждый из которых указывает, где находится один из изменяемых пикселей по отношению к исходному пикселю и какая доля ошибки должна быть к нему добавлена.

Далее определим функцию dither(), в которой происходит дизеринг:

```
# Допустим, мы работаем с изображением в оттенках серого (режим "L" в Pillow)
# Возвращает массив пикселей с дизерингом (255 для белого, 0 для черного)
def dither(image: Image.Image) -> array:
    # Распределение ошибки среди ближайших пикселей
    def diffuse(c: int, r: int, error: int, pattern: list[PatternPart]):
        for part in pattern:
            col = c + part.dc
            row = r + part.dr
            if col < 0 or col >= image.width or row >= image.height:
                continue
            current_pixel: float = image.getpixel((col, row)) # type: ignore
            # Add *error_part* to the pixel at (*col*, *row*) in *image*
            error_part = (error * part.numerator) // part.denominator
            image.putpixel((col, row), current_pixel + error_part)
```

Функция dither() начинается со вспомогательной функции diffuse(), которая принимает ошибку и распределяет ее между соседними пикселями в заданных шаблоном участках. Шаблон, в свою очередь, представляет собой обычный список кортежей PatternPart. Мы просматриваем каждый PatternPart в шаблоне, находим соответствующий пиксель и добавляем к нему часть ошибки. Как уже упоминалось, шаблон является ключевым элементом, отличающим один алгоритм распределения ошибок от другого. Обратите внимание, что все арифметические операции в данном случае являются целочисленными. Это связано с тем, что значения пикселей хранятся в виде целых чисел.

Некоторые недостатки

В документации Pillow отмечается, что методы getpixel() и putpixel() работают довольно медленно. В качестве альтернативы Pillow предлагает более прямой доступ к данным пикселей в массиве. Кроме того, проход по всем кортежам PatternPart менее эффективен, чем хранение необработанного массива координат. На самом деле, поскольку каждая error_part составляет ровно одну восьмую от общей погрешности в дизеринге Аткинсона, мы

могли бы сэкономить на множестве вычислений, просто поделив `error` один раз на 8. Однако этот код не стремится быть максимально эффективным, он должен быть максимально понятным для главы в книге, посвященной алгоритмам. Мы также хотим иметь возможность подключать различные алгоритмы диффузионного дизеринга, в которых `error_part` не всегда будет одной и той же.

Несмотря на эти недостатки, так как сначала нам нужно масштабировать каждое изображение для соответствия ограничениям MacPaint, скорость работы всей программы практически мгновенная. Если бы мы выполняли дизеринг более крупных изображений, сделанные здесь послабления могли бы стать серьезной проблемой.

В остальной части `dither()` мы просматриваем каждый пиксель изображения, меняем его на черный или белый в зависимости от `THRESHOLD`, вычисляем разницу по сравнению с исходным серым цветом и распределяем ошибку между соседними пикселями с помощью вспомогательной функции `diffuse()`:

```
    result = array('B', [0] * (image.width * image.height))
    for y in range(image.height):
        for x in range(image.width):
            old_pixel: float = image.getpixel((x, y)) # type: ignore
            # Каждый новый пиксель является либо полностью белым, либо полностью
            # черным, поскольку это все, что поддерживал оригинальный Macintosh
            new_pixel = 255 if old_pixel > THRESHOLD else 0
            result[y * image.width + x] = new_pixel
            difference = int(old_pixel - new_pixel)
            # Распределение ошибки между соседними пикселями
            diffuse(x, y, difference, ATKINSON)

    return result
```

В настоящее время шаблон `ATKINSON` жестко запрограммирован, но здесь есть простой способ перейти на другой шаблон. Обратите внимание, что переменная `result` на самом деле является массивом пикселей, а не еще одним изображением Pillow. Это связано с тем, что нам потребуется дополнительно обработать необработанные пиксельные данные изображения с дизерингом, чтобы сохранить его в формате MacPaint. В стандартной библиотеке Python тип `array`, определенный с помощью кода типа `'B'`, содержит байты без знака. Поскольку мы будем работать с большим количеством необработанных байтов, мы неоднократно будем встречать этот тип массива как в этой главе, так и в последующих главах, посвященных эмуляторам.

Что может сбить с толку, так это то, что Python предоставляет как минимум три разных типа для работы с необработанными байтами: `bytes`, `bytearray` и `array("B")`. В конечном итоге вы можете написать свой код с использованием любого из них. Тип `array` особенно хорошо подходит для компактного представления и работы с файлами.

Файловый формат MacPaint

Программы для рисования существовали и до этого, но MacPaint установил стандарт для всех последующих программ. Он был написан Биллом Аткинсоном и выпущен в 1984 году компанией Apple вместе с оригинальным Macintosh. Хотя Macintosh предшествовали Xerox Star и Apple Lisa, он был первым широко доступным персональным компьютером с мышкой и графическим интерфейсом пользователя (GUI). Приложение MacPaint было одной из демонстрационных программ, которые показали преимущества использования мышки и графического интерфейса. Один из обозревателей *New York Times* написал: «Это в 10 раз превосходит все другие программы подобного рода, предлагаемые для персональных компьютеров»[1]. Многие инструменты и методы графической обработки, впервые появившиеся в MacPaint, до сих пор используются в современном графическом программном обеспечении.

Я рекомендую вам поработать с MacPaint и оценить его возможности. В интернет-архиве есть его живая демонстрация[2], но вы сможете извлечь максимальную пользу из этой главы, если найдете время для скачивания эмулятора, чтобы иметь возможность загружать реальные изображения, которые наша программа будет выводить непосредственно в MacPaint. Вы даже можете найти старый Mac на eBay! Я сам собрал их огромное количество.

Несмотря на свою революционность, MacPaint был ограничен аппаратным обеспечением своей пионерской платформы. Как и оригинальный Macintosh, MacPaint поддерживал только черно-белый режим. Как упоминалось ранее, его документы также были ограничены фиксированным размером 576 пикселей в ширину и 720 пикселей в высоту. Оригинальный Macintosh даже не имел жесткого диска; дисковое пространство было ограничено, потому что все должно было работать с дискеты. Чтобы справиться с этим ограничением, программа MacPaint использовала простую схему сжатия, известную как *кодирование по длине серии* (run-length encoding). Мы вернемся к некоторым из этих особенностей чуть позже.

Следующим шагом в нашем проекте будет написание кода, который преобразует пиксели изображения с дизерингом в формат MacPaint.

---

[1] Erik Sandberg-Diment, «Software for the Macintosh: Plenty on the Way», New York Times, 31 января 1984 г.

[2] См. *https://archive.org/details/mac_Paint_2*.

Как показывает мой опыт, программирование с использованием двоичных форматов файлов – это просто вопрос очень тщательного следования спецификации. К сожалению, формат MacPaint сегодня несколько непонятен, поэтому поиск спецификации требует некоторых усилий. Сама Apple описала этот формат в техническом примечании PT24[1]. Однако самое доступное и полное описание, которое я нашел, было на сайте FileFormat.info[2].

На первый взгляд файл MacPaint довольно прост. Он состоит из 512-байтового заголовка, за которым следуют данные пикселей, сжатые с помощью кодирования по длине серии. Перед кодированием по длине серии каждый пиксель хранится в виде бита: 1 для черного цвета или 0 для белого. Все это звучит достаточно просто.

Однако есть одна особенность: в отличие от почти всех других операционных систем, классическая операционная система Mac хранила файлы в двух «развилках» (forks). Подробнее об этом мы поговорим чуть позже, но если говорить кратко, файлы MacPaint, которые переносятся или создаются в операционных системах, отличных от классической Mac OS, должны быть закодированы в специальном формате под названием *MacBinary*, чтобы сохранить свои метаданные при переносе. Поэтому нам придется преобразовать наш выходной файл в файл MacBinary, добавив специальный дополнительный заголовок.

Мы будем разбираться с этим специфическим форматом файлов шаг за шагом. Сначала мы обработаем данные пикселей. Затем реализуем кодирование длины последовательности. И наконец, мы создадим заголовки MacPaint и MacBinary. В процессе работы нам понадобятся различные побитовые операции, включая сдвиги, OR и AND. Если вы не сталкивались с такими видами низкоуровневых операций с битами или просто немного забыли их, обратитесь к приложению книги, где приведен обзор побитовых операций в Python.

Преобразование байтов в биты

В режиме `"L"` наш `Image` в Pillow кодируется с использованием 1 байта на пиксель. Однако в растровом изображении MacPaint пиксели кодируются с использованием 1 бита на пиксель, то есть каждый байт представляет восемь пикселей. Это значительная экономия для черно-белого изображения, и это вполне логично, поскольку нам нужно только два значения (1 и 0) для представления двух цветов. После определения некоторых констант наша первая функция в *macpaint.py* – это конвертер, который принимает массив байтов, полученный от `dither()`, и преобразует его в «массив битов»:

---

[1] «Technical Note PT24: MacPaint Document Format», Apple, 1 октября 1988 г., доступ 5 августа 2022 г., *https://web.archive.org/web/20040626093131/http://developer.apple.com/technotes/pt/pt_24.html*.

[2] См. *https://www.fileformat.info/format/macpaint/egff.htm*.

RetroDither/macpaint.py

```
from array import array
from pathlib import Path
from datetime import datetime

MAX_WIDTH = 576
MAX_HEIGHT = 720
MACBINARY_LENGTH = 128
HEADER_LENGTH = 512

# Преобразование массива байтов, где каждый байт равен 0 или 255,
# в массив битов, где каждый байт, равный 0, становится 1,
# а каждый байт, равный 255, становится 0
def bytes_to_bits(original: array) -> array:
    bits_array = array('B')

    for byte_index in range(0, len(original), 8):
        next_byte = 0
        for bit_index in range(8):
            next_bit = 1 - (original[byte_index + bit_index] & 1)
            next_byte = next_byte | (next_bit << (7 - bit_index))
            if (byte_index + bit_index + 1) >= len(original):
                break
        bits_array.append(next_byte)
    return bits_array
```

Циклы здесь проходят последовательно по 8 байтам из массива `original`, проверяя, является ли каждый из них белым (255) или черным (0) пикселем. Формат растрового изображения MacPaint инвертирует это, переводя белый пиксель в 0, а черный в 1. Инверсию выполняет строка `next_bit = 1 - (original[byte_index + bit_index] & 1)`. Обратите внимание, что мы на самом деле не проверяем значения по отношению к 255, а предпочитаем проверять только первый бит (`& 1`), потому что это делает код более компактным и производительным. Мы знаем, что перед вызовом `bytes_to_bits()` в массиве хранились только значения 255 и 0, поэтому нет причин опасаться, что мы случайно захватим промежуточные значения; если где-либо в байте есть 1, то это должно быть 255. Эти 8 бит кодируются в `next_byte` путем помещения каждого бита в соответствующее место с помощью операции OR: `next_byte = next_byte | (next_bit << (7 - bit_index))`. Затем мы добавляем `next_byte` к `bits_array`.

Мы не вызываем `bytes_to_bits()` сразу для всех пиксельных данных, потому что растровые изображения MacPaint необходимо заполнять белыми пикселями (0) в каждой строке растрового изображения, где пиксельные данные не занимают всю длину. Мы обрабатываем заполнение с помощью функции `prepare()`:

```
# Преобразование массива байтов в биты с помощью вспомогательной функции.
# Заполнение всех недостающих мест белыми битами из-за того, что исходное
# изображение имеет размер менее 576x720.
def prepare(data: array, width: int, height: int) -> array:
    bits_array = array('B')
    for row in range(height):
        image_location = row * width
        image_bits = bytes_to_bits(data[image_location:(image_location + width)])
        bits_array += image_bits
        remaining_width = MAX_WIDTH - width
        white_width_bits = array('B', [0] * (remaining_width // 8))
        bits_array += white_width_bits
    remaining_height = MAX_HEIGHT - height
    white_height_bits = array('B', [0] * ((remaining_height * MAX_WIDTH) // 8))
    bits_array += white_height_bits
    return bits_array
```

Мы просматриваем необработанные данные пикселей из `dither()` по одной строке за раз и преобразуем строку в биты с помощью `bytes_to_bits()`. Если строка не занимает всю ширину документа MacPaint, мы добавляем в нее белые пиксели. То же самое мы делаем для всех строк полной длины под данными пикселей.

Реализация кодирования по длине последовательности

Хранение пикселей в виде отдельных битов, а не байтов, позволяет сэкономить значительное количество пространства, но этого недостаточно. Когда в 1984 году был выпущен MacPaint, оригинальные компьютеры Macintosh имели дисководы для хранения всего 400 КБ данных. Жестких дисков не было, а стандартная конфигурация включала только один дисковод. Представьте, каким большим был бы файл MacPaint без сжатия. 576 пикселей в строке занимают 576 бит, что составляет 72 байта. Всего 720 строк, поэтому 720, умноженное на 72 байта, дает 51 840 байт. Если добавить 512-байтовый заголовок, файл MacPaint без сжатия будет иметь размер 52 352 байта. Это означает, что на дискету не поместится даже восемь файлов MacPaint!

Для решения этой проблемы с дисковым пространством файловый формат MacPaint использует простую схему сжатия, называемую кодированием по длине последовательности (run-length encoding). В этой схеме вместо повторения одного и того же элемента указывается количество его повторений. Например, предположим, что мы хотим сохранить строку *AAAAAABCCCCCABBBB*. Если каждый символ занимает 1 байт, то она будет занимать 17 байт. Не было бы эффективнее указать семь букв *A* вместо повторения буквы *A* семь раз? Или указать пять букв *C*?

Предположим, что для хранения количества повторений перед символом используется до 1 байта. Тогда строка может быть закоди-

рована как *6AB5CA4B*, что составляет 8 байт. Однако у этой схемы есть проблема. Помните, что в памяти компьютера это будут необработанные байты. Как же мы узнаем, что *B* – это символ, а не байт, обозначающий определенное количество повторений следующего символа? Другими словами, *B*, скорее всего, будет храниться в памяти с использованием его кода ASCII/Unicode, 66. Наша программа, вероятнее всего, интерпретирует его как указание на 66 повторений следующего символа, а не на одну букву *B*.

Вместо этого можно использовать схему, в которой каждому символу, даже одиночному, предшествует число. В этом случае закодированная строка изменится на *6A1B5C1A4B*. Это 10 байт, что на 7 байт меньше, чем в исходном варианте, а это уже весьма значительная экономия.

Такая схема кодирования является одной из форм кодирования по длине серии, но она довольно быстро теряет свою эффективность: для многих строк она на самом деле менее эффективна, чем простое хранение исходных символов. Например, строка *ABC* будет кодироваться как *1A1B1C*. Это вдвое больше исходного размера.

Есть компромиссный вариант. Можно использовать схему кодирования, в которой каждое число обозначает либо повторение, либо определенное количество символьных литералов. Тогда *ABC* станет *3ABC*. Это все еще длиннее, но эта новая схема может быть хорошим компромиссом для более сложных случаев. Например, строка *AAAAABCBCAAAAAA* будет выглядеть как *5A4BCBC6A*.

Эта версия близка к схеме кодирования, используемой в формате файлов MacPaint, но все еще остается проблема: как понять, что 5 означает пять букв *A*, а 4 означает буквальную последовательность из четырех символов (*BCBC*), а не четыре буквы *B*? Вы можете сказать: «Что ж, можно считать два символа дальше и увидеть, что *C* не является цифрой, поэтому 4 не может означать четыре буквы *B*», но даже это не сработает, потому что (опять же) в памяти компьютера *C* хранится как цифра. Нам нужно еще одно улучшение.

В MacPaint устраняется неоднозначность между числами, обозначающими буквальные последовательности, и числами, обозначающими повторяющиеся последовательности, путем представления обоих типов с помощью 1-байтовых целых чисел со знаком. Число *n* от 0 до 127 обозначает, что за ним следует *n* + 1 литеральных байтов. Число *n* от –1 до –127 указывает на 1 – *n* повторений следующего байта. Число –128 не используется. Эта схема сжатия известна как *PackBits*[1], и она использовалась не только в MacPaint, но и в нескольких других популярных файловых форматах.

Функция PackBits была встроена в классические версии Mac OS. Согласно технической заметке Apple по этому вопросу, «типичные

[1] См. *https://en.wikipedia.org/wiki/PackBits*.

документы MacPaint сжимаются до размера около 10 КБ» с помощью PackBits[1]. В табл. 3.3 приведена схема кодирования PackBits.

Таблица 3.3 Схема кодирования PackBits (со знаком)

Число *n*	**Значение**
От 0 до 127	Следуют *n* + 1 байт литерала
От –1 до –127	Следующий байт повторяется 1 – *n* раз
–128	Пропуск

Мы будем работать с целыми числами без знака, поэтому удобнее переписать таблицу, чем выполнять преобразование каждого байта из числа со знаком в число без знака (или думать о дополнении до двух). В табл. 3.4 представлена схема кодирования с преобразованием байтов в целые числа без знака.

Таблица 3.4 Схема кодирования PackBits (без знака)

Число *n*	**Значение**
От 0 до 127	Следуют *n* + 1 байт литерала
От 129 до 255	Следующий байт повторяется 1 – *n* раз
128	Пропуск

Задумывались ли вы об ограничениях этой схемы? Что делать, если в строке содержится более 128 байт? К счастью, при кодировании файлов MacPaint мы не столкнемся с этой проблемой, поскольку они кодируются по одной строке за раз. Строка в MacPaint может содержать только 576 пикселей, которые хранятся в виде 72 байт. Поскольку 72 меньше 128, нам не нужно беспокоиться об ограничении.

Чтобы проверить правильность этой трактовки, попробуйте поработать с примером, приведенным Apple в техническом примечании TN1023. Для удобства я преобразовал его из шестнадцатеричного в десятичное число в табл. 3.5. Одна строка – это распакованные (unpacked) данные, а другая – упакованные (packed). Попробуйте переформатировать упакованные данные из распакованных с использованием табл. 3.4 в качестве справочника. Затем проверьте свою работу по сравнению с упакованными данными в табл. 3.5.

Таблица 3.5 Пример PackBits

Типы	**Байты**
Распакованные	170, 170, 170, 128, 0, 42, 170, 170, 170, 170, 128, 0, 42, 34, 170, 170, 170, 170, 170, 170, 170, 170, 170, 170
Упакованные	254, 170, 2, 128, 0, 42, 253, 170, 3, 128, 0, 42, 34, 247, 170

[1] «Technical Note TN1023: Understanding PackBits», Apple, 1 ноября 1987 г., доступ 4 августа 2022 г., *https://web.archive.org/web/20030218001420/http://developer.apple.com/technotes/tn/tn1023.html*.

Давайте применим кодировщик PackBits. Функция `run_length_encode()` принимает массив байтов и возвращает массив байтов, закодированный по длине последовательности, с использованием схемы PackBits. Она начинается с внутренней вспомогательной функции `take_same()`, которая может находить последовательности повторяющихся значений и возвращать их длину:

```
# MacPaint ожидает, что RLE будет применяться построчно (MAX_WIDTH).
# Другими словами, границы строк существуют.
def run_length_encode(original_data: array) -> array:
    # Find how many of the same bytes are in a row from *start*
    def take_same(source: array, start: int) -> int:
        count = 0
        while (start + count + 1 < len(source)
               and source[start + count] == source[start + count + 1]):
            count += 1
        return count + 1 if count > 0 else 0
```

Чтобы найти последовательность, `take_same()` просто перебирает байты, проверяя, совпадает ли следующий байт с текущим. При этом функция следит за тем, чтобы не выйти за пределы массива `source`. Каждый раз, когда находится совпадение, мы увеличиваем `count`, но поскольку первый проверенный байт (байт на `start`) сам по себе не совпадает с предыдущим байтом, `count` всегда будет на единицу меньше, чем количество элементов в последовательности. Следовательно, `count + 1` возвращается, если найдены совпадения, или `0` в противном случае. Это делает невозможным повторение последовательности с одним и тем же символом: это будет просто одиночный символ, который будет частью литерала, поскольку в PackBits нет возможности «повторить один раз». Другими словами, область определения `take_same()` – это 0 и все целые числа, большие или равные 2.

Функция `run_length_encode()` продолжается с небольшой настройкой:

```
    rle_data = array('B')
    # Разделение данных на границы размера MAX_WIDTH по строкам
    for line_start in range(0, len(original_data), MAX_WIDTH // 8):
        data = original_data[line_start:(line_start + (MAX_WIDTH // 8))]
```

Результат будет сохранен в `rle_data`. Мы просматриваем массив `original_data` по одной строке за раз. Формат файла MacPaint определяет, что каждая строка кодируется индивидуально по длине, а не все пиксели кодируются по длине сразу. Переменная `data` представляет одну строку пиксельных данных, готовых для кодирования по длине. Следующим шагом является поиск повторений и литеральных последовательностей:

```
        index = 0
        while index < len(data):
            not_same = 0
            while (((same := take_same(data, index + not_same)) == 0)
                and (index + not_same < len(data))):
                not_same += 1
```

Мы итеративно просматриваем `data` в каждой строке по 1 байту за раз, при этом `index` отслеживает текущий байт, который проверяется. Собираются два параметра: `same` (инициализируется на лету с помощью так называемого «оператора моржа» (walrus) `:=`) – это количество одинаковых элементов в строке, а `not_same` – это длина литеральной последовательности. Вот как они подсчитываются в цикле `while`:

1. Мы пытаемся найти повторяющийся отрезок с помощью `take_same()`.
2. Если попытка не удалась (`same` равен `0`), то это должна быть литеральная последовательность, поэтому `not_same` увеличивается.
3. Шаги 1 и 2 повторяются до тех пор, пока не будет найдена повторяющаяся последовательность (`same` не равен `0`) или рассматриваемый байт (`index + not_same`) не выйдет за пределы строки.

После этого цикла возможны три варианта:

1. Повторяющаяся последовательность находится сразу (`same` не равен `0`), и `not_same` не увеличивается, то есть равен `0`.
2. Сначала находится литеральная последовательность, и `not_same` увеличивается до тех пор, пока не будет найдена повторяющаяся последовательность, заполняющая `same`.
3. Изначально находится литеральная последовательность, которая проходит до конца строки, и цикл завершается, поскольку `index + not_same < len(data)` не выполняется.

Из-за второй возможности может возникнуть сценарий, при котором `same` и `not_same` будут больше `0`. Имейте это в виду, когда будем рассматривать остальную часть кода данной функции:

```
            if not_same > 0:
                rle_data.append(not_same - 1)
                rle_data += data[index:index + not_same]
                index += not_same
            if same > 0:
                rle_data.append(257 - same)
                rle_data.append(data[index])
                index += same
    return rle_data
```

Это часть, которая записывает закодированные PackBits данные в массив. Эти шаблоны взяты непосредственно из табл. 3.4. Благода-

ря возможности найти как not_same, так и same-последовательность в одной итерации цикла, здесь есть два оператора if вместо одного else. Кроме того, благодаря структуре внутреннего цикла while not_same runs всегда будут найдены раньше same. Поэтому оператор if для not_same появляется первым. Если есть по одной последовательности каждого типа, то index увеличивается на нужное количество not_same, чтобы оказаться в нужном месте для кодирования same.

Альтернативная реализация

Считаете ли вы функцию run_length_encode() достаточно элегантной или слишком заумной? Я несколько раз пытался переписать свою первоначальную версию в более компактной и удобочитаемой форме и в итоге получил то, что вы видите здесь. Я обнаружил, что концентрация кода на take_same() и простой подсчет случаев, когда он не срабатывает, делают код более удобочитаемым, чем одновременная попытка установить последовательности same и not_same. Однако эта версия менее эффективна, чем моя первоначальная, в которой используется гораздо больше условий, она немного длиннее и не имеет внутренней функции. Если вам не нравится эта версия, я оставил свою первоначальную версию в виде комментария внизу исходного файла на GitHub. Вы можете найти этот файл по адресу *https://github.com/davecom/ComputerScienceFromScratch/blob/main/RetroDither/macpaint.py*.

Тестирование кодирования по длине последовательности

Когда я пробовал переписать run_length_encoding() несколькими разными способами, чтобы сделать его более удобочитаемым, я понял, что мне нужен быстрый способ убедиться в правильности моих новых реализаций, поэтому я написал несколько модульных тестов. Как и все тесты для этой книги, файл с ними находится в каталоге *tests* в корневом каталоге репозитория исходного кода.

```
# tests/test_retrodither.py
import unittest
from array import array
from RetroDither.macpaint import run_length_encode

class RetroDitherTestCase(unittest.TestCase):
    # Пример из
    # web.archive.org/web/20080705155158/http://developer.apple.com/technotes/tn/tn1023.html
    def test_apple_rle_example(self):
        unpacked = array("B", [0xAA, 0xAA, 0xAA, 0x80, 0x00, 0x2A, 0xAA, 0xAA, 0xAA, 0xAA,
```

```
                            0x80, 0x00, 0x2A, 0x22, 0xAA, 0xAA, 0xAA, 0xAA, 0xAA, 0xAA,
                            0xAA, 0xAA, 0xAA, 0xAA])
        packed = run_length_encode(unpacked)
        expected = array("B", [0xFE, 0xAA, 0x02, 0x80, 0x00, 0x2A, 0xFD, 0xAA, 0x03, 0x80,
                            0x00, 0x2A, 0x22, 0xF7, 0xAA])
        self.assertEqual(expected, packed)

    # Пример, где упакованные данные длиннее, чем распакованные данные
    def test_longer_rle(self):
        unpacked = array("B", [0x55, 0x55, 0xBB, 0xBB, 0x55, 0xBB, 0xBB, 0x55])
        packed = run_length_encode(unpacked)
        expected = array("B", [0xFF, 0x55, 0xFF, 0xBB, 0x00, 0x55, 0xFF, 0xBB, 0x00, 0x55])
        self.assertEqual(expected, packed)

    def test_simple_literal(self):
        unpacked = array("B", [0x00, 0x01, 0x02, 0x03, 0x04])
        packed = run_length_encode(unpacked)
        expected = array("B", [0x04, 0x00, 0x01, 0x02, 0x03, 0x04])
        self.assertEqual(expected, packed)

    def test_simple_literal2(self):
        unpacked = array("B", [0x00])
        packed = run_length_encode(unpacked)
        expected = array("B", [0x00, 0x00])
        self.assertEqual(expected, packed)

    def test_simple_same(self):
        unpacked = array("B", [0x11, 0x11, 0x11, 0x11])
        packed = run_length_encode(unpacked)
        expected = array("B", [0xFD, 0x11])
        self.assertEqual(expected, packed)

    def test_simple_same2(self):
        unpacked = array("B", [0x11, 0x11, 0x11, 0x11, 0x22, 0x22, 0x22, 0x22])
        packed = run_length_encode(unpacked)
        expected = array("B", [0xFD, 0x11, 0xFD, 0x22])
        self.assertEqual(expected, packed)

if __name__ == "__main__":
    unittest.main()
```

Эти тесты гарантируют, что кодирование по длине последовательности работает правильно. На этом наша реализация формата файлов MacPaint завершена. Осталось сделать всего один шаг, чтобы наши картинки с дизерингом были готовы к использованию на ретрокомпьютерах Mac.

Преобразование в MacBinary

В большинстве операционных систем файл представляет собой всего лишь единый блок данных. Файловая система или операционная система могут хранить некоторые метаданные о каждом файле, но

сам по себе файл является самостоятельным. В классической Mac OS дело обстояло иначе: многие файлы имели два разветвления (форка). В *развилке данных* (data fork) хранились основные данные файла, а в *развилке ресурсов* (resource fork) могли храниться вспомогательные данные, такие как растровые изображения, звуки или даже исполняемый код. Развилка ресурсов могла хранить много разных типов данных в одном месте, поэтому она была чем-то вроде базы данных ресурсов. Это одна из особенностей операционной системы, которая позволяла многим приложениям быть полностью автономными – одним исполняемым файлом, который можно было просто перетащить как единственную иконку, без дополнительных файлов.

К сожалению, поскольку ресурсные развилки не существуют в других операционных системах, классические файлы Mac часто повреждаются при их переносе в другие системы или из них. Необходимо проявлять осторожность. Эта проблема существовала с самого начала, поэтому были быстро разработаны «связанные» форматы файлов. Один из самых популярных и хорошо стандартизированных форматов известен как MacBinary. В файле MacBinary за специальным заголовком следуют развилка данных и развилка ресурсов в совокупности.

Нам необходимо объединить наши файлы MacPaint в файлы MacBinary, чтобы они правильно работали на классической Mac OS и по умолчанию открывались в MacPaint. Как ни странно, файл MacPaint на самом деле не имеет ресурсной развилки, у него есть только развилка данных. Но файлы MacBinary также объединяют метаданные, которые хранились в файловой системе (MFS/HFS/HFS+) классической Mac OS. Это важные биты кодов *типа* (type) и *автора* (creator). Классическая Mac OS не использует расширения файлов для связывания файла с программой, которая должна его открыть. Вместо этого она использует коды типа и автора, которые в значительной степени прозрачны для пользователя. Это позволяет пользователю называть свои файлы как угодно, и при этом они все равно будут открываться «двойным щелчком». После того как созданный нашей программой файл MacBinary распаковывается с помощью MacBinary (или другой программы, такой как Stuffit) в классической Mac OS, полученный файл MacPaint должен открываться в MacPaint при двойном щелчке.

К счастью, формат файлов MacBinary довольно прост. Чтобы соответствовать спецификации MacBinary, наша программа должна:

1) добавить 128-байтовый заголовок MacBinary перед остальной частью файла (который представляет собой только развилку данных, поскольку у нас нет развилки ресурсов);
2) заполнить заголовок MacBinary правильными значениями в нескольких указанных местах;
3) убедиться, что файл заканчивается кратно 128 байтам, добавив в конце файла необходимые данные, если это потребуется.

У MacBinary есть официальная спецификация, которая была одобрена комитетом заинтересованных сторон[1]. Однако для правильного распознавания нашего файла MacBinary нам нужно заполнить только несколько полей. (Остальная часть заголовка должна состоять из нулей.) В табл. 3.6 перечислены эти значения, их длины и соответствующие смещения в 128-байтовом заголовке.

Таблица 3.6 Требуемые поля заголовка MacBinary

Сдвиг	Длина	Тип	Значение
1	1	Целое число	Длина имени файла (до 63)
2	От 1 до 63	MacRoman	Имя файла
65	4	MacRoman	Тип файла (должно быть “PNTG”)
69	4	MacRoman	Автор файла (должно быть “MPNT”)
83	4	Целое число	Длина развилки данных
91	4	Целое число	Время создания в секундах начиная с 1/1/1904
95	4	Целое число	Время изменения в секундах начиная с 1/1/1904

Обратите внимание, что значения хранятся в формате big-endian, поскольку классическая Mac OS работала на микропроцессорах big-endian. (Если вам это ничего не говорит, см. вставку «Big-Endian в сравнении с Little-Endian»). MacRoman – это кодировка символов, используемая в классической Mac OS[2].

Функция `macbinary_header()` является кодификацией табл. 3.6:

```
def macbinary_header(outfile: str, data_size: int) -> array:
    macbinary = array('B', [0] * MACBINARY_LENGTH)
    filename = Path(outfile).stem
    filename = filename[:63] if len(filename) > 63 else filename # limit to 63 characters max
    macbinary[1] = len(filename) # длина имени файла
    macbinary[2:(2 + len(filename))] = array("B", filename.encode("mac_roman")) # filename
    macbinary[65:69] = array("B", "PNTG".encode("mac_roman")) # тип файла
    macbinary[69:73] = array("B", "MPNT".encode("mac_roman")) # автор файла
    macbinary[83:87] = array("B", data_size.to_bytes(4, byteorder='big')) # размер развилки данных
    timestamp = int((datetime.now() - datetime(1904, 1, 1)).total_seconds()) # временная метка Mac
    macbinary[91:95] = array("B", timestamp.to_bytes(4, byteorder='big')) # время создания
    macbinary[95:99] = array("B", timestamp.to_bytes(4, byteorder='big')) # время изменения
    return macbinary
```

Любые имена файлов, длина которых превышает 63 символа, просто обрезаются в конце.

[1] «MacBinary II Standard», Stairways.com, 24 июля 1987 г., доступ 27 мая 2024 г., *https://files.stairways.com/other/macbinaryii-standard-info.txt*.

[2] См. *https://en.wikipedia.org/wiki/Mac_OS_Roman*.

Big-Endian в сравнении с Little-Endian

В каком порядке должны храниться байты, представляющие фрагмент данных, например число? Этот вопрос вызывает много споров, и мир компьютерных наук разделился на два лагеря: big-endian и little-endian.

Задумайтесь на мгновение о том, как мы представляем числа в повседневной жизни. Если вы принадлежите к культуре, использующей арабскую систему счисления, как, например, англоязычный мир, вы, вероятно, привыкли писать числа слева направо, начиная с цифры, представляющей наибольшую часть числа, и убывая от нее. Например, число 450 начинается с 4, представляющей сотни, затем идет 5, представляющая десятки, и потом 0, представляющая единицы. Цифра 4 представляет самую большую часть числа (400) и стоит первой. Решение поставить 4 первой было принято давно и в значительной степени является произвольным. Теоретически мы могли бы иметь систему счисления, в которой 450 записывается от меньшего к большему как 054, но, разумеется, мы ее не используем.

Число 450 требует 2 байт для представления в двоичной системе: 00000001 и 11000010. Если вы знаете двоичную систему, то знаете, что каждая единица представляет собой одну степень 2, которая, сложенная с другими «включенными» степенями 2, дает нам окончательное число. Первый байт, 00000001, помещает свою единственную единицу на 2^8 место и представляет число 256. Второй байт, 11000010, представляет число 194, потому что единицы для 2^1 (2), 2^6 (64) и 2^7 (128) включены, а 128 + 64 + 2 = 194. У нас есть байты для 256 и 194, так что 256 + 194 = 450.

Поскольку мы записываем 450 от большего к меньшему, можно предположить, что ваш компьютер аналогичным образом сначала сохранит байт, представляющий большую часть 450, что даст 0000000111000010, когда байты будут объединены. Этот порядок известен как *big-endian* (с прямым порядком байтов), но в настоящее время большинство компьютеров работают иначе. Типичный современный компьютер построен на микропроцессоре, который использует одну из двух архитектур: x86-64 (Intel, AMD) или ARM64 (Apple, Qualcomm и т. д.). По техническим и историческим причинам эти архитектуры хранят числа в порядке *little-endian*, где байт, представляющий наименьший сегмент числа, идет первым. В системе little-endian 2-байтовое число 450 хранится как 1100001000000001. Важно помнить, в какой системе вы работаете, поскольку интерпретация little-endian 1100001000000001 как big-endian приведет к совершенно другому значению.

Несмотря на доминирующее положение порядка little-endian в современных компьютерных архитектурах, некоторые системы, например работающие на архитектуре микропроцессора 68K (первоначально разработанной Motorola), включая оригинальные компьютеры Macintosh, хранят числа в порядке big-endian. Да, на оригинальном Macintosh число 450 хранилось «правильным образом» как 0000000111000010, но на компьютере перед вами оно, вероятно, хранится как 1100001000000001. Еще больше усложняет дело то, что большинство данных, передаваемых через интернет, отправляется в порядке big-endian. Когда вы просматриваете веб-страницы на своем микропроцессоре x86-64 или ARM64, преобразование endian происходит в фоновом режиме.

Сведение всего воедино

Для записи нашего файла MacPaint в формате MacBinary нам нужно взять массив пикселей из `dither()` и:

1) вызвать `prepare()`, чтобы преобразовать его из байтов в биты и дополнить нулями;
2) вызвать `run_length_encode()`, чтобы закодировать массив битов с помощью кодирования длины;
3) вызвать `macbinary_header()`, чтобы объединить результат с заголовком MacBinary;
4) также добавить 512-байтовый заголовок MacPaint;
5) дополнить конечный результат нулями до кратного 128 байтам значения, чтобы соответствовать требованиям спецификации MacBinary.

В основном это просто вопрос вызова функций, которые у нас уже есть:

```
# Запись массива *data* в *out_file*
def write_macpaint_file(data: array, out_file: str, width: int, height: int):
    bits_array = prepare(data, width, height)
    rle = run_length_encode(bits_array)
    data_size = len(rle) + HEADER_LENGTH # заголовок требует этого
    output = macbinary_header(out_file, data_size) + array('B', [0] * HEADER_LENGTH) + rle
    output[MACBINARY_LENGTH + 3] = 2 # Подпись заголовка развилки данных (1)
    # Формат MacBinary требует, чтобы заполнение нулями было кратно
    # 128 байтам для развилки данных.
    padding = 128 - (data_size % 128)
    if padding > 0:
        output += array('B', [0] * padding)
    with open(out_file + ".bin", "wb") as fp:
        output.tofile(fp)
```

Единственная часть этого кода, которую мы еще не обсуждали, – это 512-байтовый заголовок MacPaint. В MacPaint этот заголовок в основном использовался для хранения данных пользовательских шаблонов. Программы, которые экспортируют данные в MacPaint, обычно не содержат пользовательских шаблонов, поскольку они создаются пользователем в MacPaint. Поэтому мы можем оставить большую часть заголовка MacPaint в виде нулей. Единственное, что мы должны сделать, – это поместить небольшую подпись в заголовок MacPaint в байте 3, который всегда устанавливается на 2 ❶.

Результаты

Для запуска программы необходимо указать как входной, так и выходной файл. Например:

```
% python3 -m RetroDither -g /Users/dave/Downloads/IMG_0892.jpeg
/Users/dave/Downloads/AmericanFlag
```

Программа автоматически добавляет расширение *.bin* к выходному файлу.

На рис. 3.4 показано изображение, созданное с помощью нашей программы, и оно, на мой взгляд, отличается хорошим художественным качеством.

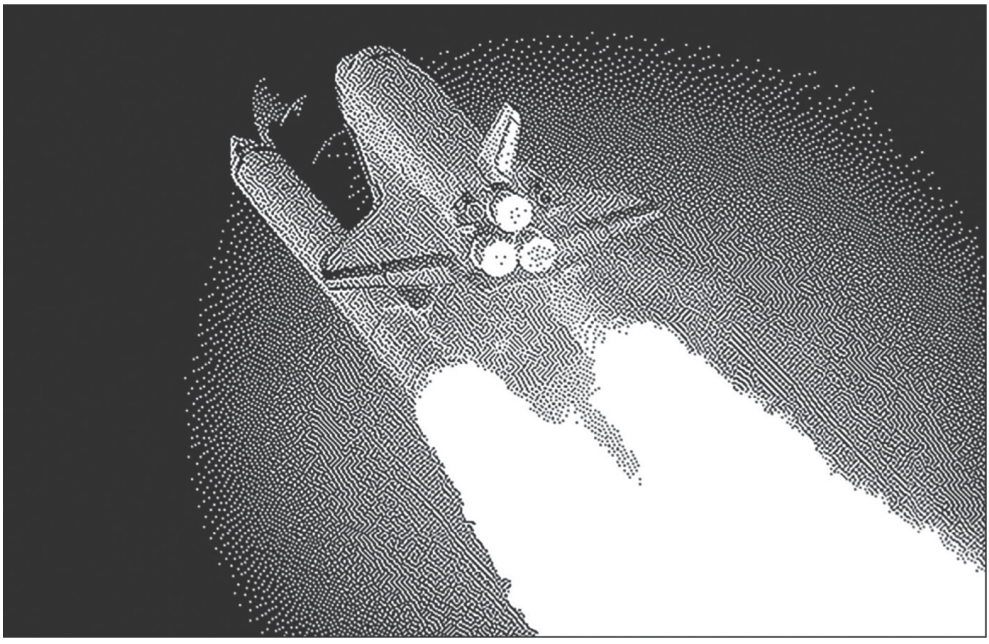


Рис. 3.4. Старт космического челнока

Но как можно посмотреть полученные результаты? Конечно же, на старинном Mac! MacPaint работает на компьютерах с System 1 вплоть до Mac OS 9. Вы должны найти настоящий классический Mac с одной из этих ОС, чтобы просматривать свои фотографии. Вам также понадобится программа для распаковки файла MacBinary. Существует программа с подходящим названием MacBinary, которую можно скачать. Очень популярное приложение для распаковки файлов для классической Mac OS, Stuffit Expander, также может открывать файлы MacBinary. Обе программы распространяются как бесплатное ПО. На большинстве Mac из 1990-х годов Stuffit Expander уже установлен.

Я понимаю, что не все будут настолько увлечены этим проектом, чтобы отправиться на поиски настоящего ретрокомпьютера. Не проблема – вы все равно можете насладиться плодами своего труда с помощью эмуляции. Много лет назад Apple начала бесплатно распространять ранние версии Mac OS, а также программу MacPaint (выпустив полный исходный код MacPaint). Вы можете получить все необходимое программное обеспечение легально и бесплатно, что позволит вам создать настоящий ретро-Macintosh в эмуляторе.

Существует несколько эмуляторов, но, пожалуй, самым «ретро» из них является Mini vMac[1], который я использовал для создания некоторых скриншотов в этой главе. Настройка Mini vMac или одного из нескольких других популярных эмуляторов классических Mac (Basilisk II, SheepShaver и т. п.) выходит за рамки этой книги, но я думаю, что вам, как программисту, это не составит большого труда. Каждый эмулятор имеет свой механизм передачи файлов с вашего компьютера в эмулируемую среду. Независимо от особенностей, это, вероятно, будет проще, чем приобрести старый Mac и подключить его к интернету, или найти дисковод для вашего современного компьютера. Тем не менее старые Mac – это весело! Или, может быть, это веселье – просто моя ностальгия по детству, проведенному с ними.

Код в реальной жизни

Я использую MacPaint с тех пор, как мой отец принес домой Macintosh LC в 1990 году. Мне было всего три года, но, по-видимому, я настолько освоил эту программу, что он привел меня на демонстрацию в класс, где он преподавал в Университете Мэна. Вероятно, отчасти это было сделано из-за новизны, а может быть, идея была в том, что «это так просто, что даже трехлетний ребенок может это сделать!». Как бы то ни было, он всегда верил в меня.

У меня до сих пор хранятся некоторые из моих детских рисунков, сохраненные на диске, и именно они недавно заинтересовали меня этим форматом файлов. Удивительно, но я не смог найти

[1] См. *https://www.gryphel.com/c/minivmac/index.html*.

на своем современном Mac ни одной программы, которая могла бы открыть классический формат MacPaint. Мне пришлось обратиться к LibreOffice или перенести файлы на старые Mac, чтобы их просмотреть.

Затем я нашел несколько загадочных файлов MacPaint, которые еще больше заинтересовали меня, на дискетке, принадлежавшей моему брату 30 лет назад. Они содержали странные рисунки, довольно сложные для конца 1980-х или начала 1990-х годов, в которых были смешаны оцифрованные (термин 1980-х годов для сканированных и подвергнутых дизерингу) объекты реального мира с чисто цифровыми рисунками. В конце концов, я пришел к выводу, что они, вероятно, были либо работой известного американского художника по имени Роберт В. Фихтер (Robert W. Fichter[1]), либо работой кого-то, кто им восхищался. В них были некоторые из тех же мотивов и точно такие же слова, как и в некоторых опубликованных работах Фихтера.

Был ли я обладателем ценной реликвии цифрового искусства, утраченной во времени? Как мой брат получил эти файлы MacPaint? Я позвонил ему. Он не думал о них десятилетиями, но сказал, что получил их от какого-то студента Университета Мэна, когда еще учился в старшей школе. Он не знал их точного происхождения, но студент сказал ему, что они очень важны и имеют скрытый смысл. Было логично, что диск попал в университет. Возможно, там читал лекции Фихтер, а может быть, файлы были скопированы с помощью «*сникернета*» (sneakernet – это термин из эпохи до интернета, означавший ручной перенос данных с машины на машину. – *Прим. перев.*), когда сети еще не были достаточно распространены. Я попытался связаться с самим Фихтером, но безрезультатно. К сожалению, он умер в 2023 году. На момент написания данной работы я все еще не знаю, являются ли эти файлы оригинальными цифровыми произведениями искусства Роберта В. Фихтера. Если вы являетесь экспертом по его творчеству или знали его, пожалуйста, свяжитесь со мной!

Примерно в то же время я думал о том, чтобы сделать проект по дизерингу для этой книги, после того как наткнулся на статью Джона Эрнеста (John Earnest) о дизеринге Аткинсона на Hacker News[2]. Я решил, что сам по себе алгоритм дизеринга слишком прост для главы книги, но потом у меня появилась идея: формат MacPaint, который недавно так меня заинтересовал, также был разработан Биллом Аткинсоном и включает в себя еще один интересный ал-

[1] См. *https://en.wikipedia.org/wiki/Robert_W._Fichter*.

[2] John Earnest, «Atkinson Dithering», Beyond Loom, 29 марта 2020 г., *https://beyondloom.com/blog/dither.html*.

горитм – кодирование по длине последовательности. Почему бы мне не объединить эти два алгоритма в одном проекте?

Я не остановился на достигнутом. После написания кода для этой главы я решил, что должен сделать MacPaint более доступным для современных пользователей Mac. Я перенес код, а также некоторые дополнительные алгоритмы дизеринга в Swift и на его основе создал удобный пользовательский интерфейс с использованием AppKit. Я продаю это программное обеспечение под названием Retro Dither в Mac App Store[1]. Обычно то, что вы делаете в качестве профессиональных или хобби-проектов, превращается в материал для книги; не так часто бывает наоборот.

Практические приложения

Дизеринг Аткинсона и MacPaint были лишь двумя из технологий, которые Билл Аткинсон использовал для оживления графики на первых компьютерах Macintosh. Он также был создателем QuickDraw, который стал основой графики на всех компьютерах Lisa и классических Macintosh. Его огромный вклад выходит далеко за рамки графики. Он внедрил несколько усовершенствований в элементы, которые мы сейчас считаем стандартными виджетами в графическом интерфейсе пользователя. Аткинсон также был создателем HyperCard, одной из первых широко распространенных гипертекстовых платформ. Можно считать ее несетевой версией интернета конца 1980-х годов. Она оказала огромное влияние.

Вам, возможно, интересно узнать, для чего Билл Аткинсон изначально разработал дизеринг Аткинсона. Мне тоже было интересно, поэтому я приобрел экземпляр редкой книги под названием «*Inside MacPaint*» Джеффри С. Янга (Jeffrey S. Young), изданной Microsoft Press в 1985 году. Я не нашел точного ответа, но нашел вероятный ответ. Оказалось, что Аткинсон работал над дигитайзером для ранних моделей Macintosh. По сути, это своего рода сканер, который позволял работать с реальными изображениями в MacPaint. Хотя в книге прямо не сказано, что в нем использовался алгоритм Аткинсона, я не могу представить, что это совпадение. Вероятно, первым реальным применением алгоритма Аткинсона был дигитайзер.

Дизеринг был широко используемой техникой в игровых консолях и компьютерах 1980-х и 1990-х годов, которые были достаточно мощными, чтобы поддерживать цифровые изображения, но зачастую имели ограниченную цветовую палитру. А устройства с ограниченной палитрой, такие как Amazon Kindle и Panic Playdate, по-прежнему являются твердыней применения дизеринга. Как упоминалось ранее, дизеринг также позволяет анимированным GIF-файлам отображать гораз-

[1] См. *https://oaksnow.com/retrodither/*.

до больше цветов, чем они поддерживают естественным образом. Без дополнительных ухищрений GIF-файл ограничен 256 цветами.

Разумеется, кодирование по длине последовательности не ограничивается MacPaint. Это широко используемый метод сжатия. Любой формат данных, в котором много повторяющихся символов, подходит для кодирования по длине последовательности. Помимо MacPaint, он использовался в качестве основного метода сжатия в нескольких других форматах растровых изображений 1980-х годов. Иногда оно также сочетается с другими методами сжатия для создания более сложных метаалгоритмов. Например, один из компонентов DEFLATE, алгоритма для файлов ZIP, использует кодирование длины последовательности.

В заключение приведу цитату из интервью Билла Аткинсона для *Inside MacPaint*. Джеффри Янг спросил его: «Что вы считаете необходимым для создания отличной программы?»

Весь секрет создания хорошей программы заключается в том, чтобы решить, что из нее убрать. Я удалил некоторые мощные функции, чтобы сделать MacPaint более чистой, простой, доступной и менее пугающей программой. Вероятно, я удалил больше кода, чем оставил. Моей целью был лаконичный, строгий и чистый дизайн. Типичный процесс программирования на 95 % состоит из отладки, а только на 5 % – из процесса реального созидания. Многие ошибки – это простые опечатки или моменты, с которыми может помочь справиться компилятор. Я предпочитаю сосредоточиться на общем алгоритме, потому что именно в этом заключается мой главный успех.

Я сравниваю программирование с лепкой из глины. Когда вы лепите горшок на гончарном круге, вы хотите, чтобы он оставался мягким и гибким как можно дольше, прежде чем приступить к его обжигу. Потому что после обжига придать горшку нужную форму будет уже гораздо сложнее[1].

Упражнения

1. Добавьте опцию командной строки, которая изменит нашу программу таким образом, чтобы она использовала дизеринг Флойда–Стейнберга.
2. Попробуйте создать свой собственный шаблон дизеринга с рассеиванием ошибок.
3. Напишите программу, которая может работать в обратном направлении, конвертируя файл MacPaint в GIF или PNG.
4. Если вы выполнили упражнение 3, напишите интеграционные тесты, которые проверяют, что файл MacPaint, конвертированный в GIF или PNG, сохраняет те же пиксельные данные, что и оригинал.

---

[1] Jeffrey S. Young, Inside MacPaint: Sailing Through the Sea of FatBits on a Single-Pixel Raft (Microsoft Press, 1985), 320.

4

СТОХАСТИЧЕСКИЙ АЛГОРИТМ ЖИВОПИСИ



Все остальные главы этой книги посвящены спецификациям, таким как грамматика языка программирования, формат файлов или архитектура компьютера. Эта глава на них совсем не похожа. В ней мы будем заниматься искусством. И этот процесс будет субъективным. Может ли простой стохастический алгоритм использовать случайно сгенерированные формы для создания рисунков, напоминающих рукотворные произведения искусства? Полагаю, что ответ – да, но вы сможете судить об этом сами, увидев результаты работы программы.

Как это работает

Представленная в этой главе программа попытается перерисовать фотографию с нуля. Мы начнем «с чистого холста». Попробуем нарисовать одну цветную фигуру произвольного размера и расположения. Если эта фигура сделает холст более похожим на фотографию, мы ее оставим. В противном случае попробуем другую фигуру. Мы повторяем этот процесс в течение определенного количества итераций. Вот и весь алгоритм.

Если это звучит просто, то это потому, что так и есть. Конечно, есть много более сложных деталей, которые нужно учесть, но они не меняют общей сути программы. Как измеряется «схожесть»? Следует ли изменять форму для лучшего соответствия? Как определяется цвет формы? Какие формы лучше использовать?

Последний вопрос приведет к появлению множества различных абстрактных вариантов наших «картин». Например, на рис. 4.1 для

приближенного представления фотографии воздушного шара использованы эллипсы. (Читатели печатной версии могут найти цветные версии изображений из этой главы в каталоге *figures* сопутствующего репозитория.)

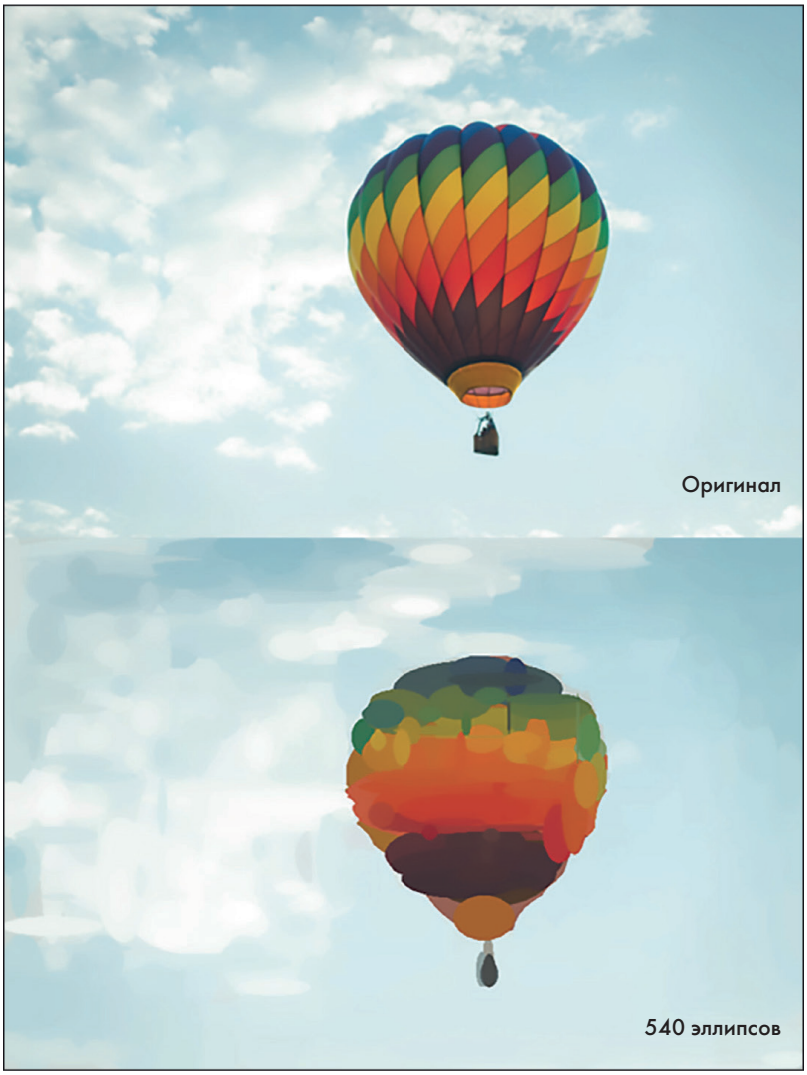


Рис. 4.1. Воздушный шар с использованием 540 эллипсов

На рисунке показаны как исходная фотография, так и результат работы нашей программы, созданный с использованием 540 эллипсов за 100 000 итераций, что заняло 104 секунды на моем ноутбуке. Как по мне, получилось неплохо. Почти импрессионизм.

На мой взгляд, эллипсы придают изображению вид витража. Это помогает, если используемая форма в некоторой степени напоминает

контуры объекта на исходной фотографии, как в случае с эллипсами для воздушного шара. На рис. 4.2 показана та же фотография, нарисованная с использованием 307 треугольников за 100 000 итераций, что заняло 109 секунд на моем ноутбуке. По моему субъективному мнению, это выглядит не очень впечатляюще.

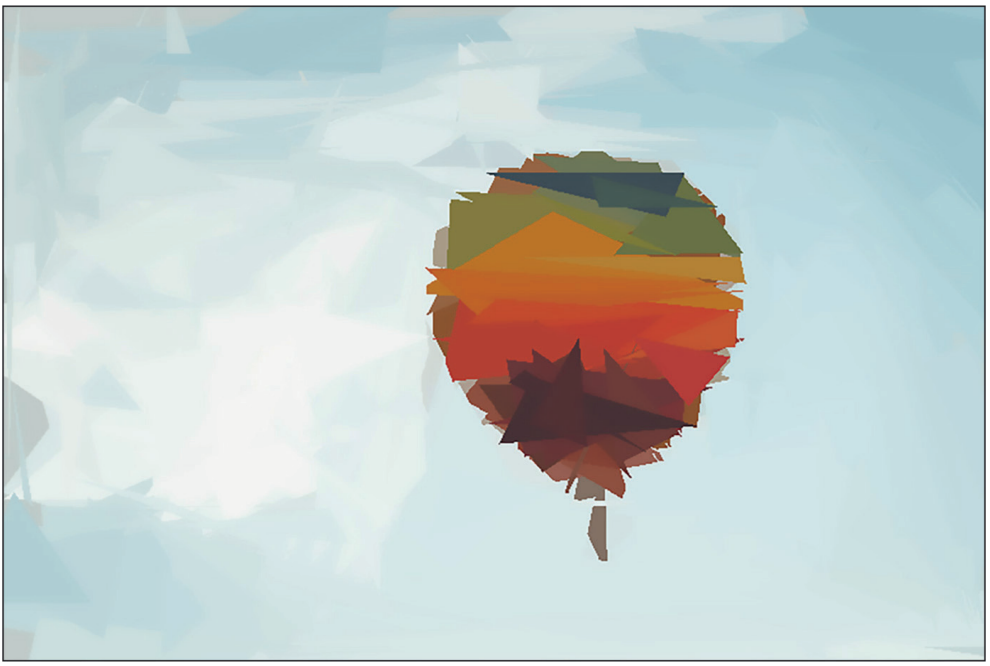


Рис. 4.2. Воздушный шар с использованием 307 треугольников

Несмотря на то что изогнутая поверхность воздушного шара не очень хорошо заполняется треугольниками, изображение бабочки получается гораздо лучше. На рис. 4.3 представлено изображение бабочки, созданное из 573 треугольников, сгенерированных за 1 млн итераций в течение 4512 секунд.

Создание изображения бабочки заняло немного больше времени, чем генерация среднего изображения с миллионом итераций, потому что я использовал метод «наиболее распространенного цвета» (most common color) вместо метода «усредненного цвета» (average color), но об этом чуть позже. В итоге все цвета в каждой фигуре стали более четкими, с меньшим эффектом смешивания.

Зачастую результат можно улучшить при использовании большего количества фигур, размещенных в ходе большего количества итера-

ций, а значит, за более длительное время. Тем не менее, поскольку этот процесс основан на стохастическом (случайном) принципе, результаты будут сильно различаться – даже при одинаковых настройках для одной и той же картинки. Примеры, которые я привожу здесь, были выбраны с особой тщательностью.

Фотографии с большим количеством деталей требуют наибольшего количества времени для создания результата с разумной степенью точности. При использовании такого инструмента необходимо найти баланс между абстракцией и степенью узнаваемости. Если изображение слишком абстрактно, его будет невозможно узнать. Но если изображение содержит так много деталей, что становится почти точной копией оригинала, оно теряет свою привлекательность в качестве «произведения искусства». Этот баланс особенно трудно достичь при работе с изображениями людей. Так, на рис. 4.4 изображены два моих друга на пляже в Санта-Монике, Калифорния. Изображение состоит из 4212 эллипсов, сгенерированных с помощью 10 млн итераций за 8262 секунды. В плане абстрактного впечатления мои друзья выглядят допустимо, зато пляж получился просто великолепным!



Рис. 4.3. Бабочка из 573 треугольников



Рис. 4.4. Пейзаж Санта-Моники из 4212 эллипсов

Мне удалось выяснить, что для создания картин с изображением людей очень хорошо подходят линии. Поскольку линии довольно тонкие, для рисования изображения требуется гораздо больше линий. На рис. 4.5 показано крошечное изображение из общественного архива, на котором Джон Ф. Кеннеди (John F. Kennedy) произносит свою знаменитую речь в Берлине, а также его живописная версия, в которой использовано 16 633 линии, полученных в результате 10 млн итераций за 6957 секунд.

Интересным аспектом примера с Кеннеди является то, что созданная программой абстрактная версия фактически имеет более высокое разрешение, чем оригинал. Это стало возможным благодаря тому, что программа работает в среде векторной графики, где результат определяется математикой, а не пикселями. Алгоритм не копирует каждый пиксель, а создает «впечатление» от оригинала с помощью векторных фигур.



Рис. 4.5. Речь Джона Ф. Кеннеди в Берлине с использованием 16 633 линий

Линии позволяют добиться наиболее впечатляющих абстрактных результатов. На рис. 4.6 представлена панорама Нью-Йорка с Манхэттенским мостом на переднем плане, где использовано 12 303 линии, сгенерированных с помощью 1 млн итераций за 909 секунд.

Программа, разработанная в этой главе, будет выполняться в течение длительного времени, если вы захотите использовать большое количество фигур. Например, для обработки изображения Кеннеди на моем ноутбуке Apple M1 потребовалось почти два часа. Время выполнения программы будет зависеть от конкретного микропроцессора вашего компьютера. Как правило, программу можно оставить работать в фоновом режиме, пока вы занимаетесь другими делами.



Рис. 4.6. Панорама Нью-Йорка из 12 303 линий

Опции командной строки

Эта программа обладает широкими возможностями настройки и множеством различных функций и регулируемых параметров. Помимо указания путей к входным и выходным файлам, наш `ArgumentParser` также должен обрабатывать все параметры командной строки, приведенные в табл. 4.1.

Таблица 4.1 Параметры командной строки для программы Impressionist (Импрессионист)

Параметр	Расширенный вид	Варианты	Значение по умолчанию	Описание
`-t`	`--trials`	Целое число	`10000`	Количество выполняемых запусков
`-m`	`--method`	`'random'`, `'average'`, `'common'`	`'average'`	Метод определения цветов фигур
`-s`	`--shape`	`'ellipse'`, `'triangle'`, `'quadrilateral'`, `'line'`	`'ellipse'`	Тип используемых фигур
`-l`	`--length`	Целое число	`256`	Длина (высота) конечного изображения в пикселях
`-v`	`--vector`	Булево	`False`	Создать векторный вывод?
`-a`	`--animate`	Целое число	`0`	Если указано число больше 0, создается анимированный GIF с заданным количеством миллисекунд на кадр, где изображение строится по одной фигуре за раз

Наш основной файл представляет собой всего лишь кодификацию табл. 4.1. Все параметры передаются в конструктор класса `Impressionist`, к которому мы вернемся чуть позже.

```
# Impressionist/__main__.py
from argparse import ArgumentParser
from Impressionist.impressionist import Impressionist, ColorMethod, ShapeType

if __name__ == "__main__":
    # Парсинг аргумента файла
    argument_parser = ArgumentParser("Impressionist")
    argument_parser.add_argument("image_file", help="The input image")
    argument_parser.add_argument("output_file", help="The resulting abstract art")
    argument_parser.add_argument('-t', '--trials', type=int, default=10000,
                                 help='The number of trials to run (default 10000).')
    argument_parser.add_argument('-m', '--method',
                                 choices=['random', 'average', 'common'], default='average',
                                 help='Shape color determination method (default average).')
    argument_parser.add_argument('-s', '--shape', choices=['ellipse', 'triangle',
                                                           'quadrilateral', 'line'],
                                 default='ellipse', help='The shape type (default ellipse).')
    argument_parser.add_argument('-l', '--length', type=int, default=256,
                                 help='The length of the final image in pixels (default 256).')
    argument_parser.add_argument('-v', '--vector', default=False, action='store_true',
                                 help='Create vector output. A SVG file will also be output.')
    argument_parser.add_argument('-a', '--animate', type=int, default=0,
                                 help='If greater than 0, will create an animated GIF '
                                     'with the number of milliseconds per frame provided.')
    arguments = argument_parser.parse_args()
    method = ColorMethod[arguments.method.upper()]
    shape_type = ShapeType[arguments.shape.upper()]
    Impressionist(arguments.image_file, arguments.output_file, arguments.trials, method,
              shape_type, arguments.length, arguments.vector, arguments.animate)
```

Опция `-v` указывает программе выводить результат в векторном формате. В нашей реализации мы будем использовать файл SVG. Прежде чем перейти к основному алгоритму приложения, давайте немного отвлечемся и посмотрим, как мы можем реализовать эту функцию.

Формат SVG

Аббревиатура SVG означает Scalable Vector Graphics (масштабируемая векторная графика). Этот формат основан на XML и используется для представления векторных изображений. Он поддерживается всеми современными популярными веб-браузерами и программами для векторного рисования. Вместо использования сторонней библиотеки для записи в SVG мы напишем свой собственный небольшой класс.

Спецификация SVG обширна, но нам нужна только небольшая ее часть для поддержки фигур, которые будет выводить наша программа, поэтому эта задача решается относительно просто.

XML – это текстовый формат, и поскольку мы его выводим, а не анализируем, нам даже не требуется, чтобы наша программа действительно понимала структуру XML. Нам просто нужно объединить строку из других строк, представляющих составляющие элементы XML. И хотя такой подход ограничивает возможности тестирования и модульность нашего SVG-редактора, объем реализуемых нами положений стандарта SVG настолько мал, что проверка на корректность вручную практически не представляет труда. Тем не менее этот подход не годится для промышленного внедрения.

Прежде чем перейти к коду, рассмотрим пример простого файла SVG, который может выводить наша программа. В нем есть только один треугольник, построенный с помощью элемента `polygon`, плюс прямоугольник фона (я немного улучшил форматирование, добавив пару отступов для удобства чтения):

```
<?xml version="1.0" encoding="utf-8"?>
<svg version="1.1" baseProfile="full" width="342" height="256"
xmlns="http://www.w3.org/2000/svg">
    <rect width="100%" height="100%" fill="rgb(108, 98, 91)" />
    <polygon points="201,3 24,9 162,182 " fill="rgb(128, 120, 112)" />
</svg>
```

Если сохранить этот код в текстовом файле с расширением *.svg*, его можно открыть в веб-браузере или редакторе векторных изображений для просмотра полученного треугольника. При создании нашего класса `SVG` помните об этом примере, чтобы представлять расположение различных элементов в полном файле SVG.

Файл SVG начинается с объявления о том, что это файл XML, а затем первым элементом идет элемент `svg`, который описывает версию спецификации SVG, а также ширину и высоту изображения. Кроме того, каждое изображение, которое генерирует наша программа, подкреплено большим прямоугольником, содержащим усредненный цвет изображения. Это помогает алгоритму добиться лучшего совмещения цветов. Соответственно, наш файл SVG также начинается с большого элемента `rect`:

```
# Impressionist/svg.py
class SVG:
    def __init__(self, width: int, height: int, background_color: tuple[int, int, int]):
        self.content = '<?xml version="1.0" encoding="utf-8"?>\n' \
                       f'<svg version="1.1" baseProfile="full" width="{width}" ' \
                       f'height="{height}" xmlns="http://www.w3.org/2000/svg">\n' \
                       f'<rect width="100%" height="100%" fill="rgb{background_color}" />'
```

Как указывает свойство `background_color` в конструкторе, цвета представлены с помощью кортежа из трех целых чисел. Это коды цветов RGB. Каждый элемент представляет собой целое число от `0` до `255`, обозначающее количественное соотношение соответствующего основного цвета (красного, зеленого или синего) в выходном изображении. Например, «чистый» красный цвет будет (`255, 0, 0`), а фиолетовый – примерно (`128, 0, 128`), поскольку он представляет собой смесь красного и синего.

Для рисования трех типов фигур, которые поддерживает наша программа (эллипсы, линии и многоугольники), достаточно просто поместить соответствующие элементы SVG `ellipse`, `line` или `polygon` в выходной текстовый файл:

```
    def draw_ellipse(self, x1: int, y1: int, x2: int, y2: int, color: tuple[int, int,
int]):
        self.content += f'<ellipse cx="{(x1 + x2) // 2}" cy="{(y1 + y2) // 2}" ' \
                        f'rx="{abs(x1 - x2) // 2}" ry="{abs(y1 - y2) // 2}" ' \
                        f'fill="rgb{color}" />\n'

    def draw_line(self, x1: int, y1: int, x2: int, y2: int, color: tuple[int, int, int]):
        self.content += f'<line x1="{x1}" y1="{y1}" x2="{x2}" y2="{y2}" stroke="rgb{color}"
' \
                        'stroke-width="1px" shape-rendering="crispEdges" />\n'

    def draw_polygon(self, coordinates: list[int], color: tuple[int, int, int]):
        points = ""
        for index in range(0, len(coordinates), 2):
            points += f"{coordinates[index]},{coordinates[index + 1]} "
        self.content += f'<polygon points="{points}" fill="rgb{color}" />\n'
```

Наконец, для вывода файла SVG мы закрываем элемент svg, начатый в конструкторе, и записываем объединенную строку на диск:

```
    def write(self, path: str):
        self.content += '</svg>\n'
        with open(path, 'w') as f:
            f.write(self.content)
```

Если вы ознакомитесь с официальной спецификацией SVG, она может показаться вам сложной, но не стоит этого пугаться. Как, надеюсь, показывает данный раздел, не обязательно тратить много времени на то, чтобы извлечь пользу из такого большого стандарта, как SVG. С помощью всего лишь 20 строк кода мы написали пусть и очень ограниченный, но весьма полезный инструмент для создания SVG.

Алгоритм

Алгоритм, который создает эти (иногда) красивые абстрактные отображения по фотографиям, удивительно прост. Если говорить кратко, он пытается рисовать фигуры случайного размера и расположения по одной фигуре за раз. Если добавленная фигура делает абстрактное изображение более похожим на исходную фотографию, она сохраняется. Улучшение может быть дополнительно доработано путем изменения размера фигуры, что сводится к перемещению каждой из ее точек. Если добавленная фигура делает изображение менее похожим на исходную фотографию, она удаляется, и начинается попытка создания новой фигуры.

Ниже приводится более подробное пошаговое объяснение алгоритма.

1. Создайте пустой холст того же размера, что и исходная фотография, с фоновым цветом, соответствующим усредненному цвету исходной фотографии.
2. Попробуйте нарисовать на холсте фигуру произвольного размера в произвольном месте. Раскрасьте фигуру, используя усредненный цвет соответствующей области исходной фотографии, наиболее часто встречающийся цвет этой области или случайный цвет.
3. Сравните цвета пикселей холста (с добавленной фигурой) с оригинальной фотографией. Если добавленная фигура сделала пиксели всего холста более похожими на пиксели оригинальной фотографии, то сохраните добавленную фигуру.
4. Попробуйте изменить форму в каждой точке (расширяя или сокращая) по одному пикселю за раз. Продолжайте перемещать точки в направлениях, которые еще больше уменьшают разницу между пикселями всего холста и пикселями исходной фотографии. Остановитесь, когда перемещение больше не улучшает эту разницу.
5. Повторите шаги 2, 3 и 4 несколько раз.
6. Выведите окончательное изображение, созданное на холсте после нескольких экспериментов.

У этого алгоритма есть множество настраиваемых параметров. Какую форму следует использовать? Сколько раз нужно повторить эксперимент? Как выбрать цвет для каждой формы? Кроме того, необходимо решить несколько дополнительных задач. Как рассчитывается разница между двумя изображениями? Как определить пиксели в пространстве, которое окружает данную форму?

Основная реализация

За вычетом комментариев, основная реализация нашего алгоритма рисования занимает менее 150 строк Python. Во многом такая лаконичность достигается благодаря мощной библиотеке Pillow, о которой уже шла речь в главе 3. В Pillow реализованы функции чтения и записи различных форматов растровых изображений. В ней также есть инструменты для рисования простых примитивов, таких как нужные нам фигуры. Наконец, Pillow обладает функциями для вычисления различий между изображениями и вычисления усредненного цвета в области изображения. Они станут важными вспомогательными функциями для нашей программы, позволяя нам сосредоточиться на основном алгоритме, а рутинную работу оставить Pillow. Именно для этого и нужна отличная библиотека.

Настройка

Начнем с основных операций импорта, определения некоторых необходимых типов, константы и вспомогательной функции:

Impressionist/impressionist.py

```
from enum import Enum
from PIL import Image, ImageDraw
from PIL import ImageChops, ImageStat
import random
from math import trunc
from timeit import default_timer as timer
from Impressionist.svg import SVG

ColorMethod = Enum("ColorMethod", "RANDOM AVERAGE COMMON")
ShapeType = Enum("ShapeType", "ELLIPSE TRIANGLE QUADRILATERAL LINE")
CoordList = list[int]
MAX_HEIGHT = 256

def get_most_common_color(image: Image.Image) -> tuple[int, int, int]:
    colors = image.getcolors(image.width * image.height)
    return max(colors, key=lambda item: item[0])[1]
```

Перечисление `ColorMethod` контролирует процесс вычисления цвета в области, то есть цвет, которым будет заполнена фигура. Перечисление `ShapeType` задает форму, которую мы будем рисовать. В текущей версии программы в каждой картинке рисуется только один тип фигуры, но код легко модифицировать для возможности рисования нескольких типов фигур. Я оставил это в качестве упражнения. Тип `CoordList` применяется к координатам, которые определяют одну фигуру.

При запуске алгоритма для обеспечения производительности нам нужно работать с ограниченным количеством пикселей. Самый простой способ добиться этого – масштабировать входное изображение, если оно больше, чем MAX_HEIGHT. Другими словами, MAX_HEIGHT – это максимальная высота масштабированного изображения. Обратите внимание, что технически мы также должны определить максимальную ширину, но на практике соотношение сторон изображения очень редко бывает таким, что ограничение только одного измерения будет недостаточным (не так много изображений, которые очень широкие, но имеют совсем небольшую высоту). Для простоты мы определили только одно максимальное измерение.

Метод `get_most_common_color()` определяет наиболее часто встречающийся цвет в изображении. Он использует метод `getcolors()` из Pillow, который возвращает все цвета в изображении вместе с их количеством. Затем он использует встроенную функцию Python `max()` для извлечения наиболее часто встречающегося цвета.

Конструктор класса `Impressionist` отвечает за настройку уникальных параметров конкретного запуска алгоритма, открытие входного файла изображения, его масштабирование, создание начального фона выходного изображения, вызов методов для запуска фактических итераций алгоритма и вызов метода для вывода окончательного файла. Это может показаться сложным, но суть алгоритма заключается в других методах. Конструктор – это всего лишь отправная точка, которая вызывает методы из библиотеки Pillow и другие методы, к которым мы скоро перейдем, для выполнения фактической работы. Вот так выглядит начало конструктора:

```
class Impressionist:
    def __init__(self, file_name: str, output_file: str, trials: int, method: ColorMethod,
                 shape_type: ShapeType, length: int, vector: bool, animation_length: int):
        self.method = method
        self.shape_type = shape_type
        self.shapes = []
        # Открыть файл изображения и сохранить в переменной экземпляра, выполнить алгоритм
        with open(file_name, "rb") as fp:
            self.original = Image.open(fp).convert('RGB')
            # Scale down image so processing is faster, 256 max height pixel dimension
            width, height = self.original.size
            aspect_ratio = width / height
            new_size = (int(MAX_HEIGHT * aspect_ratio), MAX_HEIGHT)
            self.original.thumbnail(new_size, Image.Resampling.LANCZOS)
```

Конструктор начинает с настройки некоторых параметров и масштабирования входного изображения. Результирующая картина должна иметь такое же соотношение сторон, что и исходное изображение, поэтому соотношение сторон сохраняется. Метод `thumbnail()` библиотеки Pillow представляет собой удобный способ масштабирования.

Вот следующая часть конструктора:

```
# Запустить сгенерированное изображение с фоном, который является
# усредненным значением всех пикселей оригинала по цвету
average_color = tuple((round(n) for n in ImageStat.Stat(self.original).mean))
self.glass = Image.new("RGB", new_size, average_color)
```

Модуль `ImageStat` из Pillow можно использовать для определения среднего цвета в изображении. Он анализирует значения RGB каждого пикселя в изображении и вычисляет среднее значение красного, зеленого и синего компонентов по отдельности. Мы берем полученный средний цвет и устанавливаем его в качестве фона рабочего изображения нашего алгоритма (`self.glass`). Другими словами, средний цвет исходного изображения будет начальным цветом каждого пикселя в рабочем изображении.

Примечание *Переменная для рабочего изображения называется* `glass`*, потому что изначально я назвал эту программу Stained Glass («Витраж»). После переименования я по-прежнему считаю, что название* `glass` *для переменной дает понять, что это поверхность, которая обеспечивает отфильтрованное представление оригинала.*

Продолжение конструктора:

```
# Отслеживаем, как далеко мы продвинулись, наш лучший результат на данный
# момент и сколько времени прошло с начала обработки
self.best_difference = self.difference(self.glass)
last_percent = 0
start = timer()
for test in range(trials):
    self.trial()
    percent = trunc(test / trials * 100)
    if percent > last_percent:
        last_percent = percent
        print(f"{percent}% Done, Best Difference {self.best_difference}")
end = timer()
print(f"{end-start} seconds elapsed. {len(self.shapes)} shapes created.")
self.create_output(output_file, length, vector, animation_length)
```

Сердце алгоритма находится в методе `trial()`, который пытается нарисовать фигуру и проверить, улучшает ли она показатель схожести между рабочим и исходным изображениями. Здесь `trial()` вызывается `trials` раз. По мере выполнения попыток мы отслеживаем, насколько мы близки к завершению и сколько времени занимает программа. Наконец, готовое рабочее изображение выводится с помощью `create_output()`.

Вспомогательные методы

Прежде чем перейти к `trial()`, нам понадобятся некоторые вспомогательные методы. Ключевой частью алгоритма рисования является проверка того факта, что каждая дополнительная фигура приближает рабочее изображение к исходному изображению. Метод `difference()` вычисляет оценку схожести двух изображений, измеряя степень их сходства друг с другом:

```
    def difference(self, other_image: Image.Image) -> float:
        diff = ImageChops.difference(self.original, other_image)
        stat = ImageStat.Stat(diff)
        diff_ratio = sum(stat.mean) / (len(stat.mean) * 255)
        return diff_ratio
```

Модуль `ImageChops` из Pillow имеет встроенный метод `difference()`. Он определяет разницу между двумя изображениями на уровне пикселей. Другими словами, чем отличаются друг от друга два пикселя, расположенные в одних и тех же местах на двух изображениях? Разница – это просто абсолютные значения вычитания каждого из цветовых каналов в каждом пикселе. Например, разница между пикселем RGB с цветами (10, 100, 50) и другим пикселем с цветами (10, 40, 20) будет (0, 60, 30). Однако этого недостаточно для нашего алгоритма. Нам нужно одно число в виде оценки, которое выражает степень схожести двух изображений. После нахождения разницы пиксель за пикселем мы можем сжать ее в одно число, усреднив все разницы. Мы делаем это с помощью того же модуля `ImageStat`, который усреднял для нас значения, чтобы найти усредненный цвет в конструкторе. Наконец, хотя это и не является строго необходимым (усредненные значения пикселей подойдут в качестве оценок), мы делим на максимально возможную разницу для получения оценки в виде коэффициента.

Каждый раз, когда мы генерируем новую фигуру, она размещается в случайном месте на экране. Мы вычисляем эти случайные координаты с помощью `random_coordinates()`:

```
    def random_coordinates(self) -> CoordList:
        num_coordinates = 4 # эллипс или линия
        if self.shape_type == ShapeType.TRIANGLE:
            num_coordinates = 6
        elif self.shape_type == ShapeType.QUADRILATERAL:
            num_coordinates = 8
        coordinates = []
        for d in range(num_coordinates):
            if d % 2 == 0: # координаты x
                coordinates.append(random.randint(0, self.original.width))
            else: # координаты y
                coordinates.append(random.randint(0, self.original.height))
        return coordinates
```

Различные виды фигур требуют разного количества координат. Например, треугольник имеет шесть координат, поскольку он состоит из трех точек, и каждая точка имеет одну координату *x* и одну координату *y*. Координаты должны быть действительными, то есть они должны находиться где-то на поверхности изображения. Метод обеспечивает это, гарантируя, что случайные координаты не могут быть меньше 0 или превышать ширину или высоту изображения.

Нам также нужен способ просмотра «области» исходной фотографии, соответствующей фигуре на рабочем изображении, чтобы мы могли проанализировать цвет этой области. Поиск точных пикселей под произвольной фигурой будет требовать больших вычислительных затрат. Вместо этого мы будем использовать статический метод `bounding_box()` для определения охватывающей фигуру прямоугольной области:

```
@staticmethod
def bounding_box(coordinates: CoordList) -> tuple[int, int, int, int]:
    xcoords = coordinates[::2]
    ycoords = coordinates[1::2]
    x1 = min(xcoords)
    y1 = min(ycoords)
    x2 = max(xcoords)
    y2 = max(ycoords)
    return x1, y1, x2, y2
```

Ограничивающая рамка (bounding box) – это прямоугольник, выровненный по осям (то есть его края параллельны краям изображения) вокруг заданной фигуры, определяемый на основе минимальных и максимальных координат *x* и *y* этой фигуры. Мы передадим этот прямоугольник методу `crop()`, встроенному в Pillow, чтобы обрезать исходное изображение до нужной области. Альтернативные методы извлечения более определенной области исходного изображения мы оставим для упражнений.

Пробы

Сердцем нашего алгоритма является метод `trial()`. Каждая проба – это попытка поместить одну фигуру в рабочее изображение. Если новая фигура приближает рабочее изображение к исходному, она сохраняется. Если разницу можно еще уменьшить, слегка сдвинув координаты, то координаты фигуры сдвигаются. Метод начинается с поиска места для новой фигуры с помощью `random_coordinates()` и поиска области поддержки для этих координат:

```
def trial(self):
    while True:
        coordinates = self.random_coordinates()
```

```
        region = self.original.crop(self.bounding_box(coordinates))
        if region.width > 0 and region.height > 0:
            break
```

Здесь есть некрасивый цикл `while` для учета маловероятного сценария, когда случайные координаты выровнены по одной из осей. В этом случае координаты необходимо сгенерировать заново. В конце главы есть упражнение, в котором нужно удалить этот цикл. Следующая часть метода выбирает цвет фигуры:

```
    if self.method == ColorMethod.AVERAGE:
        color = tuple((round(n) for n in ImageStat.Stat(region).mean))
    elif self.method == ColorMethod.COMMON:
        color = get_most_common_color(region)
    else: # должно быть случайным
        color = tuple(random.choices(range(256), k=3))
    original = self.glass
```

В зависимости от `ColorMethod` мы выбираем средний цвет в области фона (снова используя `ImageStat`), выбираем наиболее распространенный цвет в области фона или просто выбираем случайный цвет. Затем мы сохраняем текущее состояние рабочего изображения (`self.glass`) в локальной переменной `original`, чтобы использовать его в случае попытки сдвига координат (мы пытаемся перерисовать фигуру немного больше или немного меньше в разных направлениях, поэтому нам нужен исходный холст, на котором она была нарисована). Теперь мы готовы попробовать нарисовать фигуру:

```
    def experiment() -> bool:
        new_image = original.copy()
        glass_draw = ImageDraw.Draw(new_image)
        if self.shape_type == ShapeType.ELLIPSE:
            glass_draw.ellipse(self.bounding_box(coordinates), fill=color)
        else: # должно быть треугольником, четырехугольником или линией
            glass_draw.polygon(coordinates, fill=color)
        new_difference = self.difference(new_image)
        if new_difference < self.best_difference:
            self.best_difference = new_difference
            self.glass = new_image
            return True
        return False
```

Внутренняя функция `experiment()` возвращает `True`, если попытка нарисовать новую фигуру успешна с точки зрения уменьшения разницы между рабочим изображением и исходным изображением. Модуль `ImageDraw` в Pillow занимается фактической прорисовкой. Раз-

ница рассчитывается с помощью ранее определенного метода `difference()` и сравнивается с наилучшей разницей, найденной на данный момент. Если фигура улучшила изображение, рабочее изображение заменяется изображением, включающим новую фигуру.

В последней части функции `trial()` предпринимается попытка постепенно улучшить каждую фигуру путем сдвига ее координат. Если сдвиг улучшает показатель разницы по сравнению с версией рабочего изображения с исходными координатами фигуры, то сдвиг сохраняется и предпринимается попытка еще одного сдвига в том же направлении:

```
        if experiment():
            # Попытка расширить все направления, дальнейшее продвижение
            # в лучших направлениях
            for index in range(len(coordinates)):
                for amount in (-1, 1):
                    while True:
                        old_coordinates = coordinates.copy()
                        coordinates[index] = coordinates[index] + amount
                        if not experiment():
                            coordinates = old_coordinates
                            break
            self.shapes.append((coordinates, color))
```

Этот код представляет собой разновидность алгоритма поиска *восхождением к вершине* (hill climbing), где мы продолжаем двигаться в одном направлении для решения задачи (в данном случае оптимизации разницы) до тех пор, пока решение продолжает улучшаться. Мы останавливаемся после того, как улучшение прекращается. Это может привести к локальному максимуму, но это простой и эффективный способ улучшить существующее решение. В данном случае у нас есть существующее решение, потому что мы сохраняем только те формы, которые изначально улучшили разницу (на что указывает возвращаемое функцией `experiment()` значение `True`). См. вставку «Подъем к вершине» для получения дополнительной информации о принципе работы этого типа алгоритма.

Общий алгоритм будет работать и без процесса подгонки, но подгонка улучшает соответствие каждой формы. Это, в свою очередь, улучшает общий вид конечной картины и уменьшает количество фигур, необходимых для получения разумного результата.

После определения окончательной формы в результате подгонки мы добавляем ее координаты и цвет в список фигур. Ведение этого списка отдельно от рисования фигур на рабочем изображении необходимо для генерации конечного результата.

Восхождение к вершине

Восхождение к вершине – это простой метод оптимизации, цель которого заключается в поиске максимума или минимума функции путем непрерывного движения в одном направлении, пока поиск выглядит «успешным». В классическом объяснении этого метода вам предлагается представить, что вы стоите с завязанными глазами у подножия холма (hill), на который хотите взобраться. Вы можете почувствовать ногами уклон земли вокруг себя. С каждым шагом направление, которое кажется самым крутым подъемом, будет казаться вам тем, по которому вам следует идти, если вы хотите подняться на холм как можно быстрее. Вы можете продолжать подниматься в этом направлении до тех пор, пока ногами ощущаете, что поднимаетесь. В конце концов, вы достигнете точки, где больше не будете подниматься, независимо от направления вашего следующего шага, и тогда вы сможете остановиться.

Достигнете ли вы вершины холма? Это вполне возможно, особенно если у холма одна вершина. Но также возможно, что у холма несколько вершин, и вы достигли только одной из меньших. Это называется попаданием в *локальный максимум* (local maximum). При подъеме на холм всегда будет найден локальный максимум, но глобальный максимум может и не найтись.

Восхождение к вершине – это популярный метод в искусственном интеллекте, потому что он очень прост. Это хорошая отправная точка для решения многих задач. В нашей программе мы продолжаем сдвигать координаты в одном направлении, пока разница с исходным изображением не перестанет улучшаться. Это один из видов восхождения к вершине: мы просто продолжаем двигаться в одном направлении, пока ситуация не перестанет улучшаться. Конечно, вполне возможно, что изначально расположение фигуры было ошибочным по сравнению с некоторыми другими альтернативными вариантами, и наши сдвиги просто ведут нас в тупик к локальному максимуму. С таким простым алгоритмом нет возможности узнать это наверняка.

Вывод изображения

Рабочее изображение было масштабировано до `MAX_HEIGHT`, но окончательное изображение должно иметь заданную пользователем высоту (опять же, для упрощения мы даем пользователю возможность задавать только высоту, а не ширину). Мы не можем просто «растянуть» растровое изображение без пикселизации. Вместо этого мы перерисовываем рабочее изображение с помощью данных из списка фигур,

при этом каждая фигура масштабируется соответствующим образом. Метод вывода изображения также включает в себя опции для вывода векторного файла (с использованием класса `SVG` из предыдущего примера) и вывода анимированного GIF-файла с помощью Pillow. Это значительно увеличивает его длину. Вот начало `create_output()`:

```
def create_output(self, out_file: str, height: int, vector: bool, animation_length: int):
    average_color = tuple((round(n) for n in ImageStat.Stat(self.original).mean))
    original_width, original_height = self.original.size
    ratio = height / original_height
    output_size = (int(original_width * ratio), int(original_height * ratio))
    output_image = Image.new("RGB", output_size, average_color)
    output_draw = ImageDraw.Draw(output_image)
```

Для начала создадим новое изображение подходящего размера на основе заданного пользователем параметра высоты. Заполним начальное выходное изображение усредненным цветом исходного изображения, как это было сделано для рабочего изображения. Далее продолжим применение метода:

```
    svg = SVG(*output_size, average_color) if vector else None
    animation_frames = [] if animation_length > 0 else None
    for coordinate_list, color in self.shapes:
      ❶ coordinates = [int(x * ratio) for x in coordinate_list]
```

Выходное изображение будет сгенерировано путем итеративного воспроизведения каждой фигуры из списка фигур в нужном масштабе ❶. Чтобы также создать выходные данные в формате SVG или анимированном GIF, по мере генерации выходного изображения каждый шаг будет повторяться на объекте `svg` или копироваться в виде картинки в список `animation_frames`, который составляет анимированный «фильм» GIF:

```
        if self.shape_type == ShapeType.ELLIPSE:
            output_draw.ellipse(self.bounding_box(coordinates), fill=color)
            if svg:
                svg.draw_ellipse(*coordinates, color) # type: ignore
        else: # должен быть треугольник, или прямоугольник, или линия
            output_draw.polygon(coordinates, fill=color)
            if svg:
                if self.shape_type == ShapeType.LINE:
                    svg.draw_line(*coordinates, color) # type: ignore
                else:
                    svg.draw_polygon(coordinates, color)
        if animation_frames is not None:
            animation_frames.append(output_image.copy())
    output_image.save(out_file)
```

```
if svg:
    svg.write(out_file + ".svg")
if animation_frames is not None:
    animation_frames[0].save(out_file + ".gif", save_all=True,
                             append_images=animation_frames[1:],
                             optimize=False,
                             duration=animation_length, loop=0,
                             transparency=0, disposal=2)
```

Остальная часть метода заключается в простом рисовании фигур на выходном изображении (изображениях) и записи файла (файлов) на диск.

Результаты

В этих 150 строках кода содержится очень много информации. Программа включает в себя стохастические пробы, несколько приемов, позволяющих сделать обоснованное предположение о цвете каждой фигуры, небольшой алгоритм восхождения к вершине и работу с хорошей библиотекой. В компьютерных науках интересные результаты в гораздо большей степени зависят от алгоритма и техники, нежели от количества строк кода. Но этот алгоритм на удивление прост и при этом чрезвычайно эффективен. Нет, результат не так впечатляет, как новейшая нейронная сеть, но просто поразительно, насколько глубоко может погрузить программу простой метод.

Главный недостаток этого алгоритма заключается в его медлительности и произвольности. Вы можете попробовать одно и то же изображение несколько раз с одинаковыми параметрами и получить разные результаты. И вам, возможно, придется долго ожидать этих самых разных и порой не очень удачных результатов.

Тем не менее я хочу поделиться с вами некоторыми впечатляющими, хотя и, признаюсь, тщательно отобранными результатами. Первые из них – это несколько сцен из парка Туро в Ньюпорте, штат Род-Айленд. Мне нравится, как форма линий придает каждой из них вид, напоминающий живопись маслом. На рис. 4.7 представлен общий вид парка со знаменитой башней Ньюпорта с правой стороны.

На рис. 4.8 показан более крупный вид башни Ньюпорта.

На рис. 4.9 представлена кошка, которую я нашел катающейся по тротуару. Эллиптическая форма придает кошке приятный абстрактный вид. Могла бы это быть работа импрессиониста?



Рис. 4.7. Парк Туро содержит 19 578 линий

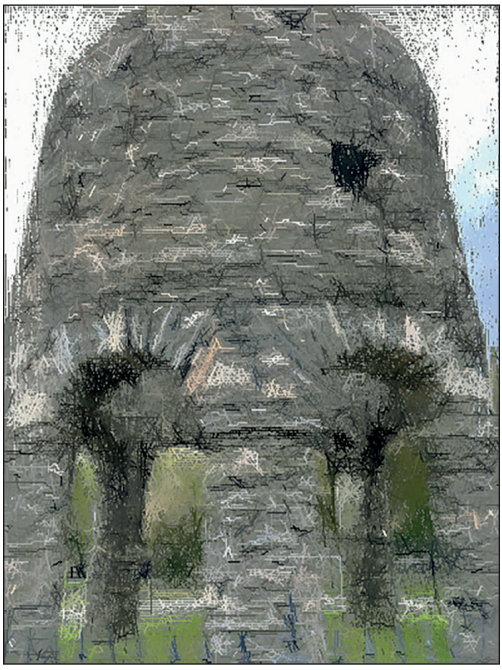


Рис. 4.8. Башня Ньюпорта содержит 11 409 линий

Наконец, на рис. 4.10 представлена сцена из Хэллоуина.



Рис. 4.9. Кошка на тротуаре, нарисованная с помощью эллипсов

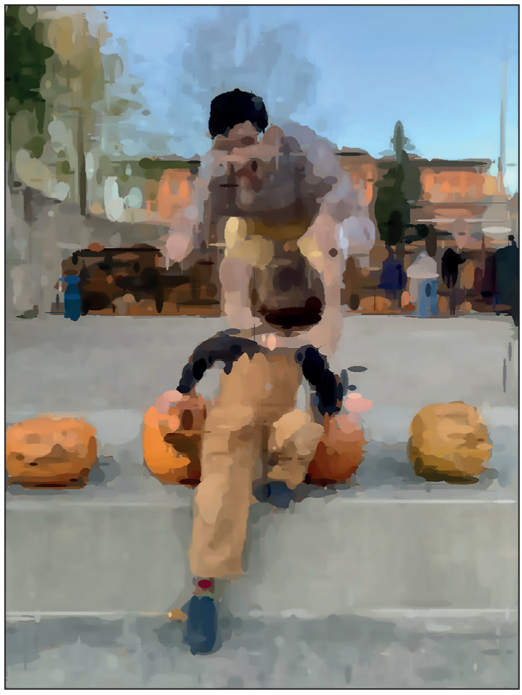


Рис. 4.10. Сцена из Хэллоуина, нарисованная с помощью эллипсов

Мы с сыном проходили мимо публичной выставки тыкв. Мне нравится, как эллипсы изобразили тыквы и людей на заднем плане.

Код в реальной жизни

В середине 2010-х годов я впервые задумался о создании программы, подобной приведенной в этой главе, с использованием генетического алгоритма (genetic algorithm). Я провел небольшое исследование и обнаружил, что несколько человек уже опередили меня. Однако в ходе этих исследований я также наткнулся на проект Primitive Майкла Фоглемана (Michael Fogleman)[1]. Он создал программу, которая генерировала абстрактное искусство, как и более старые программы с использованием генетических алгоритмов, но в его случае использовалась более простая техника, называемая имитацией отжига (simulated annealing).

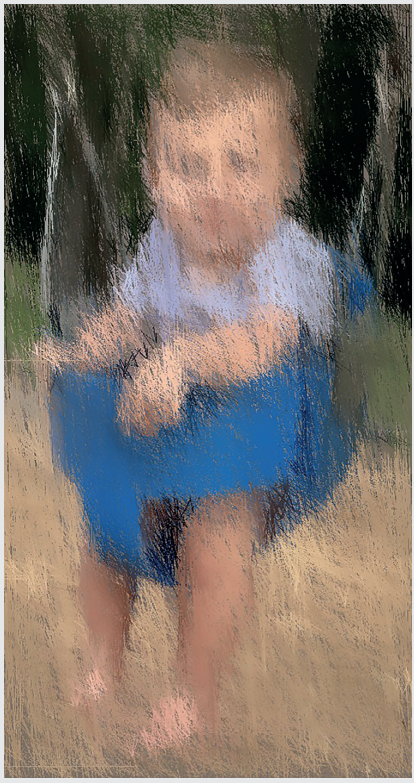


Рис. 4.11. Проект Primitive Майкла Фоглемана

[1] Michael Fogleman, «Primitive», доступ 9 января 2023 г., *https://www.michael-fogleman.com/#primitive*.

Я хочу поблагодарить Майкла за то, что он оказал большое влияние на мою карьеру программиста. Майкл – очень талантливый программист, но помимо этого он также умеет создавать невероятно читабельный код. И он создает проекты во многих интересующих меня областях. Вы узнаете больше о другом проекте Майкла в разделе этой книги, посвященном эмуляции.

Хотя этот проект не имеет ничего общего с кодом Primitive Майкла, тот факт, что он смог превратить фотографии в абстрактное искусство с помощью такого простого алгоритма, натолкнул меня на мысль о том, что я могу сделать то же самое с помощью своей собственной, еще более простой техники. Я приступил к реализации своего алгоритма в виде приложения для iOS. В целом я добился успеха, но, к сожалению, iPhone 2017 года выпуска не были достаточно быстрыми, чтобы исполнять мою программу за разумное время. Я попытался оптимизировать ее, но проблема была в моем алгоритме, а не в реализации. В то же время для iOS начали появляться интересные приложения на основе машинного обучения для творческого преобразования фотографий, и я понял, что моя медлительная и упрощенная техника просто не выдерживает конкуренции. Однако она все равно была хорошей демонстрацией, и я вспомнил о ней, когда придумывал проекты для этой книги. Полагаю, это отличная иллюстрация возможностей случайных алгоритмов и метода восхождения к вершине.

После переноса своего кода Swift на Python для этой книги я решил протестировать его с помощью своих друзей, опубликовав на Facebook фотографию своего годовалого сына Дэниела на качелях.

Моя тетя, которая имеет довольно тренированный художественный взгляд, подумала, что я занялся живописью. Тогда я понял, что программа получилась довольно удачной.

Практические приложения

Помимо довольно эффектного внешнего вида, практических применений у результатов этой программы не так много. Однако методы, которые были использованы для ее создания, безусловно, имеют практическое значение. Они относятся к общему определению, известному как *стохастическая оптимизация* (stochastic optimization). Предположим, вам нужно решить задачу оптимизации, но вы не знаете детерминированного алгоритма (алгоритма, который каждый раз дает одинаковый результат, следуя одним и тем же шагам). В этом случае может оказаться целесообразным использование метода на основе случайных (стохастических) экспериментов.

Возможно, это не совсем очевидно, но рассматриваемая в этой главе проблема является примером задачи оптимизации. Наша программа пытается оптимизировать рисунок, чтобы он был максимально близок к исходной фотографии. Целевой функцией (то есть тем, что проверяет, движемся ли мы в правильном направлении) является метод `difference()`. Чем меньше разница, тем более оптимальным является потенциальное решение задачи, которую представляет конкретное изображение.

Одной из известных практических областей, в которой стохастические алгоритмы оптимизации могут оказаться полезными, является классическая задача коммивояжера (traveling salesperson problem). Задача заключается в том, чтобы путешественник посетил каждое указанное место на карте ровно один раз и вернулся в исходную точку по кратчайшему маршруту. Именно этим ежедневно занимаются службы доставки (например, FedEx или UPS), поэтому задача действительно имеет практическое применение. К сожалению, не существует известного детерминированного алгоритма с оптимальным решением задачи коммивояжера для большого количества мест за разумное время. Однако стохастические методы оптимизации, такие как генетические алгоритмы, являются эффективным способом решения этой задачи, хотя полученные решения могут быть неоптимальными. Генетический алгоритм не всегда дает идеальное решение задачи коммивояжера, но почти всегда дает достаточно хорошее решение.

В нашей программе также применяется метод восхождения к вершине. Хотя это одна из самых простых процедур локального поиска (просто продолжайте двигаться в том же направлении, если это направление работает), она является очень распространенной техникой и во многих сценариях работает так же хорошо, как и более сложные алгоритмы. Метод восхождения к вершине лежит и в основе других, более совершенных алгоритмов. Например, в симплексном алгоритме для решения задач линейного программирования также используется метод восхождения к вершине[1].

Упражнения

1. Измените программу таким образом, чтобы она рисовала более одного типа фигур в одном изображении. Например, она может создавать рисунок, где в конечном итоге будут присутствовать как эллипсы, так и треугольники.
2. Поскольку пиксели, используемые для расчета цвета новой фигуры, основаны на ограничивающей рамке, а не на точной площа-

[1] Steven S. Skiena, «Combinatorial Search and Heuristic Methods», в Algorithm Design Manual, 2-е изд. (Springer, 2008), 252–253.

ди под фигурой, результаты получаются неточными. Измените функцию `trial()` таким образом, чтобы поэкспериментировать не только со сдвигом координат, но и со сдвигом цвета. Это может привести к более точному совпадению цветов.

3. Измените функцию `trial()` таким образом, чтобы использовать точные пиксели под фигурой для определения цвета этой фигуры. Это сложная задача. Один из способов ее решения – использовать некоторые геометрические вычисления и определить правильные пиксели для каждого типа фигуры. Вероятно, это будет гораздо менее эффективно с точки зрения вычислений, чем просто обрезать ограничительную рамку, как это делает исходная программа. Вместо этого рассмотрите возможность применения средств маскирования в Pillow.
4. Цикл `while True` в начале функции `trial()` выглядит как код с ошибкой. Перепишите начало функции `trial()` без него.

ЧАСТЬ III

ЭМУЛЯТОРЫ

5

СОЗДАНИЕ ВИРТУАЛЬНОЙ МАШИНЫ CHIP-8



В этой главе мы разработаем версию виртуальной машины, известной как CHIP-8 – платформы из ранних дней персональных компьютеров, которая в основном использовалась для игр. Хотя наша программа сможет запускать игры CHIP-8, нас интересуют не собственно игры, а понимание того, как создание виртуальной машины CHIP-8 может помочь нам в изучении низкоуровневого программирования и принципов работы компьютера на уровне регистров и инструкций. Эти знания и превращают создание виртуальной машины CHIP-8 в важный первый шаг на пути в мир программирования эмуляторов.

Виртуальные машины

Считайте *виртуальную машину* (Virtual Machine, VM) компьютером, который полностью определяется программным обеспечением. Программы для запуска в VM могут работать на любой платформе, где реализована эта VM. Таким образом, VM позволяют создавать подлинно портируемое программное обеспечение.

Виртуальные машины имеют тесную связь с эмуляторами. *Эмулятор* – это программное обеспечение, которое имитирует аппаратное обеспечение. Это позволяет программам, написанным для такой аппаратной платформы, работать на других машинах в отсутствие необходимого аппаратного обеспечения. Эмулятор должен точно следовать спецификации оригинального оборудования, для того чтобы

воссоздать все функции, которые ожидают неизвестные программы, запущенные на эмуляторе. Говоря «*неизвестные*», я имею в виду, что программное обеспечение, запущенное на эмуляторе, не знает, что оно не работает на реальном оборудовании; для корректной работы программы эмулятор должен работать точно так же, как оригинальное оборудование.

Виртуальная машина также представляет собой программное обеспечение, строго следующее спецификациям среды, в которой работает программа. Разница заключается в том, что эмулятор следует спецификациям аппаратного обеспечения, а виртуальная машина следует спецификациям, которые могут быть полностью определены как абстракция программного обеспечения.

И хотя одно является аппаратной спецификацией, а другое – программной, реализация простого эмулятора очень похожа на реализацию простой VM. На самом деле они настолько похожи, что хотя выполненный в этой главе проект технически является именно VM-проектом, его обычно выбирают в качестве первого проекта по эмуляции. Если вы новичок в сообществе разработчиков эмуляторов и задаетесь вопросом, с чего начать, то почти всегда ответом будет CHIP-8.

Пожалуй, наиболее известной виртуальной машиной является виртуальная машина Java (JVM). Когда Java впервые появилась в середине 1990-х годов, была провозглашена ее философия «написал один раз, запускай везде». Было разработано несколько JVM для всех основных операционных систем (Windows, Linux, Mac OS и т. д.), и одна и та же программа на Java могла быть скомпилирована в нативный байт-код JVM и запущена на любом компьютере с JVM без изменений, независимо от базовой платформы. Это верно и сегодня, но первоначальная позиция Java «написал один раз – запускай везде» в значительной степени была вытеснена веб-приложениями.

Технология VM CHIP-8 появилась гораздо раньше. В 1970-х годах Джозеф Вайсбекер (Joseph Weisbecker) выступил пионером в области разработки одного из первых 8-битных микропроцессоров RCA 1802. Он и компания RCA создали один из первых персональных компьютеров на основе его изобретения[1]. Он искал способ программирования игр для этой машины на языке более высокого уровня, нежели машинный код, поэтому разработал CHIP-8 (и сопутствующий ему язык операционных кодов). Его дочь, Джойс Вайсбекер (Joyce Weisbecker), впоследствии стала первой женщиной-разработчиком видеоигр с помощью CHIP-8[2]. В 1980-х годах CHIP-8 был портирован на

---

[1] Joe Weisbecker, «A Practical, Low-Cost, Home/School Microprocessor System», Computer 7, № 08 (август 1974): 20–31.

[2] Katianne Williams, «Joyce Weisbecker: The First Indie Game Developer», IEEE Women in Engineering Magazine 16, № 2 (декабрь 2022): 15–20, doi:10.1109/MWIE.2022.3203181.

многие другие платформы, включая графические калькуляторы. Так он стал по-настоящему портируемой виртуальной машиной, аналогом ранней формы того подхода к виртуальным машинам, который мы применяем сегодня.

Виртуальная машина CHIP-8

Виртуальная машина CHIP-8 изначально была разработана для персональных компьютеров конца 1970-х годов с чрезвычайно ограниченными ресурсами, таких как COSMAC VIP. Выпущенный в 1977 году, COSMAC VIP имел 8-битный микропроцессор RCA 1802 с тактовой частотой менее 2 мегагерц (МГц), 2 Кбайта оперативной памяти (с возможностью расширения до 4 Кбайт) и 512 байт постоянной памяти. Он оснащался также специализированными микросхемами для отображения 1-битной графики с разрешением до 64×128, чтения и записи кассет, а также воспроизведения звуковых сигналов[1].

По сегодняшним меркам просто удивительно, что на такой машине, как COSMAC VIP, можно было программировать что-то стоящее, но она была разработана для видеоигр. На самом деле эти игры даже работали через другой уровень абстракции, CHIP-8 VM. Самая популярная игровая консоль того времени, Atari 2600, также была выпущена в 1977 году и имела аналогичные характеристики. Эти ограничения были совершенно обычными для того времени.

При программировании VM или эмулятора в первую очередь важна производительность используемых инструментов. Виртуальная машина или эмулятор добавляют еще один уровень абстракции между программой и аппаратным обеспечением, а каждый уровень абстракции, как правило, сопровождается некоторыми потерями производительности. Чтобы достичь желаемого уровня скорости исходной системы, необходимо свести к минимуму избыточные издержки, а некоторые языки программирования (или, точнее, основные реализации среды выполнения некоторых языков программирования) этому мешают. Поэтому виртуальные машины и эмуляторы обычно программируются на низкоуровневых языках, таких как C, C++ и Rust. Тем не менее, учитывая ограниченность исходного целевого оборудования CHIP-8, сегодня несложно создать высокопроизводительную виртуальную машину CHIP-8 на любой современной системе. Достаточно даже относительно медленной среды выполнения языка программирования, такой как CPython. Вы бы не стали программировать эмулятор современной игровой консоли на Python или JVM. Но CHIP-8? Для этого более чем подходит Python.

Чтобы лучше понимать CHIP-8, давайте сначала обсудим его регистры и структуру памяти. Затем я представлю общий обзор инст-

[1] RCA COSMAC VIP CDP18S711 Instruction Manual (RCA Corporation, 1978).

рукций, которые может выполнять виртуальная машина, после чего перейду к подробному описанию реализации.

Регистры и память

В любом микропроцессоре *регистры* являются самым быстрым видом памяти. Они находятся непосредственно в микропроцессоре и не предполагают задержки при доступе к другому чипу. Запись данных в регистры зачастую является единственным способом работы с ними, поскольку большинство поддерживаемых микропроцессором инструкций по обработке данных (например, арифметических) работают с данными именно в регистрах. Отдельные инструкции по загрузке/сохранению передают данные между регистрами и внешней оперативной памятью.

Когда речь заходит о регистрах, возникает классический компромисс между временем и пространством: регистры обеспечивают самое быстрое сохранение данных, но их размер чрезвычайно ограничен. Например, типичный 8-битный микропроцессор конца 1970-х годов мог иметь только несколько 8-битных регистров (да, каждый из них может хранить лишь один байт), но он мог адресовать десятки килобайтов внешней оперативной памяти.

Большинство VM, таких как CHIP-8, также имеют регистры, но эти регистры не всегда напрямую сопоставляются с физическими аппаратными регистрами микропроцессора. Поэтому они не обязательно быстрее, чем оперативная память. Это может показаться странным, но регистры обеспечивают основу, на которой могут работать инструкции. Ничто не мешает конкретной реализации VM сопоставлять виртуальные регистры с реальными аппаратными регистрами для повышения производительности – при условии что количество виртуальных регистров не превышает количество физических регистров.

Примечание *В дальнейшем описании для обозначения регистров CHIP-8 будут использоваться те же названия, что и в коде Python для этой реализации.*

Виртуальная машина CHIP-8 имеет 16 универсальных 8-битных регистров, обозначаемых от `v[0]` до `v[15]`. Они могут использоваться для любых типов данных, и все основные арифметические и логические инструкции работают с этими регистрами. Из этих регистров общего назначения `v[15]` (или `v[0xF]` в шестнадцатеричном формате) является особым, поскольку используется для хранения флага. Регистр индекса `i` предназначен для одновременной работы с несколькими ячейками памяти и для указания места в памяти для данных, которые необходимо вывести на экран. Счетчик программы `pc` – это специальный регистр, который отслеживает в памяти адрес следующей инструкции для выполнения.

Основные регистры – это `vs`, `i` и `pc`, и они сопровождаются парой псевдорегистров для синхронизации. Эти два байта, `delay_timer` и `sound_timer`, используются для реализации паузы в игре или для указания длительности звукового сигнала. Существуют специальные команды для изменения этих таймеров. Все регистры перечислены в табл. 5.1. Регистры были первоначально описаны в руководстве по эксплуатации RCA COSMAC VIP CDP18S711[1].

Таблица 5.1 Регистры и псевдорегистры CHIP-8

Регистр	Название	Описание
`v[0]–v[14]`	Регистры общего назначения	Каждый из них может хранить любые 8-битные данные
`v[15]`	Регистр флага	Хранит флаг (1 или 0) после определенных операций, например флаг переноса после сложения
`pc`	Счетчик программы	Отслеживает в памяти 16-битный адрес текущей выполняемой инструкции
`i`	Регистр индекса памяти	Хранит 16-битный адрес, используемый для выполнения инструкций, которые занимают несколько смежных ячеек в памяти
`delay_ timer`	Таймер задержки	Хранит 8-битное значение, которое уменьшается 60 раз в секунду, пока не достигнет 0
`sound_ timer`	Звуковой таймер	Хранит 8-битное значение, которое уменьшается 60 раз в секунду, пока не достигнет 0; пока оно выше 0, компьютерный динамик воспроизводит звуковой сигнал

Типичная VM CHIP-8 располагает 4 Кбайтами оперативной памяти общего назначения. Это соответствует COSMAC VIP при загрузке с расширенной памятью. Однако есть одна загвоздка: на VIP первые 512 байт памяти должны были содержать код для самой виртуальной машины CHIP-8 (да, вся виртуальная машина помещалась всего в 512 байтах машинного кода – помните об этом, когда будем писать нашу версию). В результате оставалось только 3,5 Кбайта доступной оперативной памяти. Чтобы обеспечить обратную совместимость, наша виртуальная машина также должна резервировать первые 512 байт оперативной памяти.

Инструкции

В основном VM CHIP-8 использовалась для программирования игр, поэтому она включает в себя специализированные инструкции для таких действий, как перемещение спрайтов и воспроизведение звукового сигнала. Они соседствуют со всеми обычными, прикладными инструкциями, которые можно найти в любом наборе инструкций микропроцессора или низкоуровневом языке программирования, – инструкциями для работы с памятью, выполнения арифметических операций, контроля потока управления, обслуживания таймеров и управления отображением. Всего мы реализуем 35 инструкций. Все

[1] RCA COSMAC VIP CDP18S711 Instruction Manual (RCA Corporation, 1978).

инструкции указаны в шестнадцатеричном (hexadecimal) формате – подробнее об этой системе счисления см. в разделе «Шестнадцатеричный формат».

Шестнадцатеричный формат

Шестнадцатеричная система счисления, или система по *основанию 16*, обычно используется для работы с низкоуровневыми байтами в вычислительных системах (адреса ОЗУ, инструкции процессора и т. п.). Она обеспечивает более компактное и последовательное представление значений в байтах, чем двоичная или стандартная десятичная система счисления (с основанием 10, к которой мы привыкли). Например, любое 8-битное число можно представить с помощью двух шестнадцатеричных цифр, и, что очень удобно, каждая из этих двух цифр соответствует ровно половине байта при записи в двоичном формате (половина байта, или полубайт, также называется *ниббл* – nibble). Если бы вы были программистом в 1970-х или 1980-х годах, вы бы часто работали с шестнадцатеричной системой, но сегодня среднестатистический разработчик Python редко использует ее за пределами низкоуровневого программирования.

В шестнадцатеричной системе, помимо 10 символов 0–9, используются еще шесть символов A–F, соответствующих десятичным значениям 10–15. В Python шестнадцатеричные литералы начинаются с префикса `0x`. Например, `0xFF` соответствует десятичному числу `255`, или двоичному числу `0b11111111`. Один символ F в шестнадцатеричной версии относится к первой половине единиц в двоичной версии (1111), а другой символ F относится ко второй группе единиц (1111). Это максимальное значение 1 байта. Чтобы более наглядно проиллюстрировать преобразование, шестнадцатеричное число `0xF0` можно записать в двоичном виде как `0b11110000`, где F обозначает 1111, а 0 обозначает 0000.

Для преобразования из шестнадцатеричной системы в десятичную умножьте каждую шестнадцатеричную цифру справа налево на степень 16, начиная с числа 16^0. Например, `0xFF` можно переписать как $(15 \times 16^0) + (15 \times 16^1)$. Правая цифра (F) становится $15 \times 1 = 15$, левая цифра становится $15 \times 16 = 240$, и $240 + 15 = 255$. Вот еще один пример: `0xA5B` – это $(11 \times 16^0) + (5 \times 16^1) + (10 \times 16^2)$. Это эквивалентно 2651 в десятичной системе.

Инструкции приведены здесь в качестве краткого справочника и для того, чтобы дать вам представление о «положении дел». Мы подробно рассмотрим принцип работы каждой инструкции в коде, но на самом деле большая часть кода довольно понятна по описаниям

инструкций. Подавляющее большинство инструкций можно реализовать всего в паре строк Python.

Я долго думал над тем, как сгруппировать инструкции для этого раздела. В конце концов, я решил расположить их в порядке нумерации, чтобы они отображались здесь в том же порядке, что и в коде. Каждая инструкция в CHIP-8 имеет размер 16 бит, или, другими словами, 2 байта, или 4 ниббла, что соответствует четырем шестнадцатеричным цифрам. Любая заглавная шестнадцатеричная цифра от 0 до F в инструкции является литералом. Любая строчная буква представляет значение, которое будет использоваться в рамках реализации инструкции. Подчеркивание (_) означает, что ниббл является произвольным. Инструкции были первоначально описаны в руководстве по эксплуатации RCA COSMAC VIP CDP18S711[1].

Примечание *Некоторые из перечисленных здесь инструкций не были представлены в исходной спецификации CHIP-8 (например, 8x_6 и 8x_E). Их функциональность может отличаться в различных реализациях CHIP-8.*

Очистка экрана и базовые переходы

Первый набор инструкций используется для очистки всего экрана за один раз и для перехода из одной части программы в другую.

- **`00E0`** Очистка экрана.
- **`00EE`** Возвращение из подпрограммы.
- **`0nnn`** Вызов программы по адресу `nnn`, сброс таймеров и регистров, очистка экрана.
- **`1nnn`** Переход по адресу `nnn` без сброса.
- **`2nnn`** Вызов подпрограммы по адресу `nnn`.

Условные переходы

Следующий набор инструкций предназначен для перехода к другой части программы при выполнении определенного условия.

- **`3xnn`** Пропуск следующей инструкции, если `v[x]` равно `nn`.
- **`4xnn`** Пропуск следующей инструкции, если `v[x]` не равно `nn`.
- **`5xy_`** Пропуск следующей инструкции, если `v[x]` равно `v[y]`.

Настройка регистров общего назначения, арифметические операции и работа с битами

Далее идут стандартные инструкции, которые используются в любом процессоре или виртуальной машине для выполнения таких действий, как математические вычисления, настройка регистров и сдвиг битов.

---

[1] RCA COSMAC VIP CDP18S711 Instruction Manual (RCA Corporation, 1978).

6xnn Установка v[x] в nn.
7xnn Добавление nn к v[x].
8xy0 Установка v[x] в v[y].
8xy1 Установка v[x] в v[x] | v[y] (битовое OR).
8xy2 Установка v[x] равным v[x] & v[y] (битовое AND).
8xy3 Установка v[x] равным v[x] ^ v[y] (битовое XOR).
8xy4 Добавление v[y] к v[x] и установка флага переноса.
8xy5 Вычитание v[y] из v[x] и установка флага заимствования.
8x_6 Сдвиг v[x] вправо на один бит и установка флага в младший бит.
8xy7 Вычитание v[x] из v[y] и сохранение результата в v[x]; установка флага заимствования.
8x_E Сдвиг v[x] влево на один бит и установка флага в старший бит.

Прочие инструкции

Эти инструкции не имеют единой тематики, но их коды операций близки друг к другу по номеру.

9xy0 Пропуск следующей инструкции, если v[x] не равно v[y].
Annn Установка i в nnn.
Bnnn Переход к nnn + v[0].
Cxnn Установка v[x] в случайное целое число (0–255) & nn (битовое AND).
Dxyn Рисование спрайта высотой n в точке (v[x], v[y]); установка флага при возникновении коллизии.

Инструкции для клавиш и таймера

Следующая группа инструкций предназначена для управления таймерами виртуальной машины, проверки состояния различных клавиш или ожидания нажатия определенной клавиши.

Ex9E Пропуск следующей инструкции, если клавиша v[x] установлена (нажата).
ExA1 Пропуск следующей инструкции, если клавиша v[x] не установлена (не нажата).
Fx07 Установка v[x] для таймера задержки.
Fx0A Ожидание следующего нажатия клавиши, затем сохранение клавиши в v[x].
Fx15 Установка таймера задержки для v[x].
Fx18 Установка таймера звука для v[x].

Инструкции регистра i

Все инструкции в этом последнем наборе связаны с регистром индекса памяти (i).

Fx1E Добавление v[x] к i.
Fx29 Установка i в положение символа v[x] в наборе шрифтов.

Fx33 Сохранение значения двоично-десятичного числа (binary-coded decimal, BCD) в `v[x]` в ячейках памяти `i`, `i + 1` и `i + 2` (подробнее об этом см. ниже в тексте «Двоично-десятичные числа»).

Fx55 Дамп регистров `v[0]` через `v[x]` в память, начиная с `i`.

Fx65 Сохранение памяти из `i` через `i + x` в регистры `v[0]` через `v[x]`.

Задумайтесь на мгновение, насколько прозаично звучат эти инструкции. На самом деле для работы «компьютера» не нужны никакие сложные механизмы. Сравните описанные в этой главе 35 инструкций CHIP-8 с 8 инструкциями в нашей реализации Brainfuck из главы 1. Обе системы являются «машинами Тьюринга» с ограниченным объемом памяти, и они не так сильно отличаются друг от друга, как может показаться на первый взгляд, если судить по синтаксису инструкций.

Двоично-десятичные числа

Двоично-десятичные числа (BCD) – это способ хранения десятичных чисел в двоичном формате. Сегодня он используется не так часто, но в ранних компьютерах был очень популярен. Например, несколько микропроцессоров 1970-х годов включали явные инструкции для арифметики BCD, которая обеспечивала большую точность при округлении десятичных чисел и в некоторой степени делала машинный код более читабельным. Для среднестатистического современного программиста изучение BCD не представляет большого интереса, разве что из любопытства. Существовало несколько различных схем BCD, и, честно говоря, я не думаю, что изучение конкретной схемы, используемой в виртуальной машине CHIP-8, является ценным использованием места в этой книге.

Реализация

Теперь, когда мы познакомились с архитектурой CHIP-8, можно приступать к реализации нашей VM. Файл *__main__.py* будет содержать основной цикл выполнения, который обрабатывает ввод пользователя, обновляет отображение, управляет таймерами и, что наиболее важно, дает команду виртуальной машине перейти к следующей инструкции. В этом файле также анализируется аргумент командной строки, который указывает файл ПЗУ. Между тем *vm.py* – это и есть сама виртуальная машина.

ПЗУ

Вы когда-нибудь задумывались, почему файлы с играми, используемые в эмуляторах, называются ROM? Аббревиатура *ROM* означает *ПЗУ*, то есть «память только для чтения» (read-only memory). В большинстве ранних игровых систем использовались пластиковые картриджи, которые были не чем иным, как держателями для ПЗУ-чипов, и которые подключались непосредственно к консолям. Когда игры конвертировались в файлы для эмуляторов, кто-то должен был вставить ПЗУ-чип в специальное устройство, подключенное к компьютеру, и «скопировать» данные с ПЗУ-чипа, чтобы сохранить их в файле. Файл содержал точную копию данных с ПЗУ-чипа и, возможно, некоторую дополнительную информацию в заголовке, в зависимости от экосистемы эмуляции.

В то время как данные на оригинальных ПЗУ-чипах не могли быть изменены, эти «файлы ПЗУ» ничем не отличаются от других файлов и могут быть изменены для внесения поправок в игры. Отсюда и возникла субкультура *взлома ПЗУ*, в которой разработчики изменяют графику или игровой процесс игр, предназначенных для запуска в эмуляторах.

В нашей реализации будут использоваться две внешние библиотеки. Библиотека Pygame, предназначенная для разработки игр на Python, позволяет легко создавать окна на экране, заполнять их пикселями с дисплея нашей виртуальной машины и обрабатывать ввод с клавиатуры. Библиотека NumPy для числовых операций поможет создать двумерный массив, который будет использоваться в качестве буфера для пикселей окна Pygame. Этот массив будет служить «графической оперативной памятью» нашей виртуальной машины. Библиотека Pygame изначально работает с массивами NumPy, а массивы NumPy более производительны, чем что-либо в стандартной библиотеке Python для представления этого буфера. Перед запуском программы убедитесь, что вы установили Pygame и NumPy.

Как и при воспроизведении формата файла в главе 3, реализация виртуальной машины или эмулятора требует значительного количества низкоуровневых операций с битами. См. приложение, чтобы ознакомиться с битовыми операторами Python.

Цикл выполнения

Цикл выполнения отвечает за продвижение VM на одну инструкцию, перерисовку экрана, обработку любых событий (нажатия клавиш, которые должны быть переданы VM), воспроизведение звукового сиг-

нала и обновление двух таймеров CHIP-8. С помощью Pygame рисование, воспроизведение звуков и чтение ввода с клавиатуры становятся практически элементарными задачами; это очень простая в использовании библиотека. Начнем с кода инициализации и продолжим до начала цикла выполнения:

Chip8/__main__.py

```
import sys
from argparse import ArgumentParser
from Chip8.vm import VM, SCREEN_WIDTH, SCREEN_HEIGHT
from Chip8.vm import TIMER_DELAY, FRAME_TIME_EXPECTED, ALLOWED_KEYS
import pygame
from timeit import default_timer as timer
import os

def run(program_data: bytes, name: str):
    # Запуск Pygame, создание окна и загрузка звука
    pygame.init()
    screen = pygame.display.set_mode((SCREEN_WIDTH, SCREEN_HEIGHT),
                                     pygame.SCALED)
    pygame.display.set_caption(f"Chip8 - {os.path.basename(name)}")
    bee_sound = pygame.mixer.Sound(os.path.dirname(os.path.realpath(__file__))
                                   + "/bee.wav")
    currently_playing_sound = False
    vm = VM(program_data) # загрузка виртуальной машины с данными программы
    timer_accumulator = 0.0 # используется для ограничения таймера до 60 Гц
    # Основной цикл виртуальной машины
    while True:
        frame_start = timer()
        vm.step()
        if vm.needs_redraw:
            pygame.surfarray.blit_array(screen, vm.display_buffer)
            pygame.display.flip()
```

В начале цикла выполнения время записывается с помощью `frame_start = timer()`, чтобы измерить продолжительность каждой итерации цикла. Это связано с тем, что таймеры CHIP-8 необходимо уменьшать 60 раз в секунду (если они больше нуля). Затем VM получает команду выполнить инструкцию (и, следовательно, перейти к следующей инструкции) с помощью `vm.step()`. Если это указано в `vm.needs_redraw`, дисплей перерисовывается с помощью двух простых вызовов Pygame. Один копирует буфер дисплея VM на экран, а другой отображает его.

Обратите внимание, что в коде термин «*фрейм*» (frame) используется немного иначе, чем обычно. В большинстве программ фрейм – это одно полное обновление всего графического вывода программы, но в данном контексте наш цикл выполнения не обязательно будет перерисовывать графику при каждой итерации, поскольку `vm.needs_redraw` не всегда может быть `True`.

Что определенно *будет* происходить в каждом «фрейме», так это то, что одна *инструкция* будет выполняться в результате вызова `vm.step()`. Поэтому я подумал об использовании слова «инструкция» вместо «фрейм» в этом разделе кода, например `instruction_start` вместо `frame_start`. Однако в цикле выполнения происходит не только выполнение инструкции – есть еще графический вывод, обработка клавиатуры и звуковой вывод. Поэтому слово «*инструкция*» показалось слишком ограниченным. Но и «*фрейм*» – тоже не совсем точное слово. Верно ведь говорят, что одна из самых сложных проблем в информатике – это присвоение имен.

Цикл выполнения заканчивается обработкой событий клавиатуры, воспроизведением звука, когда булево значение `vm.play_sound` виртуальной машины указывает на это, и временных параметров:

```
# Обработка событий клавиатуры
for event in pygame.event.get():
    if event.type == pygame.KEYDOWN:
        key_name = pygame.key.name(event.key)
        if key_name in ALLOWED_KEYS:
            vm.keys[ALLOWED_KEYS.index(key_name)] = True
    elif event.type == pygame.KEYUP:
        key_name = pygame.key.name(event.key)
        if key_name in ALLOWED_KEYS:
            vm.keys[ALLOWED_KEYS.index(key_name)] = False
    elif event.type == pygame.QUIT:
        sys.exit()

# Звук
if vm.play_sound:
    if not currently_playing_sound:
        bee_sound.play(-1)
        currently_playing_sound = True
else:
    currently_playing_sound = False
    bee_sound.stop()

# Управление временными событиями
frame_end = timer()
frame_time = frame_end - frame_start # время получено в секундах
timer_accumulator += frame_time
# Каждую 1/60 секунды происходит уменьшение таймеров
if timer_accumulator > TIMER_DELAY:
  ❶ vm.decrement_timers()
    timer_accumulator = 0
# Ограничение скорости всей машины до 500 "фреймов" в секунду
if frame_time < FRAME_TIME_EXPECTED:
    difference = FRAME_TIME_EXPECTED - frame_time
  ❷ pygame.time.delay(int(difference * 1000))
    timer_accumulator += difference
```

Несмотря на то что мы не используем фреймы для измерения традиционных кадров в секунду (frames per second, FPS), как вы, возможно, знаете из игр, время каждой итерации по-прежнему имеет важное значение. Нам необходимо отслеживать время, чтобы обеспечить отсчет таймеров виртуальной машины каждые 1/60 секунды, как того требует спецификация CHIP-8 ❶, и ограничить общую скорость виртуальной машины ❷. Если виртуальная машина работает слишком быстро, игры будут непригодны, поскольку они были разработаны для медленных компьютеров 1970-х годов. Вы можете настроить скорость виртуальной машины и, следовательно, любого программного обеспечения, работающего на ней, изменив константу `FRAME_TIME_EXPECTED` в *vm.py*. В ходе тестирования я обнаружил, что 500 «кадров» в секунду, или, другими словами, каждый «кадр» длительностью примерно 1/500 секунды, является оптимальной скоростью для большинства игр.

Аргументы командной строки

Как и в предыдущих программах, для обработки аргументов командной строки используется `ArgumentParser`:

```
if __name__ == "__main__":
    # Парсинг аргумента файла
    file_parser = ArgumentParser("Chip8")
    file_parser.add_argument("rom_file",
                             help="A file containing a Chip-8 game.")
    arguments = file_parser.parse_args()
    with open(arguments.rom_file, "rb") as fp:
        file_data = fp.read()
        run(file_data, arguments.rom_file)
```

В данном случае у нас есть только один аргумент командной строки = имя файла, который содержит данные программы для виртуальной машины CHIP-8. Необработанные байты файла считываются и передаются в `run()`, где они, в свою очередь, передаются в конструктор VM.

Настройка VM и вспомогательные функции

Мы готовы к фактической реализации VM. Как обычно, начнем с нескольких констант:

Chip8/vm.py

```
from array import array
from random import randint
import numpy as np
import pygame
import sys
```

```
RAM_SIZE = 4096 # в байтах, то есть 4 килобайта
SCREEN_WIDTH = 64
SCREEN_HEIGHT = 32
SPRITE_WIDTH = 8
WHITE = 0xFFFFFFFF
BLACK = 0
TIMER_DELAY = 1/60 # в секундах ... примерно 60 Гц
FRAME_TIME_EXPECTED = 1/500 # для ограничения скорости VM
ALLOWED_KEYS = ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9",
                "a", "b", "c", "d", "e", "f"]

# Набор шрифтов, жестко запрограммированный
FONT_SET = [
    0xF0, 0x90, 0x90, 0x90, 0xF0, # 0
    0x20, 0x60, 0x20, 0x20, 0x70, # 1
    0xF0, 0x10, 0xF0, 0x80, 0xF0, # 2
    0xF0, 0x10, 0xF0, 0x10, 0xF0, # 3
    0x90, 0x90, 0xF0, 0x10, 0x10, # 4
    0xF0, 0x80, 0xF0, 0x10, 0xF0, # 5
    0xF0, 0x80, 0xF0, 0x90, 0xF0, # 6
    0xF0, 0x10, 0x20, 0x40, 0x40, # 7
    0xF0, 0x90, 0xF0, 0x90, 0xF0, # 8
    0xF0, 0x90, 0xF0, 0x10, 0xF0, # 9
    0xF0, 0x90, 0xF0, 0x90, 0x90, # A
    0xE0, 0x90, 0xE0, 0x90, 0xE0, # B
    0xF0, 0x80, 0x80, 0x80, 0xF0, # C
    0xE0, 0x90, 0x90, 0x90, 0xE0, # D
    0xF0, 0x80, 0xF0, 0x80, 0xF0, # E
    0xF0, 0x80, 0xF0, 0x80, 0x80 # F
]
```

Большинство этих констант не требуют пояснений и соответствуют исходным спецификациям CHIP-8. Виртуальная машина имеет 4 КБ основной памяти. Она определяет графику в виде черно-белого изображения с разрешением 64×32. Таймеры обновляются 60 раз в секунду. Исходные системы CHIP-8 имели 16 клавиш, которые можно было нажимать на контроллере. Вероятно, для игр их можно было бы расположить более эргономично, сопоставив с другими клавишами, но в нашей реализации мы просто оставим клавиши там, где они находятся на клавиатуре.

Самой необычной константой, пожалуй, в этом случае является `FONT_SET`. Это 80 байт графических данных для отображения цифр 0–9 и букв A–F. Каждый символ задается битами, представляющими пиксели символа, который должен отображаться на экране. Считайте это примитивным шрифтом, который имеет только 16 символов. В некоторых играх эти данные должны находиться в первых 80 байтах памяти, чтобы можно было выводить сообщения пользователю на экран.

Далее у нас есть вспомогательная функция, не связанная с состоянием виртуальной машины:

```
def concat_nibbles(*args: int) -> int:
    result = 0
    for arg in args:
        result = (result << 4) | arg
    return result
```

Функция `concat_nibbles()` принимает произвольное количество целых чисел и соединяет их одно за другим, сдвигая каждое на 4 бита влево и выполняя побитовое OR со следующим. Это пригодится только в том случае, если сами целые числа имеют размер 4 бита. Предположим, у нас есть целое число `0111`. Сдвиг на 4 бита влево приведет к тому, что за исходными 4 битами последуют четыре нуля, как в `01110000`. Теперь предположим, что у нас есть еще одно 4-битное целое число, `1010`. Если мы выполним операцию OR с `01110000`, то получим результат `01111010`, который является соединением двух исходных 4-битных целых чисел. Мы можем продолжать делать это для произвольного количества 4-битных целых чисел, чтобы соединить их вместе.

Напомним, что 4-разрядное целое число называется *полубайтом*, или *нибблом* (nibble). В CHIP-8 16-битные инструкции делятся на четыре ниббла, и каждый ниббл зачастую имеет отдельное значение. По умолчанию мы будем делить каждую инструкцию на четыре составляющих полубайта, но для некоторых инструкций нам понадобится использовать значение нескольких объединенных нибблов. Отсюда и полезность вспомогательной функции `concat_nibbles()`.

Класс `VM` начинается с конструктора, который инициализирует в нем все изменяемые состояния, включая регистры, ОЗУ, стек, буфер отображения (то, что сегодня мы называем VRAM, или видеопамятью), таймеры и несколько других вспомогательных переменных:

```
class VM:
    def __init__(self, program_data: bytes):
        # Инициализированные регистры и конструкты памяти
        # Регистры общего назначения - CHIP-8 имеет 16 таких регистров
        self.v = array('B', [0] * 16)
        # Регистр Index
        self.i = 0
        # Программный счетчик
        # Стартует от 0x200, поскольку адреса ниже использует
        # сама VM в оригинальных машинах CHIP-8
        self.pc = 0x200
        # Память – стандартные 4k в оригинальных машинах CHIP-8
        self.ram = array('B', [0] * RAM_SIZE)
        # Загрузка набора шрифтов в первые 80 байт
        self.ram[0:len(FONT_SET)] = array('B', FONT_SET)
        # Копирование программы в ОЗУ, начиная с байта 512 по соглашению
        self.ram[512:(512 + len(program_data))] = array('B', program_data)
        # Стек - в реальном оборудовании он обычно ограничен
```

```
        # 12 или 16 адресами ПК для переходов, но мы используем современное
        # оборудование, и наш стек может расширяться/сокращаться неограниченно
        # по мере необходимости.
        self.stack = []
        # Графический буфер для экрана - 64x32 пикселя
        self.display_buffer = np.zeros((SCREEN_WIDTH, SCREEN_HEIGHT),
                                       dtype=np.uint32)
        self.needs_redraw = False
        # Таймеры – очень простые регистры, которые отсчитывают до 0
        # с частотой 60 Гц
        self.delay_timer = 0
        self.sound_timer = 0
        # Они хранят информацию о том, нажаты ли клавиши
        # CHIP-8 имеет 16 клавиш
        self.keys = [False] * 16
```

Некоторые из этих переменных состояния имеют важные значения по умолчанию. Например, программный счетчик (`pc`) всегда должен быть установлен на место `0x200` (512 в десятичной системе), поскольку первые 512 байт памяти в машинах CHIP-8 изначально использовались для хранения самой виртуальной машины CHIP-8. Это означает, что программы CHIP-8 не могли использовать эту память и должны были начинаться с байта 512. Я подробно прокомментировал конструктор, чтобы объяснить каждую переменную по мере ее объявления. Обратите внимание, что подавляющая часть нашей VM использует для своей реализации стандартную библиотеку Python, за исключением `display_buffer`, являющегося массивом NumPy. Это формат, который поддерживает Pygame.

Далее у нас есть простой вспомогательный метод `decrement_timers()` и простое динамическое свойство `play_sound`:

```
    def decrement_timers(self):
        if self.delay_timer > 0:
            self.delay_timer -= 1
        if self.sound_timer > 0:
            self.sound_timer -= 1

    @property
    def play_sound(self) -> bool:
        return self.sound_timer > 0
```

Обе функции, `decrement_timers()` и `play_sound`, использовались в цикле выполнения, который мы рассматривали ранее в *__main__.py*.

Графика

В CHIP-8 экран представляет собой плоскость размером 64×32 пикселя в декартовой системе координат, начало которой находится в левом верхнем углу (0,0), а ось *y* направлена вниз. Другими словами,

координата *x* увеличивается по мере перемещения слева направо, а координата *y* увеличивается по мере перемещения сверху вниз. Таким образом, пиксель в правом нижнем углу находится в точке (63,31). Отрицательных координат не существует, и доступ к пикселям за пределами экрана невозможен.

Каждый пиксель представлен в памяти в виде одного бита. В нашей реализации 1 представляет белый пиксель, а 0 – черный пиксель. Графическая память (или «буфер») отделена от основной памяти программы и может управляться только косвенно с помощью инструкций CHIP-8. В Pygame для представления пикселей на экране в формате RGBA (буква A обозначает *альфа-канал*, или прозрачность) используются 32-битные целые числа, поэтому каждое из наших 1-битных значений пикселей при сохранении в `display_buffer` должно преобразовываться в 32-битное целое число.

Для рисования CHIP-8 использует *спрайты* (sprites) – небольшие растровые изображения (или картинки, если хотите), которые могут перемещаться по экрану. Каждый спрайт в CHIP-8 имеет ширину 8 пикселей и высоту от 1 до 15 пикселей. На рис. 5.1 показан спрайт размером 8×3 с изображением слова *HI* (привет), нарисованный на экране в точке (28,15).

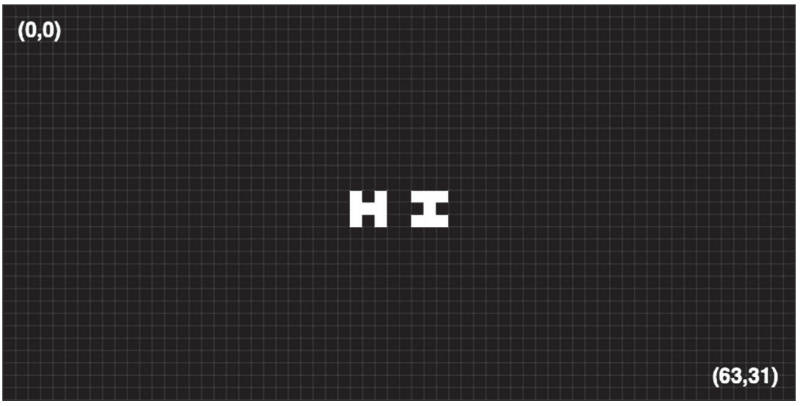


Рис. 5.1. Слово HI в виде спрайта 8×3

Поскольку каждая строка в спрайте CHIP-8 имеет ширину ровно 8 пикселей, она отображается с помощью 8 бит. Так как 8 бит составляют 1 байт, каждая строка спрайта может быть представлена одним байтом. Поскольку спрайт *HI* имеет высоту три строки, он может быть представлен 3 байтами. В двоичном формате эти 3 байта будут выглядеть следующим образом:

```
10100111
11100010
10100111
```

Обратите внимание, что каждая единица соответствует белому пикселю, а каждый ноль – черному пикселю. С этой информацией, надеюсь, набор используемых шрифтов, который мы определили ранее, также приобретает больше смысла: каждый символ в наборе шрифтов представляет собой просто спрайт размером 8×5.

Рисование спрайтов – это единственный способ изменить буфер отображения, кроме его очистки, поэтому виртуальная машина CHIP-8 имеет одну инструкцию рисования, `Dxyn`. Она рисует спрайт заданной высоты, расположенный в ячейке памяти, указанной регистром `i`. Буква `D` в инструкции – это постоянная четверть байта, а четверти байта `x` и `y` представляют индексы в регистрах `v`, где должны находиться координаты *x* и *y* для левого верхнего угла спрайта. Другими словами, координата *x* извлекается из регистра `v[x]`, а координата *y* – из регистра `v[y]`. Четверть байта `n` представляет высоту спрайта. Вот почему спрайты не могут быть выше 15 пикселей: ниббл состоит из 4 бит, а 4 бита могут представить максимальное число 15.

Ниббл `Dxyn` соответствуют параметрам вспомогательного метода `draw_sprite()`:

```
# Рисование спрайта в точке *x*, *y* по данным из *i* и высотой *height*
def draw_sprite(self, x: int, y: int, height: int):
    flipped_black = False # был ли перевернут какой-либо пиксель при рисовании?
    for row in range(0, height):
        row_bits = self.ram[self.i + row]
        for col in range(0, SPRITE_WIDTH):
            px = x + col
            py = y + row
            if px >= SCREEN_WIDTH or py >= SCREEN_HEIGHT:
                continue # игнорирование пикселей за пределами экрана
            new_bit = (row_bits >> (7 - col)) & 1
            old_bit = self.display_buffer[px, py] & 1
            if new_bit & old_bit: # если установлены оба, перевернуть white -> black
                flipped_black = True
            # CHIP-8 рисует с помощью операции XOR
            new_pixel = new_bit ^ old_bit
            self.display_buffer[px, py] = WHITE if new_pixel else BLACK
    # Установка флага переворота для обнаружения коллизий
    self.v[0xF] = 1 if flipped_black else 0
```

В CHIP-8 для рисования спрайтов используются операции XOR. Операция *XOR* (*исключающее ИЛИ*) – это побитовая операция, которая возвращает 1, если два бита отличаются, и 0, если они одинаковы. В Python для XOR используется оператор `^`. В табл. 5.2 представлена схема истинности для XOR.

`0^0`	`0^1`	`1^0`	`1^1`
0	1	1	0

Таблица 5.2 Схема истинности для XOR

Инструкция рисования CHIP-8 берет спрайт и выполняет операции XOR над его пикселями с теми пикселями, которые уже находятся на экране в указанном месте. Если это место на экране состоит из черных пикселей, то в результате просто будет нарисован спрайт. Однако если это место на экране содержит белые пиксели (1), то черные пиксели будут нарисованы там, где белые пиксели спрайта перекрываются с белыми пикселями экрана. Это происходит по той причине, что 1 XOR 1 равняется 0. Инструкция рисования CHIP-8 отслеживает случаи такого перекрытия (белый пиксель экрана превратился в черный пиксель при рисовании спрайта). В случае перекрытия она устанавливает регистр флага (`v[0xF]`).

Метод `draw_sprite()` является кодификацией этого процесса. Мы просматриваем все строки и столбцы спрайта, который начинается в ячейке памяти, указанной регистром `i`, и извлекаем каждый пиксель спрайта с помощью операции сдвига вправо, сохраняя его в `new_bit`. Операция `&` с данными, поступающими в `new_bit`, гарантирует, что в `new_bit` сохраняется только последний бит операции сдвига. Мы сравниваем каждый `new_bit` с тем битом, который уже находится на экране, `old_bit`, и если `old_bit` будет перевернут с белого на черный, мы устанавливаем регистр флага. Мы изменяем буфер отображения, выполняя операцию XOR для `new_bit` и `old_bit`.

Зачем нам нужен флаг для отслеживания того, приводит ли рисование спрайта к отключению ранее освещенного пикселя экрана? По сути, это форма выявления коллизий. Если спрайт сталкивается с чем-то, что уже было на экране, это особенно важно учитывать в играх. Например, если вы программируете игру в теннис, вам нужно знать, когда мяч движется и попадает в ракетку, уже находящуюся на экране.

Выполнение инструкций

Настало время перейти к главному механизму VM. У нас остался один метод, но он очень важный: нам требуется выполнить все инструкции виртуальной машины. Это не отличается от выполнения операторов в наших интерпретаторах из глав 1 и 2. Независимо от того, выполняем ли мы операторы интерпретатора, инструкции VM или операционные коды микропроцессора в эмуляторе, нам необходимо сделать нечто довольно простое: распознать следующую инструкцию, а затем выполнить несколько строк кода, которые управляют состоянием VM в соответствии с ее целевой операцией.

Например, если мы видим инструкцию «add», то нужно сложить два указанных числа и сохранить результат в указанном месте. Если мы видим инструкцию «jump», то нужно перейти к выполнению инструкции в указанном месте памяти. Речь идет буквально о распознавании выполняемой инструкции и изменении нескольких переменных состояния, представляющих память, регистры и т. п., согласно этой инструкции. Самый простой способ данной реализации предполага-

ет использование большого количества операторов if. Соответствующий псевдокод может выглядеть следующим образом:

```
if instruction == ADD:
    сложить некоторые числа и сохранить сумму
elif instruction == JUMP:
    перейти к указанному месту, изменив программный счетчик
elif instruction == DRAW:
    нарисовать спрайт в указанном месте, изменив буфер отображения
и т. д.
```

Помимо использования множества операторов if, существует три распространенных шаблона записи кода для выполнения инструкций. Первый заключается в использовании гигантского оператора switch – конструкции, присутствующей во многих языках, но не существующей в Python в этой же форме. Полагаю, что большинство читателей уже сталкивались с оператором switch в таких языках, как C или Java. Если нет, то можно представить его как примитивную форму оператора match в Python, который мы использовали в главах 1 и 2. Выполнение оператора switch зависит от инструкции. Это в некоторой степени похоже на только что показанный псевдокод. В действительности до введения оператора match в Python 3.10 этот способ реализовывался в Python с помощью множества операторов if и elif. Это самый простой способ реализации выполнения инструкций, но он может оказаться неудобным в случае работы с большим набором инструкций.

Следующий способ заключается в применении *таблицы переходов* (jump table), которая состоит из массива указателей на функции. Мы индексируем массив в зависимости от инструкции, а затем выполняем соответствующую возвращаемую функцию. Инструкции представляют собой обычные целые числа, поэтому их можно использовать в качестве индексов массива. Если бы инструкции по какой-то причине были строками, мы могли бы вместо этого использовать словарь, в котором ключами являются инструкции, а значениями – указатели функций, хотя это немного менее эффективно. Поскольку этот механизм распределяет работу между многими вспомогательными функциями, он обычно приводит к более чистому коду, чем огромный оператор switch, и может быть предпочтительным при работе с большим набором инструкций.

Третий способ заключается в использовании *динамической перекомпиляции* (dynamic recompilation), где каждая инструкция переводится в инструкцию, понятную базовому оборудованию (или в нечто, что может быть далее переведено в такую инструкцию). Например, если у нас есть инструкция сложения в виртуальной машине, работающей на микропроцессоре x86, мы можем перевести инструкцию сложения VM в машинный код для эквивалентной инструкции сло-

жения на платформе x86. Это самый сложный для реализации подход, поскольку он требует глубокого знания не только исходного набора инструкций, но и целевого набора инструкций. Однако он позволяет добиться максимальной производительности.

В этой программе мы будем использовать гигантское выражение `match`, поскольку набор инструкций CHIP-8 относительно невелик. Когда в следующей главе мы будем создавать эмулятор NES, то воспользуемся таблицей переходов, поскольку микропроцессор 6502 имеет набор инструкций, размер которого примерно в два раза больше (хотя и по-прежнему намного меньше, чем у почти всех других микропроцессоров). Динамическая перекомпиляция – это значительно более сложная техника, выходящая за рамки данной книги.

Метод `step()` отвечает за выполнение инструкций, но сначала ему необходимо получить следующую инструкцию для выполнения:

```
def step(self):
    # Мы рассматриваем операционный код в виде его нибблов (4-битных
    # фрагментов)
    # Операционный код состоит из 16 бит, которые составляют следующие
    # два байта в памяти
    first2 = self.ram[self.pc]
    last2 = self.ram[self.pc + 1]
    first = (first2 & 0xF0) >> 4
    second = first2 & 0xF
    third = (last2 & 0xF0) >> 4
    fourth = last2 & 0xF

    self.needs_redraw = False
    jumped = False
```

Следующая инструкция находится по адресу памяти, сохраненному в программном счетчике (`pc`). Поскольку инструкции состоят из 16 бит, мы извлекаем следующие 2 байта по адресу `pc` и сохраняем их в `first2` и `last2`. Как обсуждалось ранее, удобно думать о каждой инструкции CHIP-8 как о комбинации четырех нибблов, поскольку каждый отдельный ниббл имеет значение для многих инструкций. Мы сохраняем нибблы в `first`, `second`, `third` и `fourth`. Все сопоставления шаблонов для наших инструкций будут осуществляться в категориях нибблов.

При выполнении инструкции мы также будем отслеживать, требует ли она перерисовки через `needs_redraw` и изменила ли она `pc` через `jumped`. Цикл выполнения использует `needs_redraw` как оптимизацию. Зачем рисовать, если ничего не изменилось? Отслеживание `jumped` позволяет разместить некоторый общий код внизу `step()`, что немного сокращает дублирование кода.

Теперь мы подошли к собственно инструкциям. Перед нами гигантское выражение `match`. В нашей реализации используется элегантный

синтаксис match языка Python для захвата нибблов, необходимых для выполнения инструкции во временных переменных. Подробности выполнения каждой инструкции вытекают непосредственно из ее описания, приведенного ранее в этой главе. Многие инструкции можно реализовать всего одной строкой кода. Было бы слишком скучно описывать каждую из них по очереди. Вместо этого ниже приводится воспроизведение остальной части step() с комментариями, обеспечивающими дополнительный контекст.

Однако прежде чем смотреть на код, стоит остановиться и попробовать реализовать инструкции самостоятельно. Необязательно использовать оператор match. Можно использовать серию операторов if...elif, как я делал в Python 3.9 до появления оператора match. (Я проверил, и между ними практически нет разницы в производительности.) У вас уже есть все необходимое, чтобы сосредоточиться только на том, что должна делать каждая инструкция, вместо настройки памяти системы или представления регистров. Вам не нужно думать о загрузке файла ПЗУ или о том, какими должны быть некоторые константы. Просто подумайте о логике и о том, каким образом каждая операция изменит состояние VM.

Некоторые описания инструкций, приведенные ранее в этой главе, были довольно краткими, но вы можете найти более подробные инструкции в любом из множества онлайн-справочников по CHIP-8. Однако не тратьте слишком много времени на одну инструкцию. Если вы вдруг застрянете, всегда можете посмотреть реализацию в этой книге. После того как вы попробуете написать свои собственные реализации инструкций, вы можете вернуться к коду этой книги, чтобы перепроверить свою работу. Сделав эту работу самостоятельно, вы получите хорошее представление о том, что нужно для написания простой виртуальной машины или эмулятора. Не сомневайтесь: вы будете удивлены простотой реализации многих инструкций. Помните, что оригинальная виртуальная машина CHIP-8 помещалась всего в 512 байтах памяти!

```
match (first, second, third, fourth):
    case (0x0, 0x0, 0xE, 0x0): # очистка дисплея
        self.display_buffer.fill(0)
        self.needs_redraw = True
    case (0x0, 0x0, 0xE, 0xE): # возврат из подпрограммы
        self.pc = self.stack.pop()
        jumped = True
    case (0x0, n1, n2, n3): # вызов программы
        self.pc = concat_nibbles(n1, n2, n3) # go to start
        # Очистка регистров
        self.delay_timer = 0
        self.sound_timer = 0
        self.v = array('B', [0] * 16)
        self.i = 0
```

```
            # Очистка экрана
            self.display_buffer.fill(0)
            self.needs_redraw = True
            jumped = True
        case (0x1, n1, n2, n3): # переход по адресу
            self.pc = concat_nibbles(n1, n2, n3)
            jumped = True
        case (0x2, n1, n2, n3): # вызов подпрограммы
            self.stack.append(self.pc + 2) # размещение возвращаемого значения в стеке
            self.pc = concat_nibbles(n1, n2, n3) # переход к подпрограмме
            jumped = True
        case (0x3, x, _, _): # условный пропуск v[x] равно last2
            if self.v[x] == last2:
                self.pc += 4
                jumped = True
        case (0x4, x, _, _): # условный пропуск v[x] не равно last2
            if self.v[x] != last2:
                self.pc += 4
                jumped = True
        case (0x5, x, y, _): # условный пропуск v[x] равно v[y]
            if self.v[x] == self.v[y]:
                self.pc += 4
                jumped = True
        case (0x6, x, _, _): # установка v[x] на last2
            self.v[x] = last2
        case (0x7, x, _, _): # добавление last2 к v[x]
            self.v[x] = (self.v[x] + last2) % 256
        case (0x8, x, y, 0x0): # установка v[x] на v[y]
            self.v[x] = self.v[y]
        case (0x8, x, y, 0x1): # установка v[x] на v[x] | v[y]
            self.v[x] |= self.v[y]
        case (0x8, x, y, 0x2): # установка v[x] на v[x] & v[y]
            self.v[x] &= self.v[y]
        case (0x8, x, y, 0x3): # установка v[x] на v[x] ^ v[y]
            self.v[x] ^= self.v[y]
        case (0x8, x, y, 0x4): # добавление с флагом переноса
            try:
                self.v[x] += self.v[y]
                self.v[0xF] = 0 # указание отсутствия флага переноса
            except OverflowError:
                self.v[x] = (self.v[x] + self.v[y]) % 256
                self.v[0xF] = 1 # установка флага переноса
        case (0x8, x, y, 0x5): # вычитание с флагом заимствования
            try:
                self.v[x] -= self.v[y]
                self.v[0xF] = 1 # указание отсутствия заимствования (да, странно, но это 1)
            except OverflowError:
                self.v[x] = (self.v[x] - self.v[y]) % 256
                self.v[0xF] = 0 # указание, что это было заимствование
        case (0x8, x, _, 0x6): # v[x] >> 1 v[f] = least significant bit
            self.v[0xF] = self.v[x] & 0x1
```

```
            self.v[x] >>= 1
        case (0x8, x, y, 0x7): # вычитание с флагом заимствования (y - x in x)
            try:
                self.v[x] = self.v[y] - self.v[x]
                self.v[0xF] = 1 # указание отсутствия заимствования (да, странно, но это 1)
            except OverflowError:
                self.v[x] = (self.v[y] - self.v[x]) % 256
                self.v[0xF] = 0 # указание отсутствия заимствования
        case (0x8, x, _, 0xE): # v[x] << 1 v[f] = most significant bit
            self.v[0xF] = (self.v[x] & 0b10000000) >> 7
            self.v[x] = (self.v[x] << 1) & 0xFF
        case (0x9, x, y, 0x0): # условный пропуск, если v[x] != v[y]
            if self.v[x] != self.v[y]:
                self.pc += 4
                jumped = True
        case (0xA, n1, n2, n3): # установка i на адрес n1n2n3
            self.i = concat_nibbles(n1, n2, n3)
        case (0xB, n1, n2, n3): # переход на n1n2n3 + v[0]
            self.pc = concat_nibbles(n1, n2, n3) + self.v[0]
            jumped = True
        case (0xC, x, _, _): # v[x] = произвольное число (0-255) & last2
            self.v[x] = last2 & randint(0, 255)
        case (0xD, x, y, n): # рисование спрайта в (vx, vy) высотой n
            self.draw_sprite(self.v[x], self.v[y], n)
            self.needs_redraw = True
        case (0xE, x, 0x9, 0xE): # условный пропуск, если нажаты клавиши (v[x])
            if self.keys[self.v[x]]:
                self.pc += 4
                jumped = True
        case (0xE, x, 0xA, 0x1): # условный пропуск, если не нажаты клавиши (v[x])
            if not self.keys[self.v[x]]:
                self.pc += 4
                jumped = True
        case (0xF, x, 0x0, 0x7): # установка v[x] для delay_timer
            self.v[x] = self.delay_timer
        case (0xF, x, 0x0, 0xA): # ожидание нажатия следующей клавиши и сохранение в v[x]
            # Ожидание нажатия следующей клавиши, затем продолжение
            while True:
                event = pygame.event.wait()
                if event.type == pygame.QUIT:
                    sys.exit()
                if event.type == pygame.KEYDOWN:
                    key_name = pygame.key.name(event.key)
                    if key_name in ALLOWED_KEYS:
                        self.v[x] = ALLOWED_KEYS.index(key_name)
                        break
        case (0xF, x, 0x1, 0x5): # установка delay_timer на v[x]
            self.delay_timer = self.v[x]
        case (0xF, x, 0x1, 0x8): # установка sound_timer на v[x]
            self.sound_timer = self.v[x]
        case (0xF, x, 0x1, 0xE): # добавление vx к i
```

```
            self.i += self.v[x]
        case (0xF, x, 0x2, 0x9): # установка i на локацию символа v[x]
            self.i = self.v[x] * 5 # встроенный набор шрифтов имеет интервал 5 байт
        case (0xF, x, 0x3, 0x3): # сохранение BCD в v[x] при i, i+1, i+2
            self.ram[self.i] = self.v[x] // 100 # разряд сотен
            self.ram[self.i + 1] = (self.v[x] % 100) // 10 # разряд десятков
            self.ram[self.i + 2] = (self.v[x] % 100) % 10 # разряд единиц
        case (0xF, x, 0x5, 0x5): # выгрузка регистров v0 в vx начиная с i
            for r in range(0, x + 1):
                self.ram[self.i + r] = self.v[r]
        case (0xF, x, 0x6, 0x5): # сохранение i через i+r в v0 через vr
            for r in range(0, x + 1):
                self.v[r] = self.ram[self.i + r]
        case _:
            print(f"Unknown opcode {(hex(first), hex(second),
                                     hex(third), hex(fourth))}!")

    if not jumped:
        self.pc += 2 # инкремент счетчика программы
```

В конце `step()` мы увеличиваем счетчик программы, если переход не выполнялся. Это гарантирует, что при следующем вызове `step()` мы перейдем к следующей инструкции. Поскольку каждая инструкция CHIP-8 имеет длину 2 байта, программный счетчик увеличивается на 2. Если произошел переход, то выполнение было непосредственно перенесено к другой инструкции в другом месте памяти.

Тестирование VM

Наиболее детальным способом тестирования виртуальной машины было бы написание собственных модульных тестов для каждой из инструкций. В каждом тестировании мы бы пробовали запустить инструкцию, а затем убедились, что последующее внутреннее состояние VM было правильным. Хотя это было бы идеальным вариантом, в интересах экономии времени и места вместо этого мы сделаем что-то, более похожее на интеграционные тесты: посмотрим, как наша VM работает с реальными программами CHIP-8. Выполняются ли они корректно?

Так получилось, что существуют даже тестовые ПЗУ, которые предлагают своего рода универсальный инструмент для тестирования виртуальной машины CHIP-8. Два таких тестовых ПЗУ включены в подкаталог *Chip8/Tests* репозитория исходного кода книги. Оба тестовых ПЗУ были выпущены их разработчиками под открытыми лицензиями, которые включены в эти подкаталоги. Запустим первый тестовый ПЗУ из домашнего каталога репозитория:

```
% python3 -m Chip8 Chip8/Tests/chip8-test-rom/test_opcode.ch8
```

Если виртуальная машина работает правильно, вы должны увидеть экран с надписями OK, как показано на рис. 5.2.

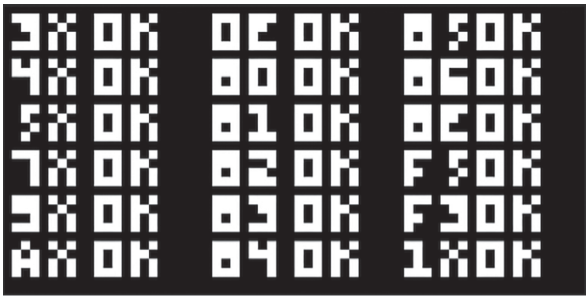


Рис. 5.2. Запуск первого тестового ПЗУ

Теперь давайте проверим нашу работу со вторым тестовым ПЗУ:

```
% python3 -m Chip8 Chip8/Tests/chip8-test-rom-2/chip8-test-rom.ch8
```

Этот тест просто отображает OK один раз в левом верхнем углу (см. рис. 5.3).



Рис. 5.3. Запуск второго тестового ПЗУ

Эти тесты не являются исчерпывающими, но они являются хорошей отправной точкой. Теперь пришло время для окончательных интеграционных тестов: сможет ли наша виртуальная машина правильно запускать игры?

Запуск игр

Подкаталог *Chip8/Games* в репозитории книги содержит подборку ПЗУ CHIP-8, которые были размещены в открытом доступе. Если вы считаете, что схемы управления некоторых из них немного неудобны, попробуйте изменить стандартные настройки клавиш. В настоящее время `ALLOWED_KEYS` считываются непосредственно с соответствую-

щих клавиш, поэтому *A* в виртуальной машине – это клавиша *A* на клавиатуре. Однако системы, на которых эти игры запускались, могли иметь совершенно разные раскладки клавиатуры, поэтому для некоторых игр может быть удобнее использование другой схемы.

Большинство игр довольно просты, что вполне логично с учетом ограничений аппаратного обеспечения, на котором изначально должна была работать VM. Существуют клоны популярных игр для более мощных систем. Для начала у нас есть *BLINKY* – своего рода клон *Pac-Man* (см. рис. 5.4).

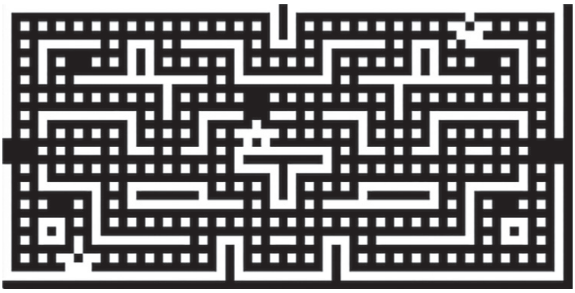


Рис. 5.4. Игра BLINKY, запущенная на VM

Игра *INVADERS* является клоном *Space Invaders* (см. рис. 5.5).

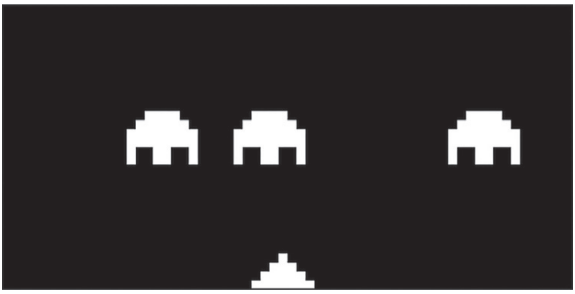


Рис. 5.5. Игра INVADERS, запущенная на VM

Игра *VBRIX* – это вертикальная версия игры *Breakout* (см. рис. 5.6).

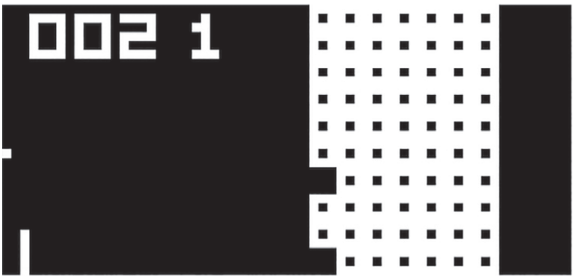


Рис. 5.6. Игра VBRIX, запущенная на VM

И наконец, игра *PONG* (см. рис. 5.7).

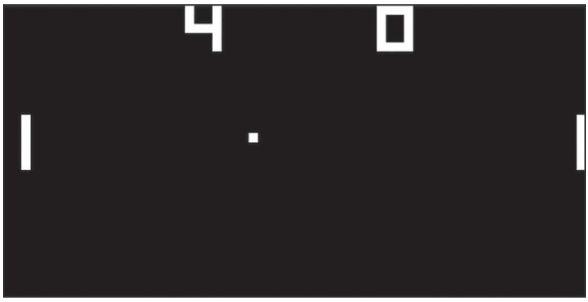


Рис. 5.7. Игра PONG, запущенная на VM

В репозитории исходного кода есть еще несколько игр, которые вы можете проверить самостоятельно. Обратите внимание на размер файлов: большинство из этих игр имеют размер 500 байт или меньше! Самая большая, *BLINKY*, занимает всего 2 Кбайта.

Код в реальной жизни

Я всегда был заинтересован в разработке собственного эмулятора, но не чувствовал себя достаточно уверенно для его создания до тех пор, пока не прошел значительную часть своего пути в программировании. Когда я начал изучение вопроса создания эмулятора, стандартным советом, который я нашел, было сначала попробовать написать виртуальную машину CHIP-8, так как это проще, чем написание практически любого эмулятора, но требует всех тех же элементов (обработка операционных кодов, симуляция памяти и регистров, графика и т. д.).

Я нашел в интернете довольно хороший учебник. Однако решил, что хочу сделать задачу немного сложнее, поэтому разработал свою первую виртуальную машину CHIP-8 на новом тогда языке Swift, на котором я в то время выполнял большую часть своей профессиональной работы. Это был проект на выходные – необходимая мне отправная точка для начала разработки эмуляторов.

Практические приложения

Применение виртуальных машин при разработке программного обеспечения было широко распространено и в прошлом, и в наши дни. Их главным преимуществом является портируемость. Программа, написанная для виртуальной машины, будет работать на любой платформе, для которой существует реализация этой VM. Виртуаль-

ные машины также предоставляют инфраструктуру, которая снижает нагрузку на разработчика языка, устраняя тем самым необходимость в реализации общих функций языковой среды выполнения, таких как сборка мусора.

Одним из таких примеров может служить компиляция языка Pascal некоторыми компиляторами в 1970-х и 1980-х годах в так называемый *p-код* (p-code – разновидность байт-кода), который мог работать на p-code виртуальной машины. Двумя наиболее известными современными средами виртуальных машин являются JVM, упомянутая ранее в этой главе, и конкурирующая с ней Common Language Runtime (CLR) от Microsoft, которая является частью платформы .NET. И JVM, и CLR являются целевыми средами для нескольких популярных языков программирования. Например, C#, F# и Visual Basic – это языки, которые обычно ориентированы на CLR, но существуют также реализации для CLR таких популярных языков, как Python и Swift.

Почему эти языковые реализации компилируются в байт-код для CLR, а не в машинный код? После компиляции этот байт-код может выполняться на любой платформе, на которой установлен CLR. Это своего рода мгновенная портируемость после компиляции. Кроме того, сложная виртуальная машина, такая как CLR, предоставляет языковые службы, такие как сборка мусора, многопоточность и механизмы безопасности. Наконец, когда виртуальная машина, такая как CLR, компилирует промежуточный код в машинный код по методу «just-in-time» (JIT), она применяет оптимизации, о которых автору языка задумываться не нужно.

Помимо абстрактных машин, используемых в качестве среды выполнения языков, термин «*виртуальная машина*» также используется для обозначения всей аппаратной реализации в программном обеспечении, то есть эмулятора. Создание эмулятора является темой следующей главы.

Упражнения

1. Попробуйте измерить производительность основного кода интерпретатора операционных кодов с помощью трех различных методологий: уже реализованного оператора match, серии операторов if…elif и таблицы переходов. Определите, какой метод является самым быстрым, воспользовавшись профилировщиком или простым таймером. Для этого вам может понадобиться отключить код синхронизации в основном цикле выполнения или воспользоваться набором модульных тестов.
2. Существует немного расширенная версия CHIP-8, известная как SCHIP (Super-Chip). Она требует реализации нескольких дополнительных операционных кодов и изменения некоторых элементов исходной виртуальной машины CHIP-8, таких как разрешение

экрана. Изучите документацию по SCHIP и попробуйте превратить нашу виртуальную машину CHIP-8 в виртуальную машину SCHIP. Затем попробуйте поиграть в некоторые игры SCHIP!

3. Попробуйте написать очень простую игру, которая просто отображает пару букв на экране, с помощью инструкций машинного кода CHIP-8. Для этого вам понадобится шестнадцатеричный редактор. Приятно видеть, как написанный вами двоичный код работает в понятной вам виртуальной машине.

6

ЭМУЛЯЦИЯ ИГРОВОЙ КОНСОЛИ NES



В этой главе мы создадим ограниченный эмулятор для любимой многими игровой консоли 1980-х годов – Nintendo Entertainment System (NES). Другими словами, мы создадим программное обеспечение, которое будет имитировать аппаратную часть NES, чтобы программное обеспечение, написанное для NES, можно было «обмануть» и запустить на современной платформе. Создание этого проекта дает опыт работы с эмуляцией полноценной компьютерной системы. Хотя NES – относительно простой компьютер, он имеет все те же основные компоненты (микропроцессор, память, графику и т. д.), что и более сложные проекты эмуляции. И в отличие от CHIP-8 в главе 5, мы будем эмулировать не только спецификацию программного обеспечения – мы будем симулировать реальное аппаратное оборудование!

Наш эмулятор не будет стопроцентно точной реконструкцией оригинального оборудования. Мы сделаем несколько упрощений, чтобы создание эмулятора можно было уместить в одной главе книги. Несмотря на эти упрощения, наш эмулятор все равно сможет запускать некоторые базовые игры для NES, в том числе несколько любительских бесплатных игр с открытым исходным кодом, которые включены в репозиторий исходного кода книги. Мы не будем тестировать наш эмулятор с помощью коммерческих игр, хотя он будет способен запускать некоторые простые из них. Наша цель – изучить эмуляторы, а не добиться полной совместимости с играми. Я оставлю на усмотрение читателя дальнейшее улучшение совместимости этого эмулятора с играми.

Мы будем создавать эмулятор на чистом Python, который на момент написания этой книги не достаточно быстр на современных ПК, чтобы эмулировать NES на полной скорости. Созданный нами код можно усовершенствовать с помощью Cython, расширений C или других видов нативных кодовых слоев, чтобы он работал на полной скорости. Это тоже оставим в качестве упражнения для читателя.

Это самый сложный проект в книге. В данной главе я исхожу из того, что вы уже имеете опыт выполнения проектов из глав 1, 2 и особенно 5. Перед началом этой главы вам следует по крайней мере выполнить проект CHIP-8 из главы 5, но в этом проекте используются концепции, объясненные почти во всех предыдущих главах.

Предупреждение *Если у вас есть юридические сомнения по поводу выполнения этого проекта, изучите законы вашей страны или проконсультируйтесь с юристом. Обратите внимание, что информация, использованная для разработки эмулятора NES в данной главе, не основана на каких-либо запатентованных документах Nintendo. Имейте в виду, что большинство файлов ПЗУ для коммерческих игр защищены законом об авторском праве. Нет необходимости загружать какие-либо защищенные файлы ПЗУ для тестирования вашего эмулятора, так как репозиторий исходного кода книги включает в себя несколько некоммерческих ПЗУ, которые находятся под лицензиями с открытым исходным кодом или выпущены в общественное достояние.*

О платформе NES

NES была одной из самых продаваемых игровых консолей всех времен. Впервые выпущенная в Японии в 1983 году под названием Famicom, NES очаровала целое поколение игроков по всему миру после своего международного выпуска в 1985 году[1]. На момент дебюта NES индустрии видеоигр исполнилось всего около десяти лет. Микропроцессоры и другое оборудование в игровых консолях были еще довольно примитивными. Несмотря на это, NES обеспечивала изображение с обновлением 60 кадров в секунду, с красочными спрайтами и фоном, а также исполняла запоминающуюся музыку в стиле чиптюн (chip tune). Кроме того, она стала платформой для некоторых из наиболее культовых игр всех времен.

Примечание *Многие технические характеристики NES и большая часть информации о ее функциональности, представленной в этой главе, взяты из NesDev, сообщества разработчиков домашних игр для NES и создателей эмуляторов, которое находится по адресу https://www.nesdev.org. Хотя я считаю, что эта глава является лучшим обзором и ру-*

[1] David Sheff, Game Over: How Nintendo Conquered the World (GamePress, 1999).

ководством по написанию эмулятора NES, я настоятельно рекомендую посетить веб-сайт NesDev для получения более подробной информации. Вместо большого количества ссылок на этот сайт я представляю вам это важное примечание. Кроме того, несколько изображений в этой главе также взяты с NesDev и были опубликованы в открытом доступе, как указано на странице авторских прав данной книги. Спасибо сотрудникам NesDev за то, что они создали такой фантастический ресурс и опубликовали столько полезных справочных материалов в открытом доступе.

Аппаратная платформа

Центральный процессор (CPU) консоли NES был клоном микропроцессора MOS Technology 6502, который производился компанией Ricoh и работал на частоте чуть менее 2 МГц. Модель 6502 – это тот же микропроцессор, который использовался в популярных домашних компьютерах того времени, таких как Apple II и Commodore 64. Состоящий из примерно 3500 транзисторов, 6502 был чрезвычайно простым микропроцессором; например, в нем отсутствовали инструкции для умножения и деления[1]. Эти арифметические операции приходилось реализовывать в программном обеспечении из множества более простых инструкций, таких как сложение, вычитание и сдвиг битов. Если подумать о том, насколько медленным и простым был 6502 по сравнению с современными микропроцессорами, то просто удивительно, чего удалось добиться с его помощью.

Процессор 6502 в системе NES был объединен с аудиочипом в одном корпусе. В терминологии разработчиков NES этот чип известен как *аудиопроцессор* (audio processing unit, APU). Этот APU поддерживал пять различных звуковых каналов. Для упрощения мы не будем реализовывать APU в нашем эмуляторе. При работе со звуком очень важна синхронизация, а наш эмулятор не сможет обеспечить ее точность.

Процессор NES мог обращаться к двухкилобайтовой встроенной оперативной памяти устройства. Да, вы прочитали правильно. Рабочая память процессора NES составляла всего 2 Кбайта, чего недостаточно даже для хранения текста этого раздела главы. Это также меньше, чем объем оперативной памяти, который обычно имели системы CHIP-8, выпущенные десятилетием ранее. Некоторые картриджи содержали дополнительную оперативную память.

Ключевым элементом, обеспечивающим производительность NES, был *процессор обработки изображений* (picture processing unit, PPU), который выпускался компанией Ricoh под индексом 2C02 на основе более ранней разработки Texas Instruments. Этот PPU не только мог выводить фоновую графику в виде тайлов, но также поддерживал

[1] Russ Cox, «The MOS 6502 and the Best Layout Guy in the World», research!rsc, 3 января 2011 г., *https://research.swtch.com/6502*.

спрайты. Он имел 2 Кбайта памяти для информации о фоновых тайлах, 256 байт памяти для отслеживания до 64 спрайтов и 28 байт для хранения информации о цветовой палитре. В NES была предусмотрена поддержка 54 различных цветов, но одновременно можно было использовать только 25. В PPU даже была реализована примитивная поддержка обнаружения столкновений.

Центральный процессор взаимодействовал с APU и PPU через *отображаемые в памяти аппаратные регистры* (memory-mapped hardware registers). Это особые адреса памяти, которые при записи могут изменять работу другого аппаратного чипа, а при чтении предоставляют обновленную информацию о текущих флагах или состоянии другого чипа. Например, центральный процессор может записывать данные в регистр PPU для изменения местоположения спрайта. В дальнейшем он может считывать данные из другого регистра PPU, чтобы проверить, не столкнулся ли спрайт с чем-либо. Центральный процессор также имеет отображаемые в памяти аппаратные регистры для считывания данных с игровых контроллеров.

Позвольте мне конкретизировать концепцию отображаемого в памяти регистра на примере. Когда игре необходимо проверить состояние первого геймпада (контроллера игрока 1), она использует отображаемый в памяти регистр. Этот регистр находится по адресу памяти 0x4016. Если игра считывает данные из 0x4016, она получает 1 байт, который указывает, нажата ли определенная кнопка на геймпаде. Адрес памяти 0x4016 не может использоваться для чего-либо еще; он подключен в аппаратном обеспечении к линиям, идущим от геймпада. Для корректной работы наш эмулятор должен правильно реагировать на считывание или запись в несколько таких специальных адресов памяти. Некоторые из них доступны только для чтения, некоторые – только для записи, а некоторые – и для чтения, и для записи. Это и есть отображаемые в памяти аппаратные регистры.

Другим ключевым элементом аппаратной части NES были игровые картриджи. В основном эти картриджи, особенно ранние модели, состояли из большого чипа ПЗУ, содержащего графику и программный код игры. Игровые картриджи также могли иметь ОЗУ (иногда с батарейным питанием, чтобы можно было сохранять состояние игры), простые логические чипы и даже так называемое переключение банков (bank switching), позволяющее увеличить общий объем памяти (ОЗУ + ПЗУ) по сравнению с тем, который мог адресовать процессор 6502 в своей стандартной конфигурации. Поэтому показатель в 2 Кбайта ОЗУ немного вводит в заблуждение, поскольку код программы хранился на картридже ПЗУ, а не в ограниченной памяти игровой консоли. Вместо этого 2 Кбайта памяти могли использоваться почти исключительно для хранения состояния. Ранние игровые картриджи обычно содержали от 24 до 40 Кбайт ПЗУ, в то время как

более поздние картриджи могли располагать примерно 128 Кбайтами ПЗУ и 8 Кбайтами ОЗУ. Самый большой массово выпускавшийся картридж для NES имел 768 КБ памяти[1].

Программное обеспечение

У NES нет BIOS или операционной системы. В базовую комплектацию NES не входит программное обеспечение. Все программное обеспечение поставляется на игровых картриджах. Программы на игровых картриджах напрямую управляют CPU, PPU и APU, без каких-либо уровней абстракции между ними и аппаратной частью.

Игры для NES обычно писались на языке ассемблера процессора 6502. Это может показаться сложным, но в то время это было обычным явлением: до начала 1990-х годов большинство программ, которые должны были иметь высокую производительность на персональных компьютерах или игровых консолях, программировались на ассемблере. Помимо ассемблера, инструменты разработки часто создавались собственными силами. Не существовало NES Game Maker, который можно было бы скачать.

По сути, это была эпоха, когда загрузки еще не существовали. Консоль NES появилась за пару поколений до появления любого вида подключения к интернету. Первой массовой консолью со встроенным модемом стала Sega Dreamcast, выпущенная в конце 1990-х годов. На картридже NES была представлена окончательная версия игры; обновлений не было. Если в игре была ошибка, то она оставалась, поэтому игры должны были быть практически идеальными уже в версии 1.0. Сравните это с типичной игрой, которую вы покупаете сегодня, когда разработчики зачастую работают над первым крупным патчем еще до выпуска игры. В те времена требовалось совершенно иное внимание к деталям, но, с другой стороны, игры были гораздо менее сложными, чем сегодня.

Удивительно, что в этой примитивной среде были разработаны некоторые из самых значимых и определяющих жанр игр за всю историю. Технические навыки, необходимые программистам в командах, были совсем иными, нежели те, которые требуются современным разработчикам игр. В наши дни большинство игр создаются с использованием готовых фреймворков или движков, таких как Unreal или Unity. Разработчики могут посвящать большую часть своего времени написанию специфических для игры функций. Разработчики NES должны были писать свои собственные движки на ассемблере. Для достижения результата им приходилось напрямую управлять APU для воспроизведения каждого звука и PPU для отображения каждой

---

[1] Steven Collier, «What Was the Biggest NES Game Ever Made?», DKoldies, 24 марта 2016 г., *https://www.dkoldies.com/blog/what-wasthe-biggest-nes-game-ever-made/*.

графической детали, а также выжимать из CPU каждый цикл. Крупные компании создавали свои собственные внутренние фреймворки и инструменты, которые можно было повторно использовать в разных играх, но работа программистов по-прежнему велась на относительно низком уровне.

Создание эмулятора

Пришло время для написания кода. Однако прежде чем приступить к работе, необходимо сделать замечание относительно направления и ожиданий: эмулятор, который мы пишем, был упрощен во всех аспектах. Как упоминалось ранее, он не будет совместим со многими играми из-за очень упрощенного PPU. В нем также не будет звука, поскольку мы не реализуем APU. И он не будет работать достаточно быстро, чтобы играть в игры с их предполагаемой скоростью. Однако он будет запускать реальные игры, и в них можно будет играть. Благодаря нашей работе у вас будет прочная основа для внесения улучшений и добавления дополнительных функций, если вы того пожелаете.

Планирование структуры

Общий план реализации эмулятора не отличается от плана для виртуальной машины CHIP-8 из главы 5. Как и в случае с CHIP-8, мы будем поочередно считывать каждую инструкцию из файла ПЗУ и затем интерпретировать ее. Как и в случае с CHIP-8, мы будем использовать Pygame для отображения графики и обработки ввода пользователя. Как и в CHIP-8, у нас будет один большой цикл, который извлекает каждую инструкцию и реагирует на каждое событие. Однако структура кода будет более сложной. В частности, мы разделим эмулятор на три класса, каждый из которых будет представлять один физический компонент аппаратной части. У нас будут классы для CPU, PPU и картриджа. Ниже приводится описание каждого файла, который мы будем писать, и его назначение:

__main__.py	Обрабатывает аргументы командной строки и реализует основной цикл эмулятора, который распределяет инструкции, отображает графику и реагирует на ввод пользователя.
rom.py	Считывает файл ПЗУ и имитирует картридж.
cpu.py	Поддерживает состояние центрального процессора, интерпретирует инструкции и обрабатывает доступ к основной памяти.
ppu.py	Управляет состоянием PPU, рисует фоны и спрайты.

Мы будем рассматривать эти файлы в приведенном здесь порядке.

Создание главного цикла

В нашем главном файле (*__main__.py*) объединены различные компоненты системы (CPU, PPU, картридж). Его «цикл выполнения» оживляет эмулятор, обеспечивая непрерывное перемещение и координацию между различными компонентами, делегируя Pygame отображение графики и чтение ввода пользователя по мере необходимости. Функция `run()` принимает в качестве аргументов объект ПЗУ и имя файла ПЗУ. В нашем первом фрагменте мы инициализируем Pygame, получаем окно на экране и создаем объекты CPU и PPU:

NESEmulator/__main__.py

```
import sys
from argparse import ArgumentParser
from NESEmulator.rom import ROM
from NESEmulator.ppu import PPU, NES_WIDTH, NES_HEIGHT
from NESEmulator.cpu import CPU
import pygame
from timeit import default_timer as timer
import os

def run(rom: ROM, name: str):
    pygame.init()
    screen = pygame.display.set_mode((NES_WIDTH, NES_HEIGHT), 0, 24)
    pygame.display.set_caption(f"NES Emulator - {os.path.basename(name)}")
    ppu = PPU(rom)
    cpu = CPU(ppu, rom)
    ticks = 0
    start = None
```

Как показывают вызовы их конструкторов, как PPU, так и CPU нуждаются в доступе к ПЗУ. Центральному процессору необходимо считывать программные инструкции, а PPU – графические данные. Центральному процессору также необходим доступ к PPU, поскольку при чтении или записи определенных адресов памяти они на самом деле являются прокси для регистров PPU.

Переменная `ticks` отслеживает, сколько рабочих тактов выполнил центральный процессор. На каждый рабочий такт центрального процессора приходится ровно три рабочих такта графического процессора. Другими словами, графический процессор работает в три раза быстрее, чем центральный процессор, поэтому если частота центрального процессора составляет около 1,8 МГц, то частота графического процессора равна примерно 5,4 МГц. Наш код должен будет имитировать это, поэтому в следующем фрагменте, который является основным игровым циклом, мы отслеживаем, сколько циклов (или тиков) занимает каждая инструкция CPU (разные инструкции требуют разного количества циклов), а затем запускаем PPU для трехкратного увеличения этого количества циклов:

```
while True:
cpu.step()
new_ticks = cpu.cpu_ticks - ticks
# 3 PPU cycles for every CPU tick
for _ in range(new_ticks * 3):
    ppu.step()
    # Рисование, по одному фрейму, всех объектов на экране
    if (ppu.scanline == 240) and (ppu.cycle == 257): ❶
        pygame.surfarray.blit_array(screen, ppu.display_buffer)
        pygame.display.flip()
        end = timer()
        if start is not None:
            print(end - start)
        start = timer()
    if (ppu.scanline == 241) and (ppu.cycle == 2) and ppu.generate_nmi:
        cpu.trigger_NMI() ❷
ticks += new_ticks
```

По завершении каждого фрейма видимая пользователю графика обновляется с помощью тех же методов Pygame, которые мы использовали в главе 5. Но как мы узнаем, что фрейм закончился? Разрешение NES составляло 256 пикселей по ширине и 240 пикселей по высоте. Каждая строка пикселей называется строкой развертки – этот термин происходит от телевизоров с кинескопами – электронно-лучевыми трубками (cathode ray tube, CRT), к которым подключалась NES. На настоящей NES один пиксель обновлялся с каждым циклом PPU. Поскольку PPU работает в три раза быстрее центрального процессора (5,4 МГц против 1,8 МГц), за каждый цикл центрального процессора PPU рисует три точки. Когда мы доходим до 257-й точки на 240-й строке развертки, мы должны закончить один фрейм (кадр) ❶.

Высокоточные эмуляторы NES имитируют поведение реального оборудования, выполняя то, что PPU и должен делать в каждом цикле: определять цвет следующей точки. Мы используем гораздо более простую технику, просто рисуя все правильные тайлы и спрайты в нужных местах один раз за фрейм. Другими словами, вместо того чтобы думать об одной точке в каждом цикле, мы просто думаем о том, как должен выглядеть весь экран один раз за фрейм. Хотя этот метод быстрее и не требует от нас эмуляции множества параметров внутренней работы PPU, он подходит не для всех игр. Более сложные игры для NES вносят изменения в графику даже во время рисования фрейма на экране (то есть в промежутках между строками развертки или даже между точками).

Обратите внимание, что PPU фактически выполняет дополнительную обработку между строками развертки. Этот период называется *hblank*. Что еще более важно, PPU имеет дополнительные внеэкранные строки развертки (они идут вплоть до строки развертки 261, в то время как первая считается строкой развертки 0). Время обработки

этих дополнительных непрорисованных строк развертки называется *vblank*. В период vblank процессор может безопасно изменять любую память PPU, поскольку память PPU не используется для рендеринга видимых строк развертки. Для этой цели PPU посылает сигнал процессору о начале vblank (после завершения обработки всех видимых строк развертки) ❷. Этот сигнал является разновидностью *немаскируемого прерывания* (non-maskable interrupt, NMI).

Представьте себе NMI как прерывание программы, которое нельзя остановить. Другими словами, это сигнал, который говорит микропроцессору: «Немедленно прекрати то, что ты делаешь, потому что сейчас мы делаем другое». В случае с NES «другим делом» является обновление графического отображения игры. Каждая игра NES имеет обработчик NMI, который обновляет графику игры в течение vblank.

Остальная часть цикла просто обрабатывает события:

```
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()
        # Обработка событий клавиатуры как событий геймпада
        if event.type not in {pygame.KEYDOWN, pygame.KEYUP}:
            continue
        is_keydown = event.type == pygame.KEYDOWN
        match event.key:
            case pygame.K_LEFT:
                cpu.joypad1.left = is_keydown
            case pygame.K_RIGHT:
                cpu.joypad1.right = is_keydown
            case pygame.K_UP:
                cpu.joypad1.up = is_keydown
            case pygame.K_DOWN:
                cpu.joypad1.down = is_keydown
            case pygame.K_x:
                cpu.joypad1.a = is_keydown
            case pygame.K_z:
                cpu.joypad1.b = is_keydown
            case pygame.K_s:
                cpu.joypad1.start = is_keydown
            case pygame.K_a:
                cpu.joypad1.select = is_keydown

if __name__ == "__main__":
    # Парсинг аргумента файла
    file_parser = ArgumentParser("NESEmulator")
    file_parser.add_argument("rom_file",
                             help="An NES game file in iNES format.")
    arguments = file_parser.parse_args()
    game = ROM(arguments.rom_file)
    run(game, arguments.rom_file)
```

Мы распознаем нажатие определенных клавиш в качестве эквивалента кнопок на геймпаде NES. Мы отмечаем нажатые кнопки, чтобы процессор мог их прочитать. Наш основной файл заканчивается обработкой аргумента командной строки для чтения файла ПЗУ. Фактическое чтение выполняется классом `ROM`, о котором будет рассказано далее.

Эмуляция картриджа

Игровые картриджи NES в основном состояли из микросхем ПЗУ в пластиковом корпусе. Эти микросхемы ПЗУ содержали код игры и графические ресурсы. Код находился в микросхеме ПЗУ, известной как PRG ROM, а графика – в микросхеме ПЗУ, известной как CHR ROM. Хотя картриджи в основном состояли из микросхем ПЗУ, они могли содержать и многое другое. Как уже упоминалось ранее, одним из наиболее распространенных дополнений были логические чипы, которые обеспечивали *переключение банков* (bank switching), то есть давали возможность использовать больше ROM, чем обычно могла обрабатывать NES, но переключаться по схеме, чтобы программа могла получить доступ к определенному адресу, отображенному в памяти, и сказать: «Я закончил с первыми 8 Кбайтами CHR ROM, пожалуйста, переключите чтение памяти на следующие 8 Кбайт».

Некоторые картриджи ушли еще дальше, предоставляя дополнительную оперативную память вдобавок к скудным 2 Кбайтам основного процессора. Такая память называется PRG RAM. Некоторые картриджи даже имели элементы питания, чтобы содержимое оперативной памяти не стиралось при выключении консоли. Другого способа постоянного хранения пользовательских данных на NES не было, поскольку отсутствовал диск. Оперативная память с элементами питания позволяла играть в более продолжительные игры. Никто не захочет играть в 40-часовую игру, если его достижения будут стерты после выключения консоли.

Чем дольше NES находилась на рынке, тем более совершенными становились картриджи. Один и тот же улучшенный тип картриджа мог использоваться для множества игр. Для поддержки всех игр разработчику эмулятора нужна поддержка всех разнообразных чипсетов, которые могли использоваться в картриджах. Однако было несколько особенно популярных типов чипсетов, которые использовались в большинстве игр.

Каждый из таких чипсетов картриджа получил в мире эмуляторов NES название «*распределитель*», или просто «*маппер*» (mapper), поскольку основное назначение чипсетов картриджа заключалось в переключении между различными банками памяти. В сфере программирования этот механизм можно представить в виде сопоставления адреса с банком. Один из первых эмуляторов NES – iNES, разработан-

ный Маратом Файзуллиным (Marat Fayzullin)[1], определил схему нумерации для множества вариантов таких распределителей-мапперов. Кроме того, iNES определил формат файлов ПЗУ, который сегодня используется почти всеми эмуляторами NES.

В отличие от CHIP-8, файлы ПЗУ которого состоят только из необработанной памяти игры, для NES требуется более сложный формат файлов из-за разнообразия игровых картриджей. В частности, формат iNES определяет заголовок, на который мы должны обратить внимание при чтении файла ПЗУ. К счастью, у нас есть такой опыт благодаря нашей работе с заголовком MacBinary в главе 3. Заголовок формата файла iNES определен в табл. 6.1[2].

Таблица 6.1 Заголовок формата файла iNES

Байты	Описание
0–3	Константа 0x4E45531A (ASCII «NES», за которым следует конец файла MS-DOS)
4	Размер PRG ROM в единицах 16 Кбайт
5	Размер CHR ROM в единицах 8 Кбайт (значение 0 означает, что плата использует CHR RAM)
6	Флаги 6: маппер, зеркалирование, батарея, trainer
7	Флаги 7: маппер, VS/Playchoice, NES 2.0
8	Флаги 8: размер PRG RAM (редко используемое расширение)
9	Флаги 9: ТВ-система (редко используемое расширение)
10	Флаги 10: ТВ-система, наличие PRG RAM (неофициальное, редко используемое расширение)
11–15	Неиспользуемая заполняющая информация (должна быть заполнена нулями, но некоторые пираты помещают здесь свои имена)

Как вы можете видеть, часть заголовка определяет номер маппера игры. Каждый байт флагов может содержать несколько отдельных битов флагов, поэтому в некоторых описаниях перечислено несколько элементов. Наш эмулятор не будет использовать информацию, содержащуюся во флагах 8, 9 или 10.

Примечание *Формат файлов iNES был расширен новым форматом, известным как NES 2.0. Большая часть полей заголовка iNES по-прежнему действительна в NES 2.0, с дополнительной информацией в байтах с 11 по 15 и некоторыми изменениями в байтах флагов.*

Эмулятор, который мы создадим в этой главе, способен воспроизводить только игры, использующие самый простой маппер – mapper 0, известный как NROM. Картриджи NROM не имеют переключения банков, поэтому их проще всего эмулировать. Картридж NROM всегда имеет 16 Кбайт или 32 Кбайта PRG ROM и 8 Кбайт CHR ROM. Он может опционально иметь PRG RAM.

---

[1] Marat Fayzullin, «iNES», доступ 19 апреля 2024 г., *http://fms.komkon.org/iNES*.

[2] Таблица 6.1 основана на информации с сайта NesDev.org, опубликованной в открытом доступе. См. *https://www.nesdev.org/wiki/INES*.

В нашем коде мы определяем кортеж `namedtuple` под названием `Header` для хранения содержимого заголовка файла ROM, а также объявляем некоторые стандартные константы:

NESEmulator/rom.py

```
from pathlib import Path
from struct import unpack
from collections import namedtuple
from array import array

Header = namedtuple("Header", "signature prg_rom_size chr_rom_size "
                              "flags6 flags7 flags8 flags9 flags10 unused")
HEADER_SIZE = 16
TRAINER_SIZE = 512
PRG_ROM_BASE_UNIT_SIZE = 16384
CHR_ROM_BASE_UNIT_SIZE = 8192
PRG_RAM_SIZE = 8192
```

Конструктор класса `ROM` сначала использует функцию `unpack()` из модуля стандартной библиотеки `struct` для чтения заголовка из файла ПЗУ и распределения его по полям соответствующего размера, заданного строкой формата:

```
class ROM:
    def __init__(self, file_name: str | Path):
    with open(file_name, "rb") as file:
# Считывание заголовка и проверка сигнатуры "NES"
        self.header = Header._make(unpack("!LBBBBBB5s",
                                          file.read(HEADER_SIZE)))
```

Метод класса `._make()` для `namedtuple` можно использовать для создания экземпляра этого `namedtuple` из итерабельного объекта, такого как полученный из `unpack()`. Конкретные типы элементов в строке формата `unpack()` перечислены в табл. 6.2. Каждый элемент в строке формата соответствует элементу заголовка из табл. 6.1. Более подробную информацию о строке формата см. в документации по модулю `struct` по адресу *https://docs.python.org/3/library/struct.html#struct-format-strings*.

Таблица 6.2 Символы строки формата для модуля struct

Элемент	Количество байтов	Тип C	Тип Python
!	Не определено	Указывает, что следующее значение имеет формат big-endian	Не определено
L	4	`unsigned long`	`int`
B	1	`unsigned char`	`int`
5s	5	`char[]`	Байты

После этой перестановки `self.header` содержит правильные части заголовка iNES в четко обозначенных сегментах. Далее мы проверяем несколько элементов информации из заголовка:

```
if self.header.signature != 0x4E45531A:
    print("Invalid ROM Header Signature")
else:
    print("Valid ROM Header Signature")
# Распутываем Mapper - один ниббл в flags6 и один ниббл в flags7
self.mapper = (self.header.flags7 & 0xF0) | (
        (self.header.flags6 & 0xF0) >> 4)
print(f"Mapper {self.mapper}")
if self.mapper != 0:
    print("Invalid Mapper: Only Mapper 0 is Implemented")
```

Каждый заголовок iNES должен начинаться с одинаковой 4-байтовой сигнатуры. Между тем номер маппера строится из части флагов 7 и 8. Наш эмулятор работает только с играми, которые используют mapper 0.

Вот остальная часть конструктора:

```
self.read_cartridge = self.read_mapper0
self.write_cartridge = self.write_mapper0
# Проверка наличия trainer (4-й бит flags6) и его чтение
self.has_trainer = bool(self.header.flags6 & 4)
if self.has_trainer:
    self.trainer_data = file.read(TRAINER_SIZE)
# Проверка зеркалирования по flags6 бит 0
self.vertical_mirroring = bool(self.header.flags6 & 1)
print(f"Has vertical mirroring {self.vertical_mirroring}")
# Считывание PRG_ROM & CHR_ROM, кратных 16K и 8K соответственно
self.prg_rom = file.read(PRG_ROM_BASE_UNIT_SIZE *
                         self.header.prg_rom_size)
self.chr_rom = file.read(CHR_ROM_BASE_UNIT_SIZE *
                         self.header.chr_rom_size)
self.prg_ram = array('B', [0] * PRG_RAM_SIZE) # RAM
```

Этот код отвечает за настройку других свойств игрового картриджа. Как мы читаем и записываем из него (это может отличаться в зависимости от маппера, хотя мы поддерживаем только mapper 0)? Есть ли у него trainer (специфическая функция, которую мы проигнорируем)? Использует ли он определенный тип зеркалирования для графики? Наконец, на основе размеров, указанных в заголовке, считывается соответствующий объем данных для PRG ROM, CHR ROM и (опционально) PRG RAM.

Обратите внимание, как `read_cartridge` и `write_cartridge` назначаются в качестве псевдонимов для методов `read_mapper0()` и `write_`

mapper0(). Если бы мы поддерживали более одного маппера, мы бы поступили по-другому. В данном случае это определения для методов mapper 0:

```
    def read_mapper0(self, address: int) -> int:
        if address < 0x2000:
            return self.chr_rom[address]
        elif 0x6000 <= address < 0x8000:
            return self.prg_ram[address % PRG_RAM_SIZE]
        elif address >= 0x8000:
            if self.header.prg_rom_size > 1:
                return self.prg_rom[address - 0x8000]
            else:
                return self.prg_rom[(address - 0x8000) % PRG_ROM_BASE_UNIT_SIZE]
        else:
            raise LookupError(f"Tried to read at invalid address {address:X}")

    def write_mapper0(self, address: int, value: int):
        if address >= 0x6000:
            self.prg_ram[address % PRG_RAM_SIZE] = value
```

Если посмотреть на read_mapper0(), можно заметить три отдельные области памяти на картридже. Адреса ниже 0x2000 сопоставляются с CHR ROM, к которому PPU имеет прямой доступ. Центральный процессор обращается к PRG ROM с адресами, большими или равными 0x8000, и может читать или записывать в PRG RAM (если она есть в картридже) с адресами, большими или равными 0x6000, но меньшими 0x8000.

На этом часть нашего кода, посвященная картриджам, завершена. Говоря кратко, файл ROM преобразуется в области CHR ROM и PRG ROM, к которым могут обращаться соответственно наш PPU и CPU. Именно поэтому классы PPU и CPU в нашем эмуляторе должны иметь доступ к классу ROM.

Эмуляция центрального процессора

Центральный процессор можно рассматривать как сложную конечную машину состояний. Он поддерживает состояние в своих регистрах и в конечной памяти, к которой имеет доступ. Переходы между состояниями осуществляются с помощью инструкций, которые он может обрабатывать. Это понимание объясняет основную работу, которую должен выполнять наш эмулятор CPU: поддерживать регистры, получать доступ к памяти и правильно изменять регистры и память в соответствии с инструкциями.

Модель 6502 является одним из самых простых процессорных ядер, которые когда-либо получили широкое признание в отрасли, а версия 6502 в NES еще проще, чем стандартная 6502. В ней отсутствуют

инструкции для BCD, которые были в большинстве моделей 6502. Для построения рабочего процессора NES нам нужно реализовать всего 56 различных типов инструкций, и многие из них можно реализовать всего парой строк кода. Кроме того, 6502 имеет всего три основных регистра (A, X и Y) и несколько специализированных регистров (SP, PC и различные флаги). Единственная реальная сложность при работе с 6502 заключается в нескольких различных методах доступа к памяти, которые могут использовать различные инструкции, но мы абстрагируем их во вспомогательной функции.

Настройка

Код для нашей реализации 6502 начинается с настройки некоторых вспомогательных конструкций и констант:

NESEmulator/cpu.py

```
from __future__ import annotations
from enum import Enum
from dataclasses import dataclass
from array import array
from typing import Callable
from NESEmulator.ppu import PPU, SPR_RAM_SIZE
from NESEmulator.rom import ROM

MemMode = Enum("MemMode", "DUMMY ABSOLUTE ABSOLUTE_X ABSOLUTE_Y ACCUMULATOR "
                          "IMMEDIATE IMPLIED INDEXED_INDIRECT INDIRECT "
                          "INDIRECT_INDEXED RELATIVE ZEROPAGE ZEROPAGE_X "
                          "ZEROPAGE_Y")

InstructionType = Enum("InstructionType", "ADC AHX ALR ANC AND ARR ASL AXS "
                                          "BCC BCS BEQ BIT BMI BNE BPL BRK "
                                          "BVC BVS CLC CLD CLI CLV CMP CPX "
                                          "CPY DCP DEC DEX DEY EOR INC INX "
                                          "INY ISC JMP JSR KIL LAS LAX LDA "
                                          "LDX LDY LSR NOP ORA PHA PHP PLA "
                                          "PLP RLA ROL ROR RRA RTI RTS SAX "
                                          "SBC SEC SED SEI SHX SHY SLO SRE "
                                          "STA STX STY TAS TAX TAY TSX TXA "
                                          "TXS TYA XAA")

@dataclass(frozen=True)
class Instruction:
    type: InstructionType
    method: Callable[[Instruction, int], None]
    mode: MemMode
    length: int
    ticks: int
    page_ticks: int

@dataclass
class Joypad:
```

```
    strobe: bool = False
    read_count: int = 0
    a: bool = False
    b: bool = False
    select: bool = False
    start: bool = False
    up: bool = False
    down: bool = False
    left: bool = False
    right: bool = False

STACK_POINTER_RESET = 0xFD
STACK_START = 0x100
RESET_VECTOR = 0xFFFC
NMI_VECTOR = 0xFFFA
IRQ_BRK_VECTOR = 0xFFFE
MEM_SIZE = 2048
```

Перечисление `MemMode` содержит все различные схемы доступа к памяти в 6502. В некоторых простых микропроцессорах извлечение байта из памяти сводится к указанию адреса и получению хранящегося там байта. Например, если я попрошу «прочитать 0x1940», я получу любой байт, хранящийся в памяти по адресу 0x1940. Модель 6502 может делать это в режиме `ABSOLUTE`, но у нее есть и другие режимы памяти, которые полезны в определенных ситуациях. Некоторые из этих режимов обращаются к адресам памяти, которые рассчитываются на лету, а не указываются буквально. Например, режим `ABSOLUTE_X` добавляет значение в регистре `X` к указанному адресу и обращается к полученному месту в памяти. В этом режиме, если бы мы снова выполнили ту же инструкцию после инкремента `X`, мы бы автоматически прочитали следующий байт в памяти. При программировании на низком уровне это может быть очень удобно и даже повышает производительность, если аппаратное обеспечение оптимизировано для определенных режимов доступа. Более подробно режимы памяти NES мы обсудим позже в этой главе – к счастью, многие из них очень похожи друг на друга.

Перечисление `InstructionType` содержит все различные типы инструкций, которые может обрабатывать 6502. Некоторые из этих типов инструкций предназначены для операций BCD, которых, как упоминалось ранее, в версии 6502 для NES не было. Некоторые из них являются «неофициальными» типами инструкций, которые фактически не были задокументированы как часть 6502, но были обнаружены методом проб и ошибок. Лишь очень немногие игры используют их. Остальные 56 – это типы инструкций, которые мы действительно будем использовать. Мы указываем все возможные типы инструкций в этом перечислении, даже нереализованные, потому что будем использовать автоматически сгенерированную таблицу всех 256 возможных кодов операций 6502 и хотим, чтобы каждая запись в таблице имела действительное значение типа инструкции.

Команда `Instruction` относится к одному из 256 возможных операционных кодов, которые может понимать 6502. Каждая инструкция содержит информацию о своем типе (`type`), связанной с ней функции в нашей программе, которая ее обрабатывает (`method`), режиме доступа к памяти (`mode`), ожидаемом количестве байтов (`length`), количестве циклов центрального процессора, необходимых для ее выполнения (`ticks`), и о дополнительном количестве циклов, необходимых для ее выполнения, если в процессе выполнения происходит переход на другую страницу памяти (`page_ticks`). *Страница памяти* (memory page) – это часть оперативной памяти, к любой части которой контроллер памяти может получить доступ в быстрой последовательности по отношению к другой части. В 6502 страницы памяти имеют размер 256 байт. Если инструкция пересекает эти 256-байтовые границы, ее выполнение может занять больше времени.

Тот факт, что каждая инструкция имеет связанную с ней функцию для ее обработки, является подсказкой, что мы будем использовать совершенно другой принцип проектирования, нежели в проекте CHIP-8. Для виртуальной машины CHIP-8 мы использовали гигантское выражение `match` для обработки каждой инструкции, но для 6502 и его немного более сложного набора инструкций мы будем использовать более чистый принцип разработки. Вместо того чтобы переключаться на каждую инструкцию, мы будем искать ее в массиве по ее коду операции, а затем выполнять связанную с ней функцию. Такой дизайн является вариацией распространенного шаблона, известного как *таблица переходов* (jump table). По сути, мы индексируем массив инструкций по коду операции, чтобы найти функцию, к которой нужно перейти.

Класс `Joypad` представляет состояние геймпада (игрового манипулятора) во время выполнения программы. Центральный процессор может напрямую опрашивать геймпад через пару отображаемых в памяти регистров, поэтому было решено поместить `Joypad` в модуль `cpu`. Напомним, что наш главный цикл устанавливает состояние геймпада на основе событий, обнаруженных Pygame.

Ниже приводится разбивка остальных вспомогательных констант из предыдущего листинга кода:

`STACK_POINTER_RESET` Адрес памяти, на который изначально указывает указатель стека процессора.

`STACK_START` Место начала стека в памяти, которое, что интересно, отличается от адреса `STACK_POINTER_RESET`.

`RESET_VECTOR` Адрес в памяти, содержащий другой адрес в памяти, с которого начинается выполнение программы. PRG ROM каждой игры NES имеет какой-то стартовый код, чтобы запустить процесс по адресу, указанному в `RESET_VECTOR`.

NMI_VECTOR	То же самое, что и RESET_VECTOR, но для NMI и vblank. Когда происходит событие vblank, управление переходит к адресу в памяти, указанному в NMI_VECTOR.
IRQ_BRK_VECTOR	Адрес для менее распространенного типа прерывания, который не будет учитываться в тестируемых нами играх. Он включен сюда для полноты картины.
MEM_SIZE	Размер в байтах основной оперативной памяти, к которой имеет доступ процессор NES.

Далее давайте рассмотрим начало конструктора нашего класса CPU, который настраивает его память, регистры и конфигурируемые переменные состояния:

```
class CPU:
    def __init__(self, ppu: PPU, rom: ROM):
    # Подключения к другим модулям консоли
    self.ppu: PPU = ppu
    self.rom: ROM = rom
    # Память CPU
    self.ram = array('B', [0] * MEM_SIZE)
    # Регистры
    self.A: int = 0
    self.X: int = 0
    self.Y: int = 0
    self.SP: int = STACK_POINTER_RESET
    self.PC: int = self.read_memory(RESET_VECTOR, MemMode.ABSOLUTE) | \
                   (self.read_memory(RESET_VECTOR + 1,
                                     MemMode.ABSOLUTE) << 8)
    # Флаги
    self.C: bool = False # Carry
    self.Z: bool = False # Zero
    self.I: bool = True # Interrupt disable
    self.D: bool = False # Decimal mode
    self.B: bool = False # Break command
    self.V: bool = False # oVerflow
    self.N: bool = False # Negative
    # Различные состояния
    self.jumped: bool = False
    self.page_crossed: bool = False
    self.cpu_ticks: int = 0
    self.stall: int = 0 # количество циклов до остановки
    self.joypad1 = Joypad()
```

Для более глубокого понимания этого кода настройки давайте подробно рассмотрим регистры 6502. Все они перечислены в табл. 6.3.

Таблица 6.3 Регистры 6502

Название	Размер (в байтах)	Назначение
A	1	1 Основной регистр, используемый для арифметических операций. Иногда называется *аккумулятором* (accumulator)
X	1	*Регистр индекса* (index register), зачастую используемый в качестве счетчика циклов. Он также может использоваться в качестве регистра общего назначения, хотя не все инструкции работают с ним так же, как с A
Y	1	То же, что и X
PC	2	*Счетчик программы* (program counter), который отслеживает, где в памяти находится следующая инструкция для выполнения. Он имеет размер 2 байта, потому что 6502 может адресовать до 64 Кбайт памяти (без переключения банков)
SP	1	*Указатель стека* (stack pointer), который отслеживает, где в стеке находится программа в данный момент. Поскольку он занимает только 1 байт, стек может содержать максимум 256 байт
P	1	Регистр *состояния* (status) или *флагов* (flags). Его отдельные биты обозначают разные опции, например что-то об арифметической операции (например, результат равен нулю?) или произошел ли перерыв либо включено прерывание (IRQ)

Поскольку в Python существует только один тип для всех целых чисел вне зависимости от размера, мы представляем все эти регистры, за исключением флагов, с помощью типа `int`. Вместо необходимости разбираться с отдельными битами для каждого из флагов в регистре состояния мы разделяем их на отдельные булевы величины с использованием буквенной номенклатуры, принятой в документации по 6502. Это переменные-члены `C`, `Z`, `I`, `D`, `B`, `V` и `N`. Большинство из них устанавливаются в результате арифметических операций, `I` устанавливается, когда программа не хочет прерываться сигналом IRQ, а `B` устанавливается, когда флаги помещаются в стек после команды break. Флаг `D`, используемый для кода BCD, не имеет отношения к NES, поскольку NES не имеет команд BCD.

Переменная `jumped` отслеживает, изменила ли команда перехода регистр `PC`, а `page_crossed` используется для учета при доступе к памяти через страницу памяти, что, как уже упоминалось, может быть более затратно, нежели доступ к ближайшей памяти. Процессор NES может потребовать определенного количества циклов для завершения некоторых задач. Для этого и нужен флаг `stall`. В нашем эмуляторе он используется только при прямом доступе к памяти (direct memory access, DMA) для отправки пакета данных из основной памяти в память атрибутов объектов (object attribute memory, OAM), где PPU хранит информацию о спрайтах.

Таблица переходов

Далее мы объявим таблицу переходов – список всех потенциально возможных инструкций, которые способен обрабатывать процессор 6502. Поскольку 6502 использует 1-байтовые коды операций, а для байта существует 256 возможных значений, потенциально существует 256 различных инструкций. Позже, в нашем методе `step()`, мы

будем индексировать этот список для получения конкретной инструкции и соответствующей ей функции для данного декодированного нами операционного кода. Мы не будем реализовывать каждую инструкцию (некоторые из них являются BCD или неофициальными), поэтому некоторые из них привязаны к методу `self.unimplemented()`.

Все 256 строк таблицы переходов включены сюда для завершенности:

```
self.instructions = [
    Instruction(InstructionType.BRK, self.BRK, MemMode.IMPLIED, 1, 7, 0), # 00
    Instruction(InstructionType.ORA, self.ORA, MemMode.INDEXED_INDIRECT, 2, 6, 0),
    Instruction(InstructionType.KIL, self.unimplemented, MemMode.IMPLIED, 0, 2, 0),
    Instruction(InstructionType.SLO, self.unimplemented, MemMode.INDEXED_INDIRECT, 0, 8, 0),
    Instruction(InstructionType.NOP, self.NOP, MemMode.ZEROPAGE, 2, 3, 0), # 04
    Instruction(InstructionType.ORA, self.ORA, MemMode.ZEROPAGE, 2, 3, 0), # 05
    Instruction(InstructionType.ASL, self.ASL, MemMode.ZEROPAGE, 2, 5, 0), # 06
    Instruction(InstructionType.SLO, self.unimplemented, MemMode.ZEROPAGE, 0, 5, 0),
    Instruction(InstructionType.PHP, self.PHP, MemMode.IMPLIED, 1, 3, 0), # 08
    Instruction(InstructionType.ORA, self.ORA, MemMode.IMMEDIATE, 2, 2, 0), # 09
    Instruction(InstructionType.ASL, self.ASL, MemMode.ACCUMULATOR, 1, 2, 0), # 0a
    Instruction(InstructionType.ANC, self.unimplemented, MemMode.IMMEDIATE, 0, 2, 0),
    Instruction(InstructionType.NOP, self.NOP, MemMode.ABSOLUTE, 3, 4, 0), # 0c
    Instruction(InstructionType.ORA, self.ORA, MemMode.ABSOLUTE, 3, 4, 0), # 0d
    Instruction(InstructionType.ASL, self.ASL, MemMode.ABSOLUTE, 3, 6, 0), # 0e
    Instruction(InstructionType.SLO, self.unimplemented, MemMode.ABSOLUTE, 0, 6, 0),
    Instruction(InstructionType.BPL, self.BPL, MemMode.RELATIVE, 2, 2, 1), # 10
    Instruction(InstructionType.ORA, self.ORA, MemMode.INDIRECT_INDEXED, 2, 5, 1),
    Instruction(InstructionType.KIL, self.unimplemented, MemMode.IMPLIED, 0, 2, 0),
    Instruction(InstructionType.SLO, self.unimplemented, MemMode.INDIRECT_INDEXED, 0, 8, 0),
    Instruction(InstructionType.NOP, self.NOP, MemMode.ZEROPAGE_X, 2, 4, 0), # 14
    Instruction(InstructionType.ORA, self.ORA, MemMode.ZEROPAGE_X, 2, 4, 0), # 15
    Instruction(InstructionType.ASL, self.ASL, MemMode.ZEROPAGE_X, 2, 6, 0), # 16
    Instruction(InstructionType.SLO, self.unimplemented, MemMode.ZEROPAGE_X, 0, 6, 0),
    Instruction(InstructionType.CLC, self.CLC, MemMode.IMPLIED, 1, 2, 0), # 18
    Instruction(InstructionType.ORA, self.ORA, MemMode.ABSOLUTE_Y, 3, 4, 1), # 19
    Instruction(InstructionType.NOP, self.NOP, MemMode.IMPLIED, 1, 2, 0), # 1a
    Instruction(InstructionType.SLO, self.unimplemented, MemMode.ABSOLUTE_Y, 0, 7, 0),
    Instruction(InstructionType.NOP, self.NOP, MemMode.ABSOLUTE_X, 3, 4, 1), # 1c
    Instruction(InstructionType.ORA, self.ORA, MemMode.ABSOLUTE_X, 3, 4, 1), # 1d
    Instruction(InstructionType.ASL, self.ASL, MemMode.ABSOLUTE_X, 3, 7, 0), # 1e
    Instruction(InstructionType.SLO, self.unimplemented, MemMode.ABSOLUTE_X, 0, 7, 0),
    Instruction(InstructionType.JSR, self.JSR, MemMode.ABSOLUTE, 3, 6, 0), # 20
    Instruction(InstructionType.AND, self.AND, MemMode.INDEXED_INDIRECT, 2, 6, 0),
    Instruction(InstructionType.KIL, self.unimplemented, MemMode.IMPLIED, 0, 2, 0),
    Instruction(InstructionType.RLA, self.unimplemented, MemMode.INDEXED_INDIRECT, 0, 8, 0),
    Instruction(InstructionType.BIT, self.BIT, MemMode.ZEROPAGE, 2, 3, 0), # 24
    Instruction(InstructionType.AND, self.AND, MemMode.ZEROPAGE, 2, 3, 0), # 25
    Instruction(InstructionType.ROL, self.ROL, MemMode.ZEROPAGE, 2, 5, 0), # 26
    Instruction(InstructionType.RLA, self.unimplemented, MemMode.ZEROPAGE, 0, 5, 0),
    Instruction(InstructionType.PLP, self.PLP, MemMode.IMPLIED, 1, 4, 0), # 28
```

```
Instruction(InstructionType.AND, self.AND, MemMode.IMMEDIATE, 2, 2, 0), # 29
Instruction(InstructionType.ROL, self.ROL, MemMode.ACCUMULATOR, 1, 2, 0), # 2a
Instruction(InstructionType.ANC, self.unimplemented, MemMode.IMMEDIATE, 0, 2, 0),
Instruction(InstructionType.BIT, self.BIT, MemMode.ABSOLUTE, 3, 4, 0), # 2c
Instruction(InstructionType.AND, self.AND, MemMode.ABSOLUTE, 3, 4, 0), # 2d
Instruction(InstructionType.ROL, self.ROL, MemMode.ABSOLUTE, 3, 6, 0), # 2e
Instruction(InstructionType.RLA, self.unimplemented, MemMode.ABSOLUTE, 0, 6, 0),
Instruction(InstructionType.BMI, self.BMI, MemMode.RELATIVE, 2, 2, 1), # 30
Instruction(InstructionType.AND, self.AND, MemMode.INDIRECT_INDEXED, 2, 5, 1),
Instruction(InstructionType.KIL, self.unimplemented, MemMode.IMPLIED, 0, 2, 0),
Instruction(InstructionType.RLA, self.unimplemented, MemMode.INDIRECT_INDEXED, 0, 8, 0),
Instruction(InstructionType.NOP, self.NOP, MemMode.ZEROPAGE_X, 2, 4, 0), # 34
Instruction(InstructionType.AND, self.AND, MemMode.ZEROPAGE_X, 2, 4, 0), # 35
Instruction(InstructionType.ROL, self.ROL, MemMode.ZEROPAGE_X, 2, 6, 0), # 36
Instruction(InstructionType.RLA, self.unimplemented, MemMode.ZEROPAGE_X, 0, 6, 0),
Instruction(InstructionType.SEC, self.SEC, MemMode.IMPLIED, 1, 2, 0), # 38
Instruction(InstructionType.AND, self.AND, MemMode.ABSOLUTE_Y, 3, 4, 1), # 39
Instruction(InstructionType.NOP, self.NOP, MemMode.IMPLIED, 1, 2, 0), # 3a
Instruction(InstructionType.RLA, self.unimplemented, MemMode.ABSOLUTE_Y, 0, 7, 0),
Instruction(InstructionType.NOP, self.NOP, MemMode.ABSOLUTE_X, 3, 4, 1), # 3c
Instruction(InstructionType.AND, self.AND, MemMode.ABSOLUTE_X, 3, 4, 1), # 3d
Instruction(InstructionType.ROL, self.ROL, MemMode.ABSOLUTE_X, 3, 7, 0), # 3e
Instruction(InstructionType.RLA, self.unimplemented, MemMode.ABSOLUTE_X, 0, 7, 0),
Instruction(InstructionType.RTI, self.RTI, MemMode.IMPLIED, 1, 6, 0), # 40
Instruction(InstructionType.EOR, self.EOR, MemMode.INDEXED_INDIRECT, 2, 6, 0),
Instruction(InstructionType.KIL, self.unimplemented, MemMode.IMPLIED, 0, 2, 0),
Instruction(InstructionType.SRE, self.unimplemented, MemMode.INDEXED_INDIRECT, 0, 8, 0),
Instruction(InstructionType.NOP, self.NOP, MemMode.ZEROPAGE, 2, 3, 0), # 44
Instruction(InstructionType.EOR, self.EOR, MemMode.ZEROPAGE, 2, 3, 0), # 45
Instruction(InstructionType.LSR, self.LSR, MemMode.ZEROPAGE, 2, 5, 0), # 46
Instruction(InstructionType.SRE, self.unimplemented, MemMode.ZEROPAGE, 0, 5, 0),
Instruction(InstructionType.PHA, self.PHA, MemMode.IMPLIED, 1, 3, 0), # 48
Instruction(InstructionType.EOR, self.EOR, MemMode.IMMEDIATE, 2, 2, 0), # 49
Instruction(InstructionType.LSR, self.LSR, MemMode.ACCUMULATOR, 1, 2, 0),
Instruction(InstructionType.ALR, self.unimplemented, MemMode.IMMEDIATE, 0, 2, 0),
Instruction(InstructionType.JMP, self.JMP, MemMode.ABSOLUTE, 3, 3, 0), # 4c
Instruction(InstructionType.EOR, self.EOR, MemMode.ABSOLUTE, 3, 4, 0), # 4d
Instruction(InstructionType.LSR, self.LSR, MemMode.ABSOLUTE, 3, 6, 0), # 4e
Instruction(InstructionType.SRE, self.unimplemented, MemMode.ABSOLUTE, 0, 6, 0),
Instruction(InstructionType.BVC, self.BVC, MemMode.RELATIVE, 2, 2, 1), # 50
Instruction(InstructionType.EOR, self.EOR, MemMode.INDIRECT_INDEXED, 2, 5, 1),
Instruction(InstructionType.KIL, self.unimplemented, MemMode.IMPLIED, 0, 2, 0),
Instruction(InstructionType.SRE, self.unimplemented, MemMode.INDIRECT_INDEXED, 0, 8, 0),
Instruction(InstructionType.NOP, self.NOP, MemMode.ZEROPAGE_X, 2, 4, 0), # 54
Instruction(InstructionType.EOR, self.EOR, MemMode.ZEROPAGE_X, 2, 4, 0), # 55
Instruction(InstructionType.LSR, self.LSR, MemMode.ZEROPAGE_X, 2, 6, 0), # 56
Instruction(InstructionType.SRE, self.unimplemented, MemMode.ZEROPAGE_X, 0, 6, 0),
Instruction(InstructionType.CLI, self.CLI, MemMode.IMPLIED, 1, 2, 0), # 58
Instruction(InstructionType.EOR, self.EOR, MemMode.ABSOLUTE_Y, 3, 4, 1), # 59
Instruction(InstructionType.NOP, self.NOP, MemMode.IMPLIED, 1, 2, 0), # 5a
Instruction(InstructionType.SRE, self.unimplemented, MemMode.ABSOLUTE_Y, 0, 7, 0),
```

```
Instruction(InstructionType.NOP, self.NOP, MemMode.ABSOLUTE_X, 3, 4, 1), # 5c
Instruction(InstructionType.EOR, self.EOR, MemMode.ABSOLUTE_X, 3, 4, 1), # 5d
Instruction(InstructionType.LSR, self.LSR, MemMode.ABSOLUTE_X, 3, 7, 0), # 5e
Instruction(InstructionType.SRE, self.unimplemented, MemMode.ABSOLUTE_X, 0, 7, 0),
Instruction(InstructionType.RTS, self.RTS, MemMode.IMPLIED, 1, 6, 0), # 60
Instruction(InstructionType.ADC, self.ADC, MemMode.INDEXED_INDIRECT, 2, 6, 0),
Instruction(InstructionType.KIL, self.unimplemented, MemMode.IMPLIED, 0, 2, 0),
Instruction(InstructionType.RRA, self.unimplemented, MemMode.INDEXED_INDIRECT, 0, 8, 0),
Instruction(InstructionType.NOP, self.NOP, MemMode.ZEROPAGE, 2, 3, 0), # 64
Instruction(InstructionType.ADC, self.ADC, MemMode.ZEROPAGE, 2, 3, 0), # 65
Instruction(InstructionType.ROR, self.ROR, MemMode.ZEROPAGE, 2, 5, 0), # 66
Instruction(InstructionType.RRA, self.unimplemented, MemMode.ZEROPAGE, 0, 5, 0),
Instruction(InstructionType.PLA, self.PLA, MemMode.IMPLIED, 1, 4, 0), # 68
Instruction(InstructionType.ADC, self.ADC, MemMode.IMMEDIATE, 2, 2, 0), # 69
Instruction(InstructionType.ROR, self.ROR, MemMode.ACCUMULATOR, 1, 2, 0), # 6a
Instruction(InstructionType.ARR, self.unimplemented, MemMode.IMMEDIATE, 0, 2, 0),
Instruction(InstructionType.JMP, self.JMP, MemMode.INDIRECT, 3, 5, 0), # 6c
Instruction(InstructionType.ADC, self.ADC, MemMode.ABSOLUTE, 3, 4, 0), # 6d
Instruction(InstructionType.ROR, self.ROR, MemMode.ABSOLUTE, 3, 6, 0), # 6e
Instruction(InstructionType.RRA, self.unimplemented, MemMode.ABSOLUTE, 0, 6, 0),
Instruction(InstructionType.BVS, self.BVS, MemMode.RELATIVE, 2, 2, 1), # 70
Instruction(InstructionType.ADC, self.ADC, MemMode.INDIRECT_INDEXED, 2, 5, 1),
Instruction(InstructionType.KIL, self.unimplemented, MemMode.IMPLIED, 0, 2, 0),
Instruction(InstructionType.RRA, self.unimplemented, MemMode.INDIRECT_INDEXED, 0, 8, 0),
Instruction(InstructionType.NOP, self.NOP, MemMode.ZEROPAGE_X, 2, 4, 0), # 74
Instruction(InstructionType.ADC, self.ADC, MemMode.ZEROPAGE_X, 2, 4, 0), # 75
Instruction(InstructionType.ROR, self.ROR, MemMode.ZEROPAGE_X, 2, 6, 0), # 76
Instruction(InstructionType.RRA, self.unimplemented, MemMode.ZEROPAGE_X, 0, 6, 0),
Instruction(InstructionType.SEI, self.SEI, MemMode.IMPLIED, 1, 2, 0), # 78
Instruction(InstructionType.ADC, self.ADC, MemMode.ABSOLUTE_Y, 3, 4, 1), # 79
Instruction(InstructionType.NOP, self.NOP, MemMode.IMPLIED, 1, 2, 0), # 7a
Instruction(InstructionType.RRA, self.unimplemented, MemMode.ABSOLUTE_Y, 0, 7, 0),
Instruction(InstructionType.NOP, self.NOP, MemMode.ABSOLUTE_X, 3, 4, 1), # 7c
Instruction(InstructionType.ADC, self.ADC, MemMode.ABSOLUTE_X, 3, 4, 1), # 7d
Instruction(InstructionType.ROR, self.ROR, MemMode.ABSOLUTE_X, 3, 7, 0), # 7e
Instruction(InstructionType.RRA, self.unimplemented, MemMode.ABSOLUTE_X, 0, 7, 0),
Instruction(InstructionType.NOP, self.NOP, MemMode.IMMEDIATE, 2, 2, 0), # 80
Instruction(InstructionType.STA, self.STA, MemMode.INDEXED_INDIRECT, 2, 6, 0),
Instruction(InstructionType.NOP, self.NOP, MemMode.IMMEDIATE, 0, 2, 0), # 82
Instruction(InstructionType.SAX, self.unimplemented, MemMode.INDEXED_INDIRECT, 0, 6, 0),
Instruction(InstructionType.STY, self.STY, MemMode.ZEROPAGE, 2, 3, 0), # 84
Instruction(InstructionType.STA, self.STA, MemMode.ZEROPAGE, 2, 3, 0), # 85
Instruction(InstructionType.STX, self.STX, MemMode.ZEROPAGE, 2, 3, 0), # 86
Instruction(InstructionType.SAX, self.unimplemented, MemMode.ZEROPAGE, 0, 3, 0),
Instruction(InstructionType.DEY, self.DEY, MemMode.IMPLIED, 1, 2, 0), # 88
Instruction(InstructionType.NOP, self.NOP, MemMode.IMMEDIATE, 0, 2, 0), # 89
Instruction(InstructionType.TXA, self.TXA, MemMode.IMPLIED, 1, 2, 0), # 8a
Instruction(InstructionType.XAA, self.unimplemented, MemMode.IMMEDIATE, 0, 2, 0),
Instruction(InstructionType.STY, self.STY, MemMode.ABSOLUTE, 3, 4, 0), # 8c
Instruction(InstructionType.STA, self.STA, MemMode.ABSOLUTE, 3, 4, 0), # 8d
Instruction(InstructionType.STX, self.STX, MemMode.ABSOLUTE, 3, 4, 0), # 8e
```

```
Instruction(InstructionType.SAX, self.unimplemented, MemMode.ABSOLUTE, 0, 4, 0),
Instruction(InstructionType.BCC, self.BCC, MemMode.RELATIVE, 2, 2, 1), # 90
Instruction(InstructionType.STA, self.STA, MemMode.INDIRECT_INDEXED, 2, 6, 0),
Instruction(InstructionType.KIL, self.unimplemented, MemMode.IMPLIED, 0, 2, 0),
Instruction(InstructionType.AHX, self.unimplemented, MemMode.INDIRECT_INDEXED, 0, 6, 0),
Instruction(InstructionType.STY, self.STY, MemMode.ZEROPAGE_X, 2, 4, 0), # 94
Instruction(InstructionType.STA, self.STA, MemMode.ZEROPAGE_X, 2, 4, 0), # 95
Instruction(InstructionType.STX, self.STX, MemMode.ZEROPAGE_Y, 2, 4, 0), # 96
Instruction(InstructionType.SAX, self.unimplemented, MemMode.ZEROPAGE_Y, 0, 4, 0),
Instruction(InstructionType.TYA, self.TYA, MemMode.IMPLIED, 1, 2, 0), # 98
Instruction(InstructionType.STA, self.STA, MemMode.ABSOLUTE_Y, 3, 5, 0), # 99
Instruction(InstructionType.TXS, self.TXS, MemMode.IMPLIED, 1, 2, 0), # 9a
Instruction(InstructionType.TAS, self.unimplemented, MemMode.ABSOLUTE_Y, 0, 5, 0),
Instruction(InstructionType.SHY, self.unimplemented, MemMode.ABSOLUTE_X, 0, 5, 0),
Instruction(InstructionType.STA, self.STA, MemMode.ABSOLUTE_X, 3, 5, 0), # 9d
Instruction(InstructionType.SHX, self.unimplemented, MemMode.ABSOLUTE_Y, 0, 5, 0),
Instruction(InstructionType.AHX, self.unimplemented, MemMode.ABSOLUTE_Y, 0, 5, 0),
Instruction(InstructionType.LDY, self.LDY, MemMode.IMMEDIATE, 2, 2, 0), # a0
Instruction(InstructionType.LDA, self.LDA, MemMode.INDEXED_INDIRECT, 2, 6, 0),
Instruction(InstructionType.LDX, self.LDX, MemMode.IMMEDIATE, 2, 2, 0), # a2
Instruction(InstructionType.LAX, self.unimplemented, MemMode.INDEXED_INDIRECT, 0, 6, 0),
Instruction(InstructionType.LDY, self.LDY, MemMode.ZEROPAGE, 2, 3, 0), # a4
Instruction(InstructionType.LDA, self.LDA, MemMode.ZEROPAGE, 2, 3, 0), # a5
Instruction(InstructionType.LDX, self.LDX, MemMode.ZEROPAGE, 2, 3, 0), # a6
Instruction(InstructionType.LAX, self.unimplemented, MemMode.ZEROPAGE, 0, 3, 0),
Instruction(InstructionType.TAY, self.TAY, MemMode.IMPLIED, 1, 2, 0), # a8
Instruction(InstructionType.LDA, self.LDA, MemMode.IMMEDIATE, 2, 2, 0), # a9
Instruction(InstructionType.TAX, self.TAX, MemMode.IMPLIED, 1, 2, 0), # aa
Instruction(InstructionType.LAX, self.unimplemented, MemMode.IMMEDIATE, 0, 2, 0),
Instruction(InstructionType.LDY, self.LDY, MemMode.ABSOLUTE, 3, 4, 0), # ac
Instruction(InstructionType.LDA, self.LDA, MemMode.ABSOLUTE, 3, 4, 0), # ad
Instruction(InstructionType.LDX, self.LDX, MemMode.ABSOLUTE, 3, 4, 0), # ae
Instruction(InstructionType.LAX, self.unimplemented, MemMode.ABSOLUTE, 0, 4, 0),
Instruction(InstructionType.BCS, self.BCS, MemMode.RELATIVE, 2, 2, 1), # b0
Instruction(InstructionType.LDA, self.LDA, MemMode.INDIRECT_INDEXED, 2, 5, 1),
Instruction(InstructionType.KIL, self.unimplemented, MemMode.IMPLIED, 0, 2, 0),
Instruction(InstructionType.LAX, self.unimplemented, MemMode.INDIRECT_INDEXED, 0, 5, 1),
Instruction(InstructionType.LDY, self.LDY, MemMode.ZEROPAGE_X, 2, 4, 0), # b4
Instruction(InstructionType.LDA, self.LDA, MemMode.ZEROPAGE_X, 2, 4, 0), # b5
Instruction(InstructionType.LDX, self.LDX, MemMode.ZEROPAGE_Y, 2, 4, 0), # b6
Instruction(InstructionType.LAX, self.unimplemented, MemMode.ZEROPAGE_Y, 0, 4, 0),
Instruction(InstructionType.CLV, self.CLV, MemMode.IMPLIED, 1, 2, 0), # b8
Instruction(InstructionType.LDA, self.LDA, MemMode.ABSOLUTE_Y, 3, 4, 1), # b9
Instruction(InstructionType.TSX, self.TSX, MemMode.IMPLIED, 1, 2, 0), # ba
Instruction(InstructionType.LAS, self.unimplemented, MemMode.ABSOLUTE_Y, 0, 4, 1),
Instruction(InstructionType.LDY, self.LDY, MemMode.ABSOLUTE_X, 3, 4, 1), # bc
Instruction(InstructionType.LDA, self.LDA, MemMode.ABSOLUTE_X, 3, 4, 1), # bd
Instruction(InstructionType.LDX, self.LDX, MemMode.ABSOLUTE_Y, 3, 4, 1), # be
Instruction(InstructionType.LAX, self.unimplemented, MemMode.ABSOLUTE_Y, 0, 4, 1),
Instruction(InstructionType.CPY, self.CPY, MemMode.IMMEDIATE, 2, 2, 0), # c0
Instruction(InstructionType.CMP, self.CMP, MemMode.INDEXED_INDIRECT, 2, 6, 0),
```

```
Instruction(InstructionType.NOP, self.NOP, MemMode.IMMEDIATE, 0, 2, 0), # c2
Instruction(InstructionType.DCP, self.unimplemented, MemMode.INDEXED_INDIRECT, 0, 8, 0),
Instruction(InstructionType.CPY, self.CPY, MemMode.ZEROPAGE, 2, 3, 0), # c4
Instruction(InstructionType.CMP, self.CMP, MemMode.ZEROPAGE, 2, 3, 0), # c5
Instruction(InstructionType.DEC, self.DEC, MemMode.ZEROPAGE, 2, 5, 0), # c6
Instruction(InstructionType.DCP, self.unimplemented, MemMode.ZEROPAGE, 0, 5, 0),
Instruction(InstructionType.INY, self.INY, MemMode.IMPLIED, 1, 2, 0), # c8
Instruction(InstructionType.CMP, self.CMP, MemMode.IMMEDIATE, 2, 2, 0), # c9
Instruction(InstructionType.DEX, self.DEX, MemMode.IMPLIED, 1, 2, 0), # ca
Instruction(InstructionType.AXS, self.unimplemented, MemMode.IMMEDIATE, 0, 2, 0),
Instruction(InstructionType.CPY, self.CPY, MemMode.ABSOLUTE, 3, 4, 0), # cc
Instruction(InstructionType.CMP, self.CMP, MemMode.ABSOLUTE, 3, 4, 0), # cd
Instruction(InstructionType.DEC, self.DEC, MemMode.ABSOLUTE, 3, 6, 0), # ce
Instruction(InstructionType.DCP, self.unimplemented, MemMode.ABSOLUTE, 0, 6, 0),
Instruction(InstructionType.BNE, self.BNE, MemMode.RELATIVE, 2, 2, 1), # d0
Instruction(InstructionType.CMP, self.CMP, MemMode.INDIRECT_INDEXED, 2, 5, 1),
Instruction(InstructionType.KIL, self.unimplemented, MemMode.IMPLIED, 0, 2, 0),
Instruction(InstructionType.DCP, self.unimplemented, MemMode.INDIRECT_INDEXED, 0, 8, 0),
Instruction(InstructionType.NOP, self.NOP, MemMode.ZEROPAGE_X, 2, 4, 0), # d4
Instruction(InstructionType.CMP, self.CMP, MemMode.ZEROPAGE_X, 2, 4, 0), # d5
Instruction(InstructionType.DEC, self.DEC, MemMode.ZEROPAGE_X, 2, 6, 0), # d6
Instruction(InstructionType.DCP, self.unimplemented, MemMode.ZEROPAGE_X, 0, 6, 0),
Instruction(InstructionType.CLD, self.CLD, MemMode.IMPLIED, 1, 2, 0), # d8
Instruction(InstructionType.CMP, self.CMP, MemMode.ABSOLUTE_Y, 3, 4, 1), # d9
Instruction(InstructionType.NOP, self.NOP, MemMode.IMPLIED, 1, 2, 0), # da
Instruction(InstructionType.DCP, self.unimplemented, MemMode.ABSOLUTE_Y, 0, 7, 0),
Instruction(InstructionType.NOP, self.NOP, MemMode.ABSOLUTE_X, 3, 4, 1), # dc
Instruction(InstructionType.CMP, self.CMP, MemMode.ABSOLUTE_X, 3, 4, 1), # dd
Instruction(InstructionType.DEC, self.DEC, MemMode.ABSOLUTE_X, 3, 7, 0), # de
Instruction(InstructionType.DCP, self.unimplemented, MemMode.ABSOLUTE_X, 0, 7, 0),
Instruction(InstructionType.CPX, self.CPX, MemMode.IMMEDIATE, 2, 2, 0), # e0
Instruction(InstructionType.SBC, self.SBC, MemMode.INDEXED_INDIRECT, 2, 6, 0),
Instruction(InstructionType.NOP, self.NOP, MemMode.IMMEDIATE, 0, 2, 0), # e2
Instruction(InstructionType.ISC, self.unimplemented, MemMode.INDEXED_INDIRECT, 0, 8, 0),
Instruction(InstructionType.CPX, self.CPX, MemMode.ZEROPAGE, 2, 3, 0), # e4
Instruction(InstructionType.SBC, self.SBC, MemMode.ZEROPAGE, 2, 3, 0), # e5
Instruction(InstructionType.INC, self.INC, MemMode.ZEROPAGE, 2, 5, 0), # e6
Instruction(InstructionType.ISC, self.unimplemented, MemMode.ZEROPAGE, 0, 5, 0),
Instruction(InstructionType.INX, self.INX, MemMode.IMPLIED, 1, 2, 0), # e8
Instruction(InstructionType.SBC, self.SBC, MemMode.IMMEDIATE, 2, 2, 0), # e9
Instruction(InstructionType.NOP, self.NOP, MemMode.IMPLIED, 1, 2, 0), # ea
Instruction(InstructionType.SBC, self.SBC, MemMode.IMMEDIATE, 0, 2, 0), # eb
Instruction(InstructionType.CPX, self.CPX, MemMode.ABSOLUTE, 3, 4, 0), # ec
Instruction(InstructionType.SBC, self.SBC, MemMode.ABSOLUTE, 3, 4, 0), # ed
Instruction(InstructionType.INC, self.INC, MemMode.ABSOLUTE, 3, 6, 0), # ee
Instruction(InstructionType.ISC, self.unimplemented, MemMode.ABSOLUTE, 0, 6, 0),
Instruction(InstructionType.BEQ, self.BEQ, MemMode.RELATIVE, 2, 2, 1), # f0
Instruction(InstructionType.SBC, self.SBC, MemMode.INDIRECT_INDEXED, 2, 5, 1),
Instruction(InstructionType.KIL, self.unimplemented, MemMode.IMPLIED, 0, 2, 0),
Instruction(InstructionType.ISC, self.unimplemented, MemMode.INDIRECT_INDEXED, 0, 8, 0),
Instruction(InstructionType.NOP, self.NOP, MemMode.ZEROPAGE_X, 2, 4, 0), # f4
```

```
    Instruction(InstructionType.SBC, self.SBC, MemMode.ZEROPAGE_X, 2, 4, 0), # f5
    Instruction(InstructionType.INC, self.INC, MemMode.ZEROPAGE_X, 2, 6, 0), # f6
    Instruction(InstructionType.ISC, self.unimplemented, MemMode.ZEROPAGE_X, 0, 6, 0),
    Instruction(InstructionType.SED, self.SED, MemMode.IMPLIED, 1, 2, 0), # f8
    Instruction(InstructionType.SBC, self.SBC, MemMode.ABSOLUTE_Y, 3, 4, 1), # f9
    Instruction(InstructionType.NOP, self.NOP, MemMode.IMPLIED, 1, 2, 0), # fa
    Instruction(InstructionType.ISC, self.unimplemented, MemMode.ABSOLUTE_Y, 0, 7, 0),
    Instruction(InstructionType.NOP, self.NOP, MemMode.ABSOLUTE_X, 3, 4, 1), # fc
    Instruction(InstructionType.SBC, self.SBC, MemMode.ABSOLUTE_X, 3, 4, 1), # fd
    Instruction(InstructionType.INC, self.INC, MemMode.ABSOLUTE_X, 3, 7, 0), # fe
    Instruction(InstructionType.ISC, self.unimplemented, MemMode.ABSOLUTE_X, 0, 7, 0),
]
```

Ручное кодирование этой таблицы переходов было бы невероятно утомительным занятием. Вместо этого я написал внешний скрипт для автоматического создания таблицы из открытых источников. Скрипт был простой программой, которую я сочинил для генерации таблицы, поэтому я не включил его в репозиторий. Однако иногда такие быстрые и простые скрипты позволяют сэкономить много времени на наборе текста!

Инструкции

Далее нам нужно объявить все методы, которые воплощают инструкции 6502 в жизнь. Как уже упоминалось, нам нужно реализовать 56 уникальных методов, расположенных в алфавитном порядке от ADC до TYA, которые обрабатывают такие задачи, как арифметика, управление потоком и т. п.

Как и в случае с проектом CHIP-8, здесь уместно сделать паузу и попробовать написать некоторые методы самостоятельно, прежде чем смотреть на представленные здесь реализации. Для этого вам понадобится хороший справочник по инструкциям 6502. В интернете доступно много справочников, и на упомянутом ранее сайте *https://nesdev.org* есть ссылки на несколько из них. Хороший справочник должен содержать следующую информацию:

- название инструкции, включая ее общепринятую мнемонику;
- операционный код для различных форм инструкции;
- поддерживаемые режимы памяти;
- на какие флаги, если таковые имеются, влияет инструкция;
- сколько циклов она занимает;
- с какими регистрами она работает;
- пример того, что она делает.

Если вы решите реализовать инструкции самостоятельно, сначала ознакомьтесь с остальной частью класса `CPU`, чтобы разобраться, какие вспомогательные методы вам доступны. Существуют методы для изменения стека, чтения из памяти и записи в память, а также

несколько других служебных методов. См. «Доступ к памяти» и «Вспомогательные методы» для ознакомления с этими методами.

Вы убедитесь, что многие инструкции довольно просты. Например, AND – это именно та самая ожидаемая логическая операция AND. Мы берем аккумулятор (`self.A`), выполняем побитовую операцию AND между ним и тем, что мы читаем из памяти, а затем сохраняем результат обратно в аккумулятор:

```
def AND(self, instruction: Instruction, data: int):
    src = self.read_memory(data, instruction.mode)
    self.A = self.A & src
    self.setZN(self.A)
```

Обратите внимание, что здесь абстрагированы два момента. Чтение из памяти выполняется другим методом, `self.read_memory()`, которому передается режим памяти инструкции. Мы вернемся к реализации этого метода позже. Во-вторых, многие различные инструкции влияют на флаги, поэтому у нас есть такие методы, как `self.setZN()`, для обработки изменений флагов. Это классический принцип «не повторяйся» (don't repeat yourself – DRY).

Далее следуют реализации всех 56 необходимых методов. Мы пишем на Python то, что 6502 делал бы на аппаратном уровне, и это действительно не сложная наука. Python имеет операторы для выполнения большинства задач. Другой набор навыков, который больше всего помогает в такой работе, – это глубокое понимание побитовых операторов, так как есть несколько мест, где инструкция явно их запрашивает, или нам нужно отсечь результат, чтобы убедиться, что он по-прежнему состоит из 8 бит и помещается в регистр. О том, как работают эти побитовые операторы, рассказано в приложении.

```
# Добавить память в аккумулятор с переносом
def ADC(self, instruction: Instruction, data: int):
    src = self.read_memory(data, instruction.mode)
    signed_result = src + self.A + self.C
    self.V = bool(~(self.A ^ src) & (self.A ^ signed_result) & 0x80)
    self.A = (self.A + src + self.C) % 256
    self.C = signed_result > 0xFF
    self.setZN(self.A)

# Побитовое AND с аккумулятором
def AND(self, instruction: Instruction, data: int):
    src = self.read_memory(data, instruction.mode)
    self.A = self.A & src
    self.setZN(self.A)

# Арифметический сдвиг влево
def ASL(self, instruction: Instruction, data: int):
    src = self.A if instruction.mode == MemMode.ACCUMULATOR else (
```

```
            self.read_memory(data, instruction.mode))
        self.C = bool(src >> 7) # carry is set to 7th bit
        src = (src << 1) & 0xFF
        self.setZN(src)
        if instruction.mode == MemMode.ACCUMULATOR:
            self.A = src
        else:
            self.write_memory(data, instruction.mode, src)

    # Разветвление, если перенос обнулен
    def BCC(self, instruction: Instruction, data: int):
        if not self.C:
            self.PC = self.address_for_mode(data, instruction.mode)
            self.jumped = True

    # Разветвление, если перенос установлен
    def BCS(self, instruction: Instruction, data: int):
        if self.C:
            self.PC = self.address_for_mode(data, instruction.mode)
            self.jumped = True

    # Разветвление при нулевом результате
    def BEQ(self, instruction: Instruction, data: int):
        if self.Z:
            self.PC = self.address_for_mode(data, instruction.mode)
            self.jumped = True

    # Битовый тест битов в памяти с аккумулятором
    def BIT(self, instruction: Instruction, data: int):
        src = self.read_memory(data, instruction.mode)
        self.V = bool((src >> 6) & 1)
        self.Z = ((src & self.A) == 0)
        self.N = ((src >> 7) == 1)

    # Ветвление по результату минус
    def BMI(self, instruction: Instruction, data: int):
        if self.N:
            self.PC = self.address_for_mode(data, instruction.mode)
            self.jumped = True

    # Ветвление по ненулевому результату
    def BNE(self, instruction: Instruction, data: int):
        if not self.Z:
            self.PC = self.address_for_mode(data, instruction.mode)
            self.jumped = True

    # Ветвление по результату плюс
    def BPL(self, instruction: Instruction, data: int):
        if not self.N:
            self.PC = self.address_for_mode(data, instruction.mode)
            self.jumped = True

    # Принудительный разрыв
    def BRK(self, instruction: Instruction, data: int):
```

```
        self.PC += 2
        # Добавление PC в стек
        self.stack_push((self.PC >> 8) & 0xFF)
        self.stack_push(self.PC & 0xFF)
        # Добавление статуса в стек
        self.B = True
        self.stack_push(self.status)
        self.B = False
        self.I = True
        # Установка PC на вектор сброса
        self.PC = (self.read_memory(IRQ_BRK_VECTOR, MemMode.ABSOLUTE)) | \
                  (self.read_memory(IRQ_BRK_VECTOR + 1, MemMode.ABSOLUTE) << 8)
        self.jumped = True

    # Ветвление при сбросе переполнения
    def BVC(self, instruction: Instruction, data: int):
        if not self.V:
            self.PC = self.address_for_mode(data, instruction.mode)
            self.jumped = True

    # Ветвление при установке переполнения
    def BVS(self, instruction: Instruction, data: int):
        if self.V:
            self.PC = self.address_for_mode(data, instruction.mode)
            self.jumped = True

    # Очистка переноса
    def CLC(self, instruction: Instruction, data: int):
        self.C = False

    # Очистка десятичного знака
    def CLD(self, instruction: Instruction, data: int):
        self.D = False

    # Очистка прерывания
    def CLI(self, instruction: Instruction, data: int):
        self.I = False

    # Очистка переполнения
    def CLV(self, instruction: Instruction, data: int):
        self.V = False

    # Сравнение аккумулятора
    def CMP(self, instruction: Instruction, data: int):
        src = self.read_memory(data, instruction.mode)
        self.C = self.A >= src
        self.setZN(self.A - src)

    # Сравнение регистра X
    def CPX(self, instruction: Instruction, data: int):
        src = self.read_memory(data, instruction.mode)
        self.C = self.X >= src
        self.setZN(self.X - src)
```

```
# Сравнение регистра Y
def CPY(self, instruction: Instruction, data: int):
    src = self.read_memory(data, instruction.mode)
    self.C = self.Y >= src
    self.setZN(self.Y - src)

# Декремент памяти
def DEC(self, instruction: Instruction, data: int):
    src = self.read_memory(data, instruction.mode)
    src = (src - 1) & 0xFF
    self.write_memory(data, instruction.mode, src)
    self.setZN(src)

# Декремент X
def DEX(self, instruction: Instruction, data: int):
    self.X = (self.X - 1) & 0xFF
    self.setZN(self.X)

# Декремент Y
def DEY(self, instruction: Instruction, data: int):
    self.Y = (self.Y - 1) & 0xFF
    self.setZN(self.Y)

# Эксклюзивная память с аккумулятором
def EOR(self, instruction: Instruction, data: int):
    self.A ^= self.read_memory(data, instruction.mode)
    self.setZN(self.A)

# Инкремент памяти
def INC(self, instruction: Instruction, data: int):
    src = self.read_memory(data, instruction.mode)
    src = (src + 1) & 0xFF
    self.write_memory(data, instruction.mode, src)
    self.setZN(src)

# Инкремент X
def INX(self, instruction: Instruction, data: int):
    self.X = (self.X + 1) & 0xFF
    self.setZN(self.X)

# Инкремент Y
def INY(self, instruction: Instruction, data: int):
    self.Y = (self.Y + 1) & 0xFF
    self.setZN(self.Y)

# Переход
def JMP(self, instruction: Instruction, data: int):
    self.PC = self.address_for_mode(data, instruction.mode)
    self.jumped = True

# Переход к подпрограмме
def JSR(self, instruction: Instruction, data: int):
    self.PC += 2
```

```
        # Размещение PC в стеке
        self.stack_push((self.PC >> 8) & 0xFF)
        self.stack_push(self.PC & 0xFF)
        # Переход к подпрограмме
        self.PC = self.address_for_mode(data, instruction.mode)
        self.jumped = True

    # Загрузка аккумулятора с памятью
    def LDA(self, instruction: Instruction, data: int):
        self.A = self.read_memory(data, instruction.mode)
        self.setZN(self.A)

    # Загрузка X с памятью
    def LDX(self, instruction: Instruction, data: int):
        self.X = self.read_memory(data, instruction.mode)
        self.setZN(self.X)

    # Загрузка Y с памятью
    def LDY(self, instruction: Instruction, data: int):
        self.Y = self.read_memory(data, instruction.mode)
        self.setZN(self.Y)

    # Логический сдвиг вправо
    def LSR(self, instruction: Instruction, data: int):
        src = self.A if instruction.mode == MemMode.ACCUMULATOR else (
            self.read_memory(data, instruction.mode))
        self.C = bool(src & 1) # carry is set to 0th bit
        src >>= 1
        self.setZN(src)
        if instruction.mode == MemMode.ACCUMULATOR:
            self.A = src
        else:
            self.write_memory(data, instruction.mode, src)

    # Не операция
    def NOP(self, instruction: Instruction, data: int):
        pass

    # Or память с аккумулятором
    def ORA(self, instruction: Instruction, data: int):
        self.A |= self.read_memory(data, instruction.mode)
        self.setZN(self.A)

    # Аккумулятор отправки
    def PHA(self, instruction: Instruction, data: int):
        self.stack_push(self.A)

    # Статус отправки
    def PHP(self, instruction: Instruction, data: int):
        # https://nesdev.org/the%20'B'%20flag%20&%20BRK%20instruction.txt
        self.B = True
        self.stack_push(self.status)
        self.B = False
```

```
# Извлечение аккумулятора
def PLA(self, instruction: Instruction, data: int):
    self.A = self.stack_pop()
    self.setZN(self.A)

# Извлечение статуса
def PLP(self, instruction: Instruction, data: int):
    self.set_status(self.stack_pop())

# Поворот на один бит влево
def ROL(self, instruction: Instruction, data: int):
    src = self.A if instruction.mode == MemMode.ACCUMULATOR else (
        self.read_memory(data, instruction.mode))
    old_c = self.C
    self.C = bool((src >> 7) & 1) # carry is set to 7th bit
    src = ((src << 1) | old_c) & 0xFF
    self.setZN(src)
    if instruction.mode == MemMode.ACCUMULATOR:
        self.A = src
    else:
        self.write_memory(data, instruction.mode, src)

# Поворот на один бит вправо
def ROR(self, instruction: Instruction, data: int):
    src = self.A if instruction.mode == MemMode.ACCUMULATOR else (
        self.read_memory(data, instruction.mode))
    old_c = self.C
    self.C = bool(src & 1) # carry is set to 0th bit
    src = ((src >> 1) | (old_c << 7)) & 0xFF
    self.setZN(src)
    if instruction.mode == MemMode.ACCUMULATOR:
        self.A = src
    else:
        self.write_memory(data, instruction.mode, src)

# Возврат из прерывания
def RTI(self, instruction: Instruction, data: int):
    # Извлечение статуса
    self.set_status(self.stack_pop())
    # Извлечение PC
    lb = self.stack_pop()
    hb = self.stack_pop()
    self.PC = ((hb << 8) | lb)
    self.jumped = True

# Возврат из подпрограммы
def RTS(self, instruction: Instruction, data: int):
    # Извлечение PC
    lb = self.stack_pop()
    hb = self.stack_pop()
    self.PC = ((hb << 8) | lb) + 1 # 1 past last instruction
    self.jumped = True
```

```
# Вычитание с переносом
def SBC(self, instruction: Instruction, data: int):
    src = self.read_memory(data, instruction.mode)
    signed_result = self.A - src - (1 - self.C)
    # Установка переполнения
    self.V = bool((self.A ^ src) & (self.A ^ signed_result) & 0x80)
    self.A = (self.A - src - (1 - self.C)) % 256
    self.C = not (signed_result < 0) # set carry
    self.setZN(self.A)

# Установка переноса
def SEC(self, instruction: Instruction, data: int):
    self.C = True

# Установка десятичного разряда
def SED(self, instruction: Instruction, data: int):
    self.D = True

# Установка прерывания
def SEI(self, instruction: Instruction, data: int):
    self.I = True

# Сохранение аккумулятора
def STA(self, instruction: Instruction, data: int):
    self.write_memory(data, instruction.mode, self.A)

# Сохранение регистра X
def STX(self, instruction: Instruction, data: int):
    self.write_memory(data, instruction.mode, self.X)

# Сохранение регистра Y
def STY(self, instruction: Instruction, data: int):
    self.write_memory(data, instruction.mode, self.Y)

# Перенос A в X
def TAX(self, instruction: Instruction, data: int):
    self.X = self.A
    self.setZN(self.X)

# Перенос A в Y
def TAY(self, instruction: Instruction, data: int):
    self.Y = self.A
    self.setZN(self.Y)

# Перенос указателя стека в X
def TSX(self, instruction: Instruction, data: int):
    self.X = self.SP
    self.setZN(self.X)

# Перенос X в A
def TXA(self, instruction: Instruction, data: int):
    self.A = self.X
    self.setZN(self.A)
```

```
# Перенос X в SP
def TXS(self, instruction: Instruction, data: int):
    self.SP = self.X

# Перенос Y в A
def TYA(self, instruction: Instruction, data: int):
    self.A = self.Y
    self.setZN(self.A)

def unimplemented(self, instruction: Instruction, data: int):
    print(f"{instruction.type.name} is unimplemented.")
```

Хотя большинство инструкций довольно просты, я обнаружил, что обработка сложения с переносом (add with carry, ADC) и вычитания с переносом (subtract with carry, SBC) несколько сложнее. Основные регистры 6502 имеют всего 8 бит, поэтому переносы будут происходить часто, и при этом нужно правильно устанавливать флаги. Но у нас есть хитрость: тип int в Python не ограничен 8 битами. Поэтому мы можем выполнять арифметические операции, как если бы мы работали с обычными значениями `int`, а затем просто удалить все, что превышает 255.

Метод step()

После реализации всех инструкций мы готовы к пошаговому выполнению фактического машинного кода 6502. Метод `step()` считывает следующий операционный код в `PC` и извлекает инструкцию из таблицы переходов. Вот начало этого метода:

```
def step(self):
    if self.stall > 0:
        self.stall -= 1
        self.cpu_ticks += 1
        return

    opcode = self.read_memory(self.PC, MemMode.ABSOLUTE)
    self.page_crossed = False
    self.jumped = False
    instruction = self.instructions[opcode]
    data = 0
    for i in range(1, instruction.length):
    data |= (self.read_memory(self.PC + i,
                              MemMode.ABSOLUTE) << ((i - 1) * 8))
```

Большинство инструкций 6502 также сопровождаются некоторыми данными, количество байтов которых может варьироваться. Например, инструкция `TAY` (перенос `A` в `Y`) не требует данных, поэтому занимает всего 1 байт, но любая инструкция, которая считывает данные из памяти, потребует дополнительных данных для указания адреса па-

мяти. Объем считываемых данных указывается с помощью `instruction.length`.

Метод `step()` продолжается следующим образом:

```
    instruction.method(instruction, data)

    if not self.jumped:
    self.PC += instruction.length
    elif instruction.type in {InstructionType.BCC, InstructionType.BCS,
                              InstructionType.BEQ, InstructionType.BMI,
                              InstructionType.BNE, InstructionType.BPL,
                              InstructionType.BVC, InstructionType.BVS}:
    # Инструкции ветвления добавляют +1 тик в случае их успешного выполнения
    self.cpu_ticks += 1
    self.cpu_ticks += instruction.ticks
    if self.page_crossed:
    self.cpu_ticks += instruction.page_ticks
```

Мы вызываем фактический метод инструкции для ее выполнения, а затем инкрементируем программный счетчик по мере необходимости. Завершаем работу некоторыми подсчетами, касающимися тиков (циклов работы центрального процессора).

Доступ к памяти

Следующие несколько методов, которые мы напишем, помогут с чтением и записью в память. Доступ к памяти – одна из самых сложных областей 6502, поскольку он имеет множество различных режимов доступа к памяти. Метод `address_for_mode()` отвечает за передачу данных, связанных с инструкцией, в конкретный адрес памяти на основе режима инструкции (`MemMode`):

```
def address_for_mode(self, data: int, mode: MemMode) -> int:
    def different_pages(address1: int, address2: int) -> bool:
        return (address1 & 0xFF00) != (address2 & 0xFF00)

    address = 0
    match mode:
    case MemMode.ABSOLUTE:
        address = data
    case MemMode.ABSOLUTE_X:
        address = (data + self.X) & 0xFFFF
        self.page_crossed = different_pages(address, address - self.X)
    case MemMode.ABSOLUTE_Y:
        address = (data + self.Y) & 0xFFFF
        self.page_crossed = different_pages(address, address - self.Y)
    case MemMode.INDEXED_INDIRECT:
        # 0xFF для переноса нулевой страницы в следующие две строки
        ls = self.ram[(data + self.X) & 0xFF]
```

```
            ms = self.ram[(data + self.X + 1) & 0xFF]
            address = (ms << 8) | ls
        case MemMode.INDIRECT:
            ls = self.ram[data]
            ms = self.ram[data + 1]
            if (data & 0xFF) == 0xFF:
                ms = self.ram[data & 0xFF00]
            address = (ms << 8) | ls
        case MemMode.INDIRECT_INDEXED:
            # 0xFF для переноса нулевой страницы в следующие две строки
            ls = self.ram[data & 0xFF]
            ms = self.ram[(data + 1) & 0xFF]
            address = (ms << 8) | ls
            address = (address + self.Y) & 0xFFFF
            self.page_crossed = different_pages(address, address - self.Y)
        case MemMode.RELATIVE:
            address = (self.PC + 2 + data) & 0xFFFF if (data < 0x80) \
                else (self.PC + 2 + (data - 256)) & 0xFFFF # signed
        case MemMode.ZEROPAGE:
            address = data
        case MemMode.ZEROPAGE_X:
            address = (data + self.X) & 0xFF
        case MemMode.ZEROPAGE_Y:
            address = (data + self.Y) & 0xFF
        return address
```

Чтобы понять этот код, приведем разбивку режимов доступа к памяти 6502 и их связь с данными, ассоциированными с инструкцией:

ABSOLUTE Адрес является данными.

ABSOLUTE_X Регистр `X` добавляется к данным для формирования адреса.

ABSOLUTE_Y Регистр `Y` добавляется к данным для формирования адреса.

ACCUMULATOR Используется регистр `A`, а не память. Мы обрабатываем этот режим непосредственно в отдельных методах инструкций, поэтому он не появляется в `address_for_mode()`.

IMMEDIATE Данные являются конечным элементом; мы фактически не обращаемся к памяти. Мы обрабатываем этот режим непосредственно в методах чтения и записи в память.

INDEXED_INDIRECT 2-байтовый адрес находится в ОЗУ по адресу `data + X`.

INDIRECT 2-байтовый адрес находится в ОЗУ по адресу `data`.

INDIRECT_INDEXED Адрес по адресу `data` в ОЗУ добавляется к регистру `Y` для формирования окончательного адреса.

RELATIVE	data добавляется к PC для формирования адреса.
ZEROPAGE	Это похоже на ABSOLUTE, но в пределах первых 256 байт памяти (*нулевая страница* – zero page).
ZEROPAGE_X	Это похоже на ABSOLUTE_X, но в пределах первых 256 байт памяти.
ZEROPAGE_Y	Это похоже на ABSOLUTE_Y, но в пределах первых 256 байт памяти.

Режимы ZEROPAGE могут показаться избыточными на фоне режимов ABSOLUTE, но здесь есть хорошая оптимизация: поскольку нулевая страница в памяти составляет всего 256 байт, для указания адреса в ней требуется только один байт данных. Это экономит один цикл времени центрального процессора (второй байт адреса извлекать не нужно) при выполнении инструкции, использующей нулевую страницу. Таким образом, инструкции в режиме ZEROPAGE выполняются быстрее, чем другие инструкции с доступом к памяти. В действительности программисты 6502 иногда рассматривают 256 ячеек памяти на нулевой странице как дополнительные регистры, поскольку они очень быстрые при чтении и записи. Это помогает компенсировать небольшое количество фактических регистров в 6502.

Теперь, когда мы можем формировать адреса памяти, мы ближе к чтению и записи памяти, но нам также необходимы некоторые знания о различных областях памяти NES – какие адреса сопоставляются с ОЗУ, какие с PPU и т. д. Важно отметить, что многие области памяти NES содержат обширное зеркалирование. Например, первые 2 Кбайта в карте памяти, или адреса до 0x800 в шестнадцатеричном формате, сопоставляются с ОЗУ процессора, но любой адрес ниже 0x2000 сопоставляется с той же ОЗУ, потому что 2 Кбайта повторяются четыре раза до 0x2000. Другими словами, адрес 0x801 совпадает с адресом 0x001, так же, как и адреса 0x1001 и 0x1801 (2 × 0x800 в шестнадцатеричном формате равно 0x1000, а 3 × 0x800 равно 0x1800).

Это зеркалирование было результатом особенностей аппаратного обеспечения NES, направленных на сокращение издержек. Например, не все аппаратные линии памяти 6502, которые можно представить в виде проводов, были необходимы для адресации тех скромных 2 Кбайт ОЗУ, поэтому некоторые аппаратные линии просто игнорировались. Для иллюстрации: 0x800 – это 2048 в десятичной системе и 100000000000 в двоичной. Каждую цифру в двоичной системе можно представить в виде сигнала, передаваемого аппаратной линией. Для адресации первых 2048 адресов требуется всего 11 аппаратных линий, что соответствует 11 конечным нулям в нашем числе. Этих 11 аппаратных линий достаточно для 2048 различных значений, десятичных значений от 0 до 2047. 12-я аппаратная линия, представленная цифрой 1 в двоичной системе, может быть просто проигнорирована. Другими словами, 0x800 сопоставляется с 12 аппаратными

линиями, но только 11 из этих аппаратных линий действительно существуют, поэтому 0x800 по сути то же самое, что и 0x0.

В табл. 6.4 показана карта памяти процессора NES, как ее видит наш эмулятор. Она включает как области, так и отдельные адреса отображения памяти. Области и адреса, которые наш эмулятор игнорирует или не реализует, не представлены. В таблице также указано, является ли область или адрес доступным для чтения, записи или для обоих видов доступа. Наконец, в ней указано, применяется ли зеркалирование.

Таблица 6.4 Упрощенная карта памяти NES

Адрес или область	Длина	Описание	Чтение/запись	Зеркалирование?
0x0000–0x1FFF	0x800	Основные 2 Кбайта памяти процессора	RW	Да, первые 0x800 зеркально отражаются до 0x2000
0x2000–0x3FFF	0x8	8 регистров PPU	Варьируется	От 0x2000 до 0x2007 зеркалируется каждые 8 байт
0x4014	0x1	DMA-передача данных спрайтов	W	Нет
0x4016	0x1	Состояние геймпада 1	RW	Нет
0x6000–0xFFFF	Варьируется в зависимости от картриджа	Память картриджа	Варьируется	Варьируется

В табл. 6.4 отсутствует много адресов и деталей. Например, наш простой эмулятор не имеет APU, но реальная NES имеет адреса, отображенные в памяти в диапазоне 0x4000 для APU и других устройств ввода-вывода (таких как второй геймпад). Мы также не указываем отдельные регистры PPU между 0x2000 и 0x2007, к которым мы вернемся при реализации PPU. Пространство памяти картриджа может значительно варьироваться в зависимости от маппера. Это выходит за рамки данного раздела; мы обсудили некоторые особенности памяти картриджа в разделе «Эмуляция картриджа». Наконец, в жалких 2 Кбайтах ОЗУ процессора на самом деле есть два важных региона: уже обсуждавшийся быстрый регион нулевой страницы от 0x0000 до 0x00FF и пространство, обычно используемое для стека, от 0x0100 до 0x01FF.

Теперь, когда мы имеем некоторое представление о распределении памяти, можно рассмотреть методы ее чтения и записи. Начнем с метода `read_memory()`. Он принимает адрес для чтения и режим памяти и возвращает байт (представленный в Python как `int`) из этого адреса:

```
def read_memory(self, location: int, mode: MemMode) -> int:
    if mode == MemMode.IMMEDIATE:
        return location # location is actually data in this case
    address = self.address_for_mode(location, mode)

    # Карта памяти на https://wiki.nesdev.org/w/index.php/CPU_memory_map
```

```
if address < 0x2000: # основная оперативная память 2 Кбайта увеличивается до 0x800
    return self.ram[address % 0x800] # зеркалирование для следующих 6 Кбайт
elif address < 0x4000: # 2000-2007 – это PPU, зеркалирование каждые 8 байт
    temp = ((address % 8) | 0x2000) # получение данных из регистра PPU
    return self.ppu.read_register(temp)
elif address == 0x4016: # статус joypad 1
if self.joypad1.strobe:
    return self.joypad1.a
self.joypad1.read_count += 1
match self.joypad1.read_count:
    case 1:
        return 0x40 | self.joypad1.a
    case 2:
        return 0x40 | self.joypad1.b
    case 3:
        return 0x40 | self.joypad1.select
    case 4:
        return 0x40 | self.joypad1.start
    case 5:
        return 0x40 | self.joypad1.up
    case 6:
        return 0x40 | self.joypad1.down
    case 7:
        return 0x40 | self.joypad1.left
    case 8:
        return 0x40 | self.joypad1.right
    case _:
        return 0x41
elif address < 0x6000:
return 0 # нереализованные другие виды ввода-вывода
else: # адреса от 0x6000 до 0xFFFF принадлежат картриджу
return self.rom.read_cartridge(address)
```

Если выбран режим памяти `IMMEDIATE`, это означает, что фактические данные, связанные с инструкцией, являются теми, которые должны быть «прочитаны». Помните, что наш код абстрагирован до такой степени, что метод для отдельной инструкции должен работать с любым режимом памяти. В случае режима `IMMEDIATE` нам не нужно выполнять фактический поиск, поэтому мы просто возвращаем связанные с инструкцией данные (название `location` здесь немного странное, и это удачное название для всего, кроме режима `IMMEDIATE`). В противном случае `location`, связанное с инструкцией, преобразуется в адрес памяти с помощью `address_for_mode()`.

После получения адреса памяти мы используем серию операторов `if`, чтобы определить, откуда фактически следует читать память, как показано в табл. 6.4. Мы учитываем зеркалирование с помощью модуля. В зависимости от региона может быть доступен процессор, PPU, геймпад или картридж. В NES был необычный способ чтения геймпада. Каждый раз при чтении адреса 0x4016 возвращается состояние

другой кнопки геймпада. Поэтому необходимо выполнить восемь операций чтения, чтобы узнать состояние каждой кнопки.

Далее рассмотрим запись в память:

```
def write_memory(self, location: int, mode: MemMode, value: int):
    if mode == MemMode.IMMEDIATE:
        self.ram[location] = value
        return

    address = self.address_for_mode(location, mode)
    # Карта памяти по адресу https://wiki.nesdev.org/w/index.php/CPU_memory_map
    if address < 0x2000: # основная ОЗУ 2 Кбайта достигает 0x800
        self.ram[address % 0x800] = value # зеркалирование следующих 6 Кбайт
    elif address < 0x3FFF: # 2000-2007 – это PPU, зеркалирование каждые 8 байт
        temp = ((address % 8) | 0x2000) # запись данных в регистр PPU
        self.ppu.write_register(temp, value)
    elif address == 0x4014: # Передача данных спрайта посредством DMA
        from_address = value * 0x100 # адрес, с которого начинается копирование
        for i in range(SPR_RAM_SIZE): # копирование всех 256 байт ОЗУ спрайта
            self.ppu.spr[i] = self.read_memory((from_address + i),
                                               MemMode.ABSOLUTE)
        # Задержка на 512 циклов до завершения этого процесса
        self.stall = 512
    elif address == 0x4016: # joypad 1
        if self.joypad1.strobe and (not bool(value & 1)):
            self.joypad1.read_count = 0
        self.joypad1.strobe = bool(value & 1)
        return
    elif address < 0x6000:
        return # нереализованные другие виды ввода-вывода
    else: # адреса от 0x6000 до 0xFFFF принадлежат картриджу
        # Мы не реализовали поддержку ОЗУ картриджа
        return self.rom.write_cartridge(address, value)
```

Метод `write_memory()` очень похож на `read_memory()`. Он обрабатывает режим `IMMEDIATE`, затем обменивает местоположение на адрес для других режимов и записывает в соответствующее местоположение.

Вспомогательные методы

Мы заканчиваем описание класса CPU серией вспомогательных методов, начиная с этих трех:

```
def setZN(self, value: int):
    self.Z = (value == 0)
    self.N = bool(value & 0x80) or (value < 0)

def stack_push(self, value: int):
    self.ram[(0x100 | self.SP)] = value
```

```
        self.SP = (self.SP - 1) & 0xFF

    def stack_pop(self) -> int:
        self.SP = (self.SP + 1) & 0xFF
        return self.ram[(0x100 | self.SP)]
```

Многие инструкции требуют установки флагов нуля (Z) и отрицательного значения (N). Вместо повторения этих нескольких строк в методах инструкций мы используем `setZN()`. Метод `stack_push()` помещает новое значение в стек. Это подразумевает помещение значения в стек по адресу записи и перемещение указателя стека. Точно так же `stack_pop()` извлекает значение из указателя стека, перемещает указатель стека и возвращает значение.

Для удобства наш класс `CPU` хранит различные флаги состояния в семи отдельных булевых переменных, но реальный 6502 имеет один 8-битный регистр состояния, в котором каждый флаг представляет собой один бит. Инструкции `BRK`, `PHP`, `PLP` и `RTI` должны работать с регистром состояния в его битовом формате, поэтому следующие методы, `status()` и `set_status()`, используют побитовые операции для преобразования между двумя форматами:

```
    @property
    def status(self) -> int:
        return (self.C | self.Z << 1 | self.I << 2 | self.D << 3 |
                self.B << 4 | 1 << 5 | self.V << 6 | self.N << 7)

    def set_status(self, temp: int):
        self.C = bool(temp & 0b00000001)
        self.Z = bool(temp & 0b00000010)
        self.I = bool(temp & 0b00000100)
        self.D = bool(temp & 0b00001000)
        # https://nesdev.org/the%20'B'%20flag%20&%20BRK%20instruction.txt
        self.B = False
        self.V = bool(temp & 0b01000000)
        self.N = bool(temp & 0b10000000)
```

Наконец, у нас есть метод для обработки событий, возникающих при срабатывании NMI, и метод `log()` для отладки:

```
def trigger_NMI(self):
    self.stack_push((self.PC >> 8) & 0xFF)
    self.stack_push(self.PC & 0xFF)
    # https://nesdev.org/the%20'B'%20flag%20&%20BRK%20instruction.txt
    self.B = True
    self.stack_push(self.status)
    self.B = False
    self.I = True
    # Установка PC на вектор NMI
    self.PC = (self.read_memory(NMI_VECTOR, MemMode.ABSOLUTE)) | \
```

```
            (self.read_memory(NMI_VECTOR + 1, MemMode.ABSOLUTE) << 8)

def log(self) -> str:
    opcode = self.read_memory(self.PC, MemMode.ABSOLUTE)
    instruction = self.instructions[opcode]
    data1 = " " if instruction.length < 2 else f"{self.read_memory(self.PC + 1,
                                                                   MemMode.ABSOLUTE):02X}"
    data2 = " " if instruction.length < 3 else f"{self.read_memory(self.PC + 2,
                                                                   MemMode.ABSOLUTE):02X}"
    return f"{self.PC:04X} {opcode:02X} {data1} {data2} {instruction.type.name}{29 * ' '}" \
           f"A:{self.A:02X} X:{self.X:02X} Y:{self.Y:02X} P:{self.status:02X} SP:{self.SP:02X}"
```

NMI отправляет выполнение в блок кода по адресу, указанному в `NMI_VECTOR`. Когда срабатывает NMI, аналогично инструкциям `BRK` и `JSR`, нам нужно поставить закладку, чтобы мы могли вернуться в то место, где мы были до того, как NMI увел нас. В этом и заключается цель помещения `PC` и `status` в стек.

Реализация центрального процессора потребовала написания большого количества кода в соответствии со спецификацией. Эта спецификация длинная, но состоит из множества небольших, относительно простых для понимания частей. Последней частью эмулятора, которую нам нужно реализовать, является PPU, и она будет значительно отличаться от предыдущих. Существует спецификация со множеством деталей, но все эти детали в основном сводятся к двум основным задачам: рисованию фоновых тайлов и рисованию спрайтов.

Принципы работы PPU

Реализация PPU – это самая сложная часть проекта эмуляции NES. В PPU происходит прорисовка графики на экране. Для целей нашего простого эмулятора можно считать, что графика имеет два аспекта: фон (background) и спрайты (sprites).

Фон – это задний слой графики, где отдельные элементы обычно либо не двигаются вовсе, либо прокручиваются вместе в виде группы. Аппаратное обеспечение NES рисует фон с помощью *тайлов*, или «*плиток*» (tiles). Так, в игре-платформере тайл может быть частью платформы, частью художественного фона (например, горой), лестницей или дверью. Эти элементы обычно не перемещаются сами по себе.

Напротив, *спрайты* (sprites) – это отдельные игровые объекты, которые могут перемещаться по экрану независимо друг от друга. Вспомните игрока и врагов в игре-платформере. В NES есть специальное аппаратное оснащение, позволяющее одновременно обрабатывать до 64 спрайтов размером 8×8 или 8×16 (в пикселях) на экране.

Существует множество различных способов реализации PPU. Наиболее точным является имитация реальной работы PPU: генерация

каждого пикселя экрана по одному. Если эта реализация выполнена правильно, любая игра, написанная для NES, должна работать корректно. Этот подход также является наиболее ресурсоемким. Популярной альтернативой является реализация графики по одной строке развертки за раз. Вместо обновления каждого пикселя обновления происходят по завершении обработки каждой строки развертки.

Мы не будем делать ни того, ни другого. Как обсуждалось ранее, мы будем реализовывать PPU с помощью еще более простого подхода, обновляя весь экран по одному кадру за раз. Это наименее точный метод, потому что если игра каким-то образом изменяет графику между пикселями или между строками развертки, эти обновления не будут отображаться. Совместимость по-прежнему будет хорошей, но она не будет распространяться на все игры.

Хотя наш метод невероятно далек от работы реального PPU, он является наиболее производительным, поскольку требует наименьшего количества обновлений на кадр (только одно большое обновление вместо множества маленьких). Он также является наименее сложным с концептуальной точки зрения, поскольку не требует понимания всех деталей работы реального PPU. Поскольку наш эмулятор написан на относительно медленном языке программирования Python и предназначен для максимальной простоты в демонстрационных целях, рендеринг по кадрам, пожалуй, является лучшим выбором.

Хотя нам и не нужно понимать все детали работы PPU для реализации нашего подхода «кадр за кадром», нам все же необходимо понимать некоторые основные моменты о поступлении данных для фонов и спрайтов. Мы углубимся в эту тему в следующих разделах. Некоторая информация уже была разбросана по предыдущим частям главы, но здесь она объединена со многими новыми идеями и деталями, чтобы связно объяснить принципы работы PPU.

CHR ROM

Данные как для фоновых тайлов, так и для спрайтов изначально находятся на картридже в области памяти, известной как CHR ROM. Размер этой памяти может значительно варьироваться. Ранние игры обычно имели всего 8 Кбайт CHR ROM, но более поздние игры с маппперами могли иметь сотни килобайт, с возможностью замены любой другой области 8 Кбайт на первую, чтобы быть совместимыми с требованиями аппаратного обеспечения PPU (которое может напрямую адресовать только 8 Кбайт CHR ROM).

В некоторых играх CHR ROM заменялась CHR RAM, которая могла изменяться во время работы игры. Однако в большинстве игр CHR ROM была фиксированной. Если в игре была CHR RAM, то игра должна была загружать графику в CHR RAM по мере необходимости из PRG ROM, а не просто всегда иметь ее в наличии. В некоторых редких играх были и CHR ROM, и CHR RAM в разных банках.

Таблицы шаблонов и тайлы

В любой момент времени PPU может быть «подключен» к любой из двух 4-килобайтных частей CHR ROM на картридже. Эти части называются *таблицами шаблонов* (pattern tables). Из выбранной таблицы шаблонов NES извлекает все графические данные для данного кадра. Обратите внимание, что в некоторых документах все 8 Кбайт адресуемой CHR ROM называют одной общей таблицей шаблонов, а не отдельными таблицами шаблонов по 4 Кбайта.

Каждая таблица шаблонов разделена на 256 16-байтовых тайлов, и каждый тайл определяет возможную графику для области экрана размером 8×8 пикселей. В совокупности предопределенные тайлы в таблицах шаблонов представляют все различные элементы, которые вы можете увидеть в игре. Например, на рис. 6.1 показаны таблицы шаблонов для игры с открытым исходным кодом *BrickBreaker* от Aleff Correa, которую мы запустим в нашем эмуляторе чуть позже в этой главе.

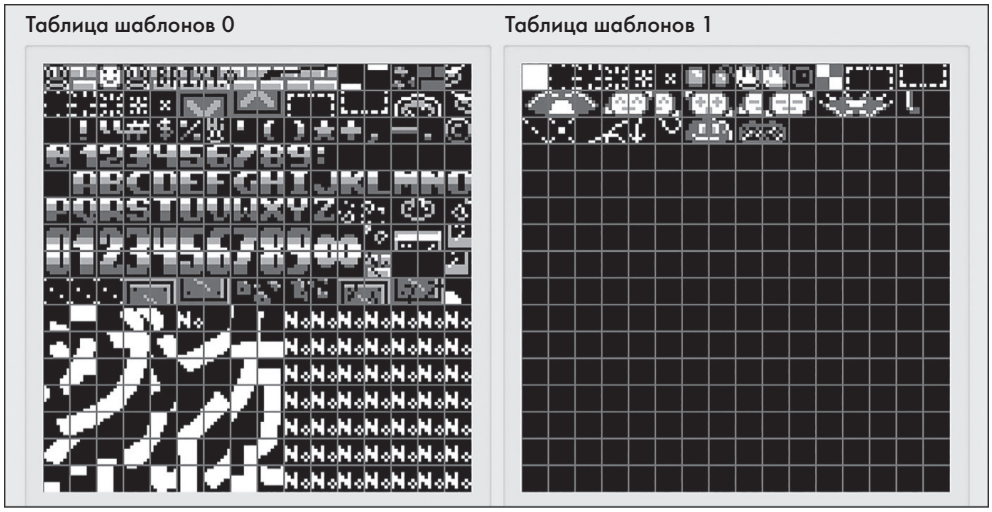


Рис. 6.1. Таблицы шаблонов BrickBreaker, отображаемые PPU Viewer эмулятора FCEUX

Вы можете убедиться, что некоторые тайлы в таблицах шаблонов представляют спрайты, иные – текст, а другие – фоновые шаблоны. Многие тайлы в таблицах шаблонов пусты, потому что *BrickBreaker* в них не нуждается (в CHR ROM для графических ресурсов у игры больше места, чем ей на самом деле нужно).

Примечание *Рисунок 6.1 был создан с помощью функции PPU Viewer эмулятора FCEUX. Эта функция позволяет просматривать содержимое CHR ROM отдельно от фактического игрового процесса. Добавление функций отладки, таких как программа просмотра таблицы шабло-*

нов, может быть очень полезно при написании эмулятора. Например, вы сможете сравнить вывод вашего PPU с выводом такого известного эмулятора, как FCEUX.

Как уже упоминалось, каждый фрагмент в таблице шаблонов занимает 16 байт. Поскольку 16 байт определяют 8 × 8 = 64 пикселя, на каждый пиксель приходится всего 2 бита, а 2 бита могут представлять только четыре разных значения (00, 01, 10, 11). Таким образом, PPU поддерживает только четыре цвета в пределах одного тайла. В действительности один из них всегда является заранее установленным цветом фона или прозрачным (00), поэтому в конкретном тайле может отображаться только три цвета, выбранных программистом. Эта палитра цветов устанавливается для областей из четырех тайлов одновременно и управляется в отдельной части памяти, отличной от самих таблиц шаблонов. Вот почему все тайлы на рис. 6.1 отображаются в оттенках серого; цветовая палитра каждого тайла не определяется таблицей шаблонов. Мы вернемся к вопросу о выборе цветов чуть позже.

К сожалению, с тайлами все немного сложнее: 2-битные значения, определяющие каждый пиксель, не располагаются последовательно. Вместо этого нулевой бит каждого пикселя располагается в первых 8 байтах тайла, а первый бит каждого пикселя – во вторых 8 байтах тайла. Каждые 8 байт образуют *битовую плоскость* (bit plane), и две плоскости объединяются по два бита за раз, чтобы определить цвет каждого пикселя.

Позвольте мне сформулировать это по-другому. Первые 8 байт (первая битовая плоскость) тайла можно рассматривать как 64 половины цветовых значений для 64 пикселей в тайле 8×8. Они расположены последовательно. Вторая 8-байтовая битовая плоскость определяет еще 64 половины цветовых значений для тех же 64 пикселей. Они также расположены последовательно. Две плоскости необходимо объединить, чтобы получить 64 окончательных цветовых значения. В качестве примера рассмотрим следующие 16 байт данных тайла[1]:

```
   Bit Plane 1
Byte 1  01000001
Byte 2  11000010
Byte 3  01000100
Byte 4  01001000     Pixel Pattern
Byte 5  00010000
Byte 6  00100000       01000003
Byte 7  01000000       11000030
Byte 8  10000000 ===== 01000300
```

[1] Пример взят у пользователя Damian Yerrick с сайта NesDev.org. Вся информация на сайте NesDev.org находится в открытом доступе. См. *https://www.nesdev.org/wiki/PPU_pattern_tables*.

```
   Bit Plane 2         01003000
Byte 9  00000001 ===== 00030220
Byte 10 00000010       00300002
Byte 11 00000100       03000020
Byte 12 00001000       30000222
Byte 13 00010110
Byte 14 00100001
Byte 15 01000010
Byte 16 10000111
```

Соответствующие биты из двух битовых плоскостей объединяются, образуя пиксельный узор (pixel pattern), как показано в списке и на рис. 6.2: изображение дроби 1/2.

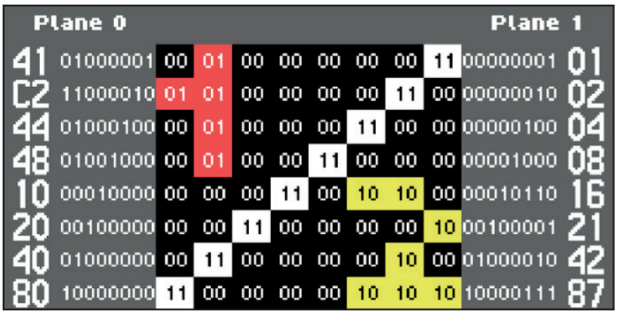


Рис. 6.2. Как из двух битовых плоскостей формируется тайл

Если в первой битовой плоскости стоит 1, а во второй – 0, то эти биты объединяются, образуя 01, или цвет 1 в цветовой схеме. Точно так же 0 из первой битовой плоскости и 1 из второй образуют 10, или цвет 2, а 1 в обеих битовых плоскостях дает 11, или цвет 3.

Таблицы имен

Таблицы имен (nametables) представляют собой место, где располагаются фактические тайлы для фона экрана. Как выглядит фон экрана в данный момент? Из каких тайлов он состоит и какие цвета используются в каждом из них? Именно этим занимается таблица имен и сопутствующая ей таблица атрибутов (о которой пойдет речь далее).

Каждая таблица имен, представляющая один экран игры, имеет ширину 32 тайла и высоту 30 тайлов. Каждый тайл в таблице имен определяется одним байтом – индексом тайла в текущей таблице шаблонов. Таким образом, тайлы таблицы шаблонов напрямую сопоставляются с таблицей имен, поэтому таблицу имен можно рассматривать как определенный порядок тайлов из таблицы шаблонов.

Каков размер таблицы имен? Он составляет $32 \times 30 = 960$, то есть в таблице имен 960 ячеек. Каждая ячейка таблицы занимает 1 байт, поэтому таблица имен имеет размер 960 байт.

Теперь у нас достаточно информации, чтобы разобраться, почему NES имела разрешение 256×240. Каждый тайл имеет ширину 8 пикселей и высоту 8 пикселей. Если таблица имен представляет фон экрана и имеет ширину 32 тайла, то получается 8 × 32 = 256. А высота экрана в пикселях определяется как 8 × 30 = 240.

В PPU всего 2 Кбайта памяти, чего хватает только на две таблицы имен (и их таблицы атрибутов). Эти две таблицы имен могут быть зеркальными, поэтому в общей сложности имеется четыре логические таблицы имен. На рис. 6.3 показаны таблицы имен для экрана заставки *BrickBreaker*.

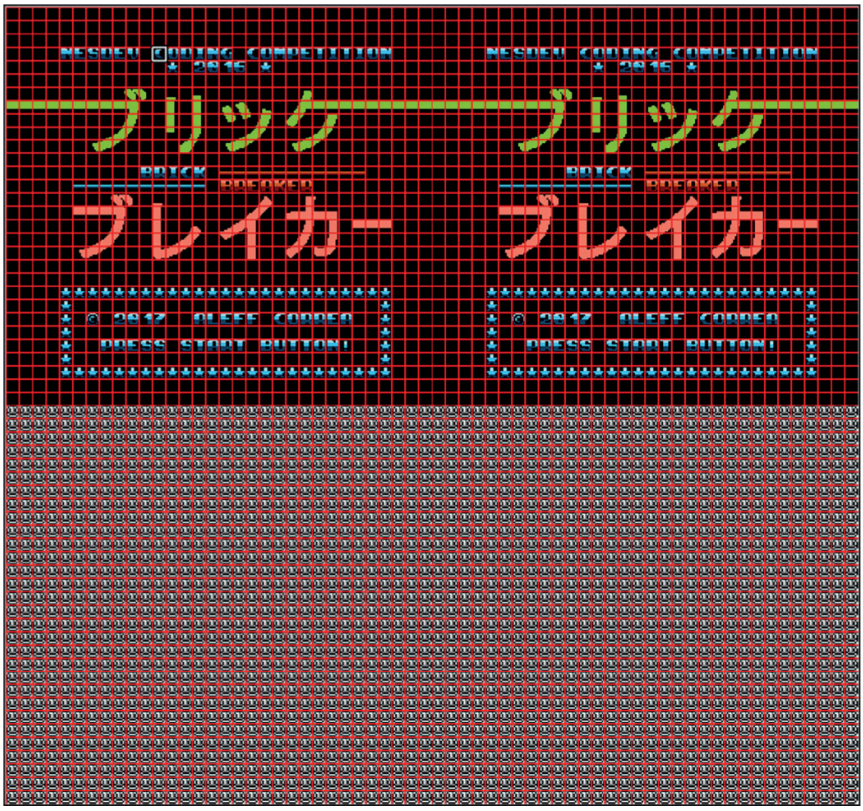


Рис. 6.3. Таблицы имен экрана заставки BrickBreaker

На рис. 6.3 только одна таблица имен содержит значимые данные. Можно заметить *горизонтальное зеркалирование*: правая половина изображения является копией левой половины.

Цветовые палитры и таблицы атрибутов

В PPU имеется *память палитры* (palette memory) для четырех различных палитр цветов фона. Вспомните из раздела «Таблицы шаблонов

и тайлы», что каждый тайл состоит из пикселей одного из четырех цветов, один из которых является цветом фона или прозрачным. Это означает, что только три цвета являются характерными для тайла. Таким образом, каждая палитра в памяти палитры PPU определяет набор из трех цветов (более подробно о памяти палитры мы поговорим в следующем разделе).

Чтобы указать, какая цветовая палитра применяется к каждому тайлу, за каждой таблицей имен следует *таблица атрибутов* (attribute table). Эта таблица имеет размер 8×8, и каждая запись в ней занимает всего 1 байт. Как может быть 960 тайлов и всего 64 записи в таблице атрибутов размером 1 байт? Каждая запись фактически определяет цвета для набора областей 2×2, и каждая из этих областей представляет собой сетку из 2×2 тайлов. Таким образом, каждая 1-байтовая запись в таблице атрибутов фактически охватывает 16 тайлов. Как это работает?

Каждые 2 бита этой 1-байтовой записи относятся к одной области, а 2 бита могут представлять до четырех значений. Каждое 2-битное значение является селектором между одной из четырех различных палитр цветов фона. Таким образом, каждая область из 2×2 тайлов на экране может иметь одну палитру из четырех возможных палитр фона. Это означает, что все четыре тайла в каждой области должны использовать одни и те же три цвета (и фон/прозрачность).

Это нелегкая задача. Художники, которые работали над графикой для NES, должны были рисовать значительные области экрана (области из четырех тайлов), используя только три цвета, отличных от цвета фона. А затем каждая из этих областей, в которой использовались только три цвета, отличных от цвета фона, должна была гармонировать с соседними областями, в которых могли использоваться другие трехцветные палитры.

Рисунки 6.4, 6.5 и 6.6, созданные для *https://nesdev.org*, демонстрируют таблицы атрибутов в действии (если вы читаете эту книгу в печатном виде, см. каталог рисунков в сопутствующем репозитории, где можно найти цветные версии изображений). Во-первых, на рис. 6.4 показан фон игры (*Thwaite*, автор Damian Yerrick), разбитый на области, для которых таблица атрибутов может выбирать цвета.

На рис. 6.5 показан фактический выбор палитры (какая палитра фона, 0–3) для каждой области из рис. 6.4.

Наконец, на рис. 6.6 показаны четыре палитры фона, из которых можно выбирать. Между палитрами есть некоторое перекрытие цветов, что позволяет различным областям экрана смешиваться друг с другом.

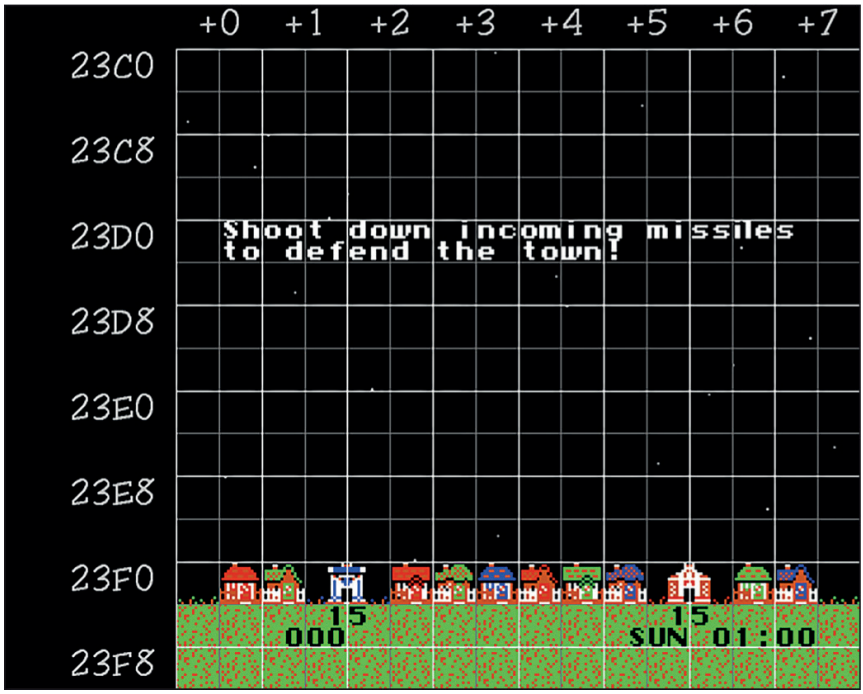


Рис. 6.4. Расположение таблицы атрибутов на экране в Thwaite

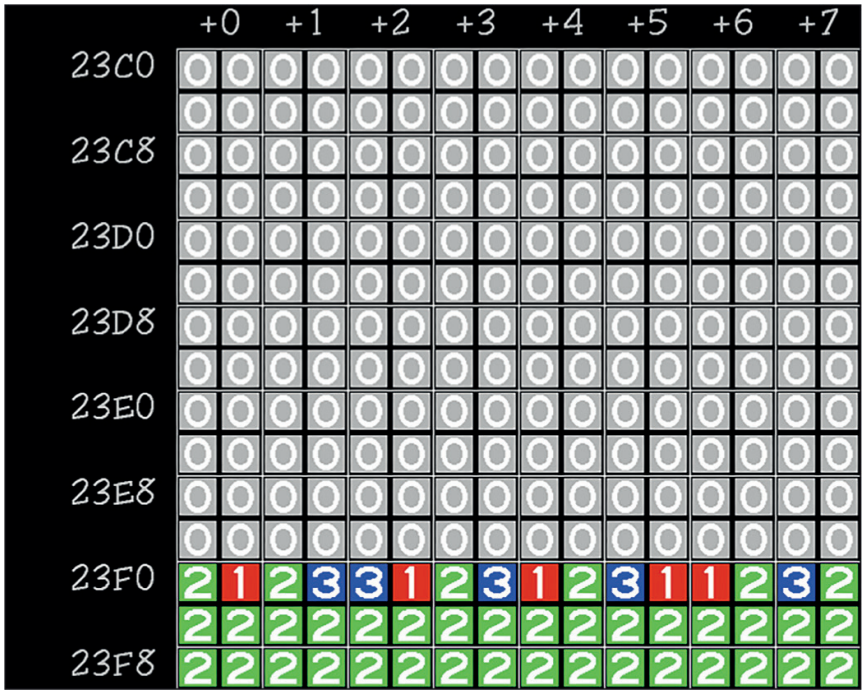


Рис. 6.5. Сопоставление цветовой палитры таблицы атрибутов в Thwaite

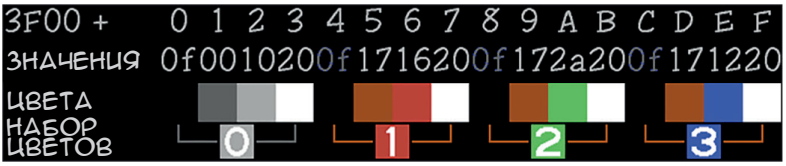


Рис. 6.6. Палитры цветов фона в Thwaite

Каждая таблица атрибутов занимает 64 байта, а каждая таблица имен – 960 байт. Таким образом, одна таблица имен и соответствующая ей таблица атрибутов занимают 960 + 64 = 1024 байта, или 1 Кбайт. Так 2 Кбайта оперативной памяти PPU заполняются двумя комбинациями таблиц шаблонов и таблиц атрибутов.

Память палитры

Память палитры PPU вмещает четыре палитры фона и четыре палитры спрайтов. Как было описано выше, палитра состоит из трех цветов (помимо фонового/прозрачного цвета). Она может использоваться для рисования четырехэлементной фоновой области экрана (см. предыдущий раздел о таблицах атрибутов) или спрайта. Другими словами, каждая четырехэлементная область фона может быть окрашена с помощью одной из четырех палитр, а каждый спрайт может быть окрашен с помощью одной из четырех палитр.

Палитра определяется с помощью 3 байт. Каждый байт определяет один из трех цветов палитры, хотя на самом деле для выбора цвета используются только 6 бит каждого байта. Поскольку 6 бит позволяют выбрать одно из 64 значений, это говорит о том, что художники NES могли работать только с 64 цветами. На практике 10 из этих 64 возможных цветов были по сути черными, поэтому на самом деле NES имела 54 цвета. На рис. 6.7 показаны эти цвета для NES стандарта NTSC. Для NES стандарта PAL цвета немного отличались (разные страны использовали разные видеостандарты – например, NTSC в Северной Америке и PAL в большей части Европы и Азии, что сказывалось на работе PPU NES).

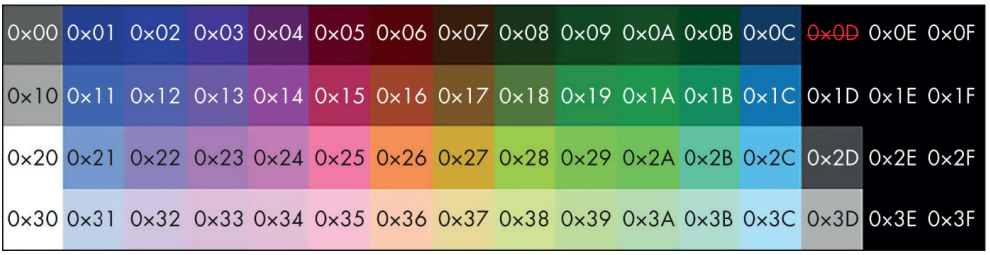


Рис. 6.7. Цвета, доступные на NES стандарта NTSC

Например, предположим, что область фона окрашена с использованием палитры фона 1, а палитра фона 1 определяет цвета 0x11, 0x0A

и 0x3D. Это означает, что вы можете окрасить эту область с использованием оттенков синего, зеленого и серого, а также цвета фона.

Память атрибутов объектов

Память атрибутов объектов (object Attribute Memory, OAM) хранит информацию о спрайтах на экране. Спрайты – это, по сути, движущиеся объекты в игре, например игрок, враги и снаряды. В NES поддерживается 64 спрайта одновременно, и для описания каждого из них используется 4 байта, поэтому объем памяти OAM составляет 256 байт.

Изображения для спрайтов берутся из того же CHR ROM на картридже, что и фоновые тайлы, то есть из таблиц шаблонов. Однако, в отличие от фона, спрайты не ограничены расположением тайлов; они могут быть нарисованы в любой точке экрана. Каждый спрайт также может быть перевернут по горизонтали или вертикали, размещен перед фоном или за ним, а также окрашен с использованием любой из четырех различных палитр. В табл. 6.5 кратко описаны 4 байта для каждого спрайта и их функции.

Таблица 6.5. Спецификация спрайтов в OAM

Байт №	Описание
0	Положение спрайта по оси *Y*
1	Индекс в таблице шаблонов, где находятся графические данные спрайта
2	Атрибуты спрайта, включая палитру (биты 0–1), передний или задний план (бит 5), горизонтальное отражение (бит 6) и вертикальное отражение (бит 7)
3	Положение спрайта по оси *X*

Почему байты для положения по оси *Y* и положения по оси *X* не находятся рядом друг с другом – это вопрос, который хотелось бы задать разработчикам NES. Возможно, это связано с физическим порядком подключения линий в PPU.

Создание фрейма и тайминги

В настоящей NES модуль PPU рисует экран от левого верхнего угла к правому нижнему, по одному пикселю за раз. Это отражает принцип работы «пушки» кинескопа в телевизорах с ЭЛТ-экраном, к которым подключалась NES. Ее электронный луч «стрелял» по экрану слева направо, по одной строке за раз, сверху вниз. NES в формате NTSC рисует весь экран 60 раз в секунду (60 кадров в секунду – frames per second, или 60 FPS). NES с поддержкой PAL рисует с меньшей скоростью – 50 FPS. *Кадр*, или *фрейм* (frame), – это момент, когда пушка кинескопа завершает свое движение по всему экрану. Как упоминалось ранее в этой главе, пушка также иногда временно выходит за пределы экрана – во время периодов hblank (между строками развертки) и vblank (между кадрами). Поэтому это наиболее безопасные моменты для программы NES, чтобы изменить память, из которой считываются пиксели.

Поскольку разрешение NES составляет 256×240, мы знаем, что на каждую строку развертки должно приходиться не менее 256 точек, а всего строк развертки – 240. Существует предварительно прорисованная строка развертки, которую мы назовем строкой развертки 0. Затем строки развертки с 1 по 240 являются «видимыми» строками развертки, которые представляют то, что программа фактически отображает на экране. Строки развертки с 241 по 261 находятся в фазе vblank. Таким образом, NMI, упомянутый ранее в этой главе, запускается в начале строки развертки 241, чтобы сообщить программе, что можно безопасно изменить память PPU. Точки с 0 по 255 представляют 256 видимых точек на каждой строке развертки. Точки с 256 по 340 являются «невидимыми» точками, во время которых происходит фаза hblank между строками развертки.

Каждая точка представляет один цикл PPU. Если посчитать, то 341 точка на строку развертки, умноженная на 262 строки развертки, означает, что для рисования каждого фрейма требуется 89 342 цикла PPU. Напомним, что на каждый цикл центрального процессора приходится 3 цикла PPU. Если разделить циклы PPU на 3 и округлить, то получим 29 781 цикл центрального процессора на фрейм. Центральный процессор в NES работает с частотой около 1,79 МГц, или 1 790 000 тактов в секунду. Если разделить 1 790 000 на 29 781, получим число, близкое к 60. Это и есть 60 кадров в секунду!

Чтобы создать эмулятор NES с точной синхронизацией циклов, важно понимать, как PPU определяет цвет для каждого пикселя по отдельности. Поскольку мы используем более простой, но менее точный подход, рисуя по одному фрейму за раз, мы можем опустить эти детали, так как они выходят за рамки нашего проекта. Что мы знаем на основе только что обсужденного времени, так это то, что в какой-то момент каждые 89 342 цикла PPU нам нужно нарисовать весь фрейм. Для меня логичным моментом для этого является время, когда видимые линии развертки закончены. В коде, который мы рассмотрим совсем скоро, вы увидите, что весь фон и все спрайты рисуются одновременно в момент сканирования линии 240, точки 256 (последняя видимая точка в последней видимой линии сканирования). Наш упрощенный рендер не выполняет рисование, кроме как во время этого цикла PPU, один раз за фрейм.

Реализация PPU

Реализация нашего PPU начинается с некоторых важных констант, включая различные размеры памяти, разрешение экрана и полную палитру доступных цветов:

NESEmulator/ppu.py

```
from array import array
from NESEmulator.rom import ROM
```

```
import numpy as np

SPR_RAM_SIZE = 256
NAMETABLE_SIZE = 2048
PALETTE_SIZE = 32
NES_WIDTH = 256
NES_HEIGHT = 240
NES_PALETTE = [0x7C7C7C, 0x0000FC, 0x0000BC, 0x4428BC, 0x940084, 0xA80020,
               0xA81000, 0x881400, 0x503000, 0x007800, 0x006800, 0x005800,
               0x004058, 0x000000, 0x000000, 0x000000, 0xBCBCBC, 0x0078F8,
               0x0058F8, 0x6844FC, 0xD800CC, 0xE40058, 0xF83800, 0xE45C10,
               0xAC7C00, 0x00B800, 0x00A800, 0x00A844, 0x008888, 0x000000,
               0x000000, 0x000000, 0xF8F8F8, 0x3CBCFC, 0x6888FC, 0x9878F8,
               0xF878F8, 0xF85898, 0xF87858, 0xFCA044, 0xF8B800, 0xB8F818,
               0x58D854, 0x58F898, 0x00E8D8, 0x787878, 0x000000, 0x000000,
               0xFCFCFC, 0xA4E4FC, 0xB8B8F8, 0xD8B8F8, 0xF8B8F8, 0xF8A4C0,
               0xF0D0B0, 0xFCE0A8, 0xF8D878, 0xD8F878, 0xB8F8B8, 0xB8F8D8,
               0x00FCFC, 0xF8D8F8, 0x000000, 0x000000]
```

Цвета задаются в шестнадцатеричных значениях RGB. Разные веб-сайты предлагают слегка отличающиеся значения цветов, и между различными вариантами аппаратного обеспечения NES (например, NTSC и PAL) действительно были различия в этих цветах. Поэтому одна и та же игра может иметь несколько отличающиеся цвета при запуске на различном оборудовании NES или различных эмуляторах.

Класс `PPU` имеет переменные экземпляра для памяти спрайтов (OAM), памяти таблиц имен и памяти палитры:

```
class PPU:
    def __init__(self, rom: ROM):
        self.rom = rom
        # Память PPU
        self.spr = array('B', [0] * SPR_RAM_SIZE) # RAM спрайта
        self.nametables = array('B', [0] * NAMETABLE_SIZE) # RAM таблицы имен
        self.palette = array('B', [0] * PALETTE_SIZE) # RAM палитры
```

Остальная часть конструктора класса `PPU` устанавливает значения по умолчанию для различных регистров PPU, настраиваемых программой, и многих вспомогательных переменных (мы более подробно рассмотрим назначение некоторых из них по мере работы над реализацией):

```
    # Регистры
    self.addr = 0 # main PPU address register
    self.addr_write_latch = False
    self.status = 0
    self.spr_address = 0
    # Переменные, управляемые регистрами контроля PPU
    self.nametable_address = 0
```

```
    self.address_increment = 1
    self.spr_pattern_table_address = 0
    self.background_pattern_table_address = 0
    self.generate_nmi = False
    self.show_background = False
    self.show_sprites = False
    self.left_8_sprite_show = False
    self.left_8_background_show = False
    # Внутренние вспомогательные переменные
    self.buffer2007 = 0
    self.scanline = 0
    self.cycle = 0
    # Пиксели для экрана
    self.display_buffer = np.zeros((NES_WIDTH, NES_HEIGHT), dtype=np.uint32)
```

Далее у нас есть метод рендеринга `step()`:

```
def step(self):
    # Наш упрощенный PPU рисует только один фрейм за раз
    if (self.scanline == 240) and (self.cycle == 256):
        if self.show_background:
            self.draw_background()
        if self.show_sprites:
            self.draw_sprites(False)
    if (self.scanline == 241) and (self.cycle == 1):
        self.status |= 0b10000000 # set vblank
    if (self.scanline == 261) and (self.cycle == 1):
        # Vblank выключен, очистка спрайта нуль, очистка переполнения спрайта
        self.status |= 0b00011111

    self.cycle += 1
    if self.cycle > 340:
        self.cycle = 0
        self.scanline += 1
        if self.scanline > 261:
            self.scanline = 0
```

Каждый вызов `step()` из главного цикла эмулятора представляет один цикл PPU. Поскольку наша стратегия заключается в рисовании всего сразу за один фрейм, а не в точности до пикселя или сканирующей линии, `step()` является чрезвычайно простым. Он просто рисует фон и спрайты сразу в конце каждой видимой части фрейма. Он также устанавливает регистр состояния при начале и конце vblank, чтобы центральный процессор мог координировать работу с PPU. Наконец, этот метод выполняет некоторые учетные операции, чтобы отслеживать текущую строку развертки и текущий цикл на каждой строке развертки.

Основная работа выполняется методами `draw_background()` и `draw_sprites()`, которые вызываются `step()`. Далее мы рассмотрим эти методы.

Рисование фона

Начнем метод `draw_background()` с вычисления адреса таблицы атрибутов:

```
def draw_background(self):
    attribute_table_address = self.nametable_address + 960
```

Таблица атрибутов всегда находится сразу после таблицы имен, и, как уже упоминалось ранее в этой главе, таблица имен имеет размер 960 байт. Я также упомянул ранее в этой главе, что таблица имен состоит из 960 байт, потому что она использует 1 байт для представления индекса каждого из 960 тайлов. Экран имеет ширину 32 тайла и высоту 30 тайлов, при этом каждый тайл представляет область 8×8 пикселей. Мы рисуем эти тайлы от левого верхнего угла до правого нижнего угла экрана, по одному ряду за раз, двигаясь слева направо:

```
    for y in range(30):
    for x in range(32):
        tile_address = self.nametable_address + y * 32 + x
        nametable_entry = self.read_memory(tile_address)
```

Каждый `tile_address` рассчитывается путем добавления базового `nametable_address` к смещению текущего тайла. Поскольку каждая строка состоит из 32 тайлов, мы умножаем строку (`y`) на 32 и добавляем компонент `x`, который можно рассматривать как столбец. Чтобы получить фактический индекс байта (`nametable_entry`), мы считываем байт памяти по адресу `tile_address`. Позже в этом методе мы будем использовать `nametable_entry` для извлечения пиксельного содержимого тайла из таблицы шаблонов.

Далее мы получаем запись таблицы атрибутов, соответствующую текущей записи в таблице имен:

```
        attrx = x // 4
        attry = y // 4
        attribute_address = attribute_table_address + attry * 8 + attrx
        attribute_entry = self.read_memory(attribute_address)
```

Поскольку каждая запись в таблице атрибутов 8×8 соответствует 16 тайлам, мы делим как `x`, так и `y` на 4, чтобы получить запись атрибута, связанную с рассматриваемым тайлом (чтобы лучше понять этот код, см. раздел «Цветовые палитры и таблицы атрибутов»). Поскольку каждая запись `attribute_entry` относится к четырем областям тайлов 2×2 (то есть в общей сложности к 16 тайлам), нам нужно перейти к конкретной области тайла:

```
block = (y & 0x02) | ((x & 0x02) >> 1)
attribute_bits = 0
if block == 0:
    attribute_bits = (attribute_entry & 0b00000011) << 2
elif block == 1:
    attribute_bits = (attribute_entry & 0b00001100)
elif block == 2:
    attribute_bits = (attribute_entry & 0b00110000) >> 2
elif block == 3:
    attribute_bits = (attribute_entry & 0b11000000) >> 4
else:
    print("Invalid block")
```

Размер `attribute_entry` составляет 1 байт, и каждые 2 бита этого байта соответствуют отдельной области тайла. Переменная `block` представляет область тайла для текущего тайла; она может быть равна 0, 1, 2 или 3. В зависимости от значения `block` мы используем соответствующие битовые операции, чтобы извлечь два конкретных бита для этой области тайла из `attribute_entry` и сохранить их в `attribute_bits`.

Теперь нам нужно извлечь отдельные пиксели каждого тайла из таблицы шаблонов:

```
for fine_y in range(8):
    low_order = self.read_memory(self.background_pattern_table_address +
                                 nametable_entry * 16 + fine_y)
    high_order = self.read_memory(self.background_pattern_table_address +
                                  nametable_entry * 16 + 8 + fine_y)
    for fine_x in range(8):
        pixel = ((low_order >> (7 - fine_x)) & 1) | (
                    ((high_order >> (7 - fine_x)) & 1) << 1) | attribute_bits
```

Вспомним из раздела «Таблицы шаблонов и плитки», что каждая таблица шаблонов состоит из 16-байтовых тайлов. Поэтому, чтобы вычислить адрес тайла, нам нужно умножить его индекс (`nametable_entry`) на 16 и прибавить его к `background_pattern_table_address`. Кроме того, тайлы таблицы шаблонов разделены на две битовые плоскости, где каждый цвет занимает 2 бита, а каждый из этих 2 бит находится на расстоянии 8 байт в отдельных плоскостях (см. рис. 6.2).

Наша стратегия заключается в чтении 2 байт, одного для плоскости `low_order` и одного для плоскости `high_order`. Каждый байт содержит половину пиксельных записей для одной строки тайла. Мы используем `fine_y` для представления каждой строки тайла, а затем фокусируемся на отдельных битах с помощью `fine_x`, который представляет каждую колонку. Комбинация битов из каждой плоскости с `attribute_bits` дает адрес в памяти палитры, где хранится цвет текущего пикселя.

Наконец, мы рисуем каждый пиксель из тайла по одному в соответствующем месте экрана, используя заранее определенные цвета в NES_PALETTE:

```
    x_screen_loc = x * 8 + fine_x
    y_screen_loc = y * 8 + fine_y
    transparent = ((pixel & 3) == 0)
    # Если фон прозрачный, используем первый цвет в палитре
    color = self.palette[0] if transparent else self.palette[pixel]
    self.display_buffer[x_screen_loc, y_screen_loc] = NES_PALETTE[color]
```

Установка пикселей для экрана означает установку значений в `display_buffer`, который является массивом NumPy, поскольку именно его принимает Pygame.

Рисование спрайтов

Рисование спрайтов имеет некоторое сходство с рисованием фоновых тайлов. Однако вместо чтения из таблицы имен мы читаем из OAM (`self.spr`). Каждая запись спрайта в памяти занимает 4 байта, представляя положение спрайта по оси *y*, индекс таблицы шаблонов, атрибуты и положение по оси *x* (см. табл. 6.5). В OAM может храниться до 64 записей спрайтов. Если положение по оси *y* равно 0xFF, то запись не используется. Мы запускаем `draw_sprites()`, перемещая по 4 байта за раз по OAM, чтобы найти все действительные записи:

```
def draw_sprites(self, background_transparent: bool):
    for i in range(SPR_RAM_SIZE - 4, -4, -4):
        y_position = self.spr[i]
    if y_position == 0xFF: # 0xFF is a marker for no sprite data
        continue
    background_sprite = bool((self.spr[i + 2] >> 5) & 1)
    x_position = self.spr[i + 3]
```

Мы извлекаем координаты *y* и *x* каждого действительного спрайта. Мы также смотрим на бит 5 его атрибутов, чтобы определить, является ли он фоновым спрайтом. Фоновые спрайты рисуются только в том случае, если фон прозрачный. Обратите внимание, что мы просматриваем память спрайтов в обратном порядке, потому что, как мы увидим в конце этого раздела, нулевой спрайт имеет особое значение.

Так же, как мы рисовали фоновые тайлы по одному пикселю за раз, мы делаем то же самое для спрайтов:

```
    for x in range(x_position, x_position + 8):
        if x >= NES_WIDTH:
            break
        for y in range(y_position, y_position + 8):
```

```
        if y >= NES_HEIGHT:
            break
```

Здесь x и y аналогичны fine_x и fine_y в draw_background(). Мы тщательно следим за тем, чтобы не рисовать пиксели, которые находятся за пределами экрана.

Еще одним атрибутом спрайта может быть возможность вертикального отражения (flip_y), которая определяется седьмым битом в байте атрибутов спрайта:

```
flip_y = bool((self.spr[i + 2] >> 7) & 1)
sprite_line = y - y_position
if flip_y:
    sprite_line = 7 - sprite_line
```

Если спрайт перевернут по вертикали, мы считываем его пиксели в обратном вертикальном порядке. Мы используем магическое число 7, потому что каждый спрайт имеет размер 8×8 пикселей.

Считывание фактических битов пикселей из таблицы шаблонов на основе индекса таблицы шаблонов очень похоже на работу, выполняемую в draw_background():

```
index = self.spr[i + 1]
bit0s_address = self.spr_pattern_table_address + (index * 16) +
                sprite_line
bit1s_address = self.spr_pattern_table_address + (index * 16) +
                sprite_line + 8
bit0s = self.read_memory(bit0s_address)
bit1s = self.read_memory(bit1s_address)
bit3and2 = ((self.spr[i + 2]) & 3) << 2
```

В данном случае я использовал несколько иную терминологию (bit0s_address и bit1s_address вместо low_order и high_order), поскольку посчитал, что другое наименование может быть более понятным для некоторых читателей. Биты цвета атрибута – это биты 0 и 1 в байте атрибута спрайта, и они хранятся в bit3and2 для конечного цвета.

Спрайты также можно отразить по горизонтали на основе бита 6 в байте атрибута:

```
flip_x = bool((self.spr[i + 2] >> 6) & 1)
x_loc = x - x_position # положение внутри спрайта
if not flip_x:
    x_loc = 7 - x_loc
```

Мы объединяем две битовые плоскости и пропускаем прорисовку прозрачных пикселей:

```
            bit1and0 = (((bit1s >> x_loc) & 1) << 1) | (
                    ((bit0s >> x_loc) & 1) << 0)
            if bit1and0 == 0: # прозрачные пиксели... пропуск
                continue
```

PPU отслеживает, сталкивается ли нулевой спрайт (первая запись в OAM) с какими-либо непрозрачными пикселями фона. Это называется *столкновением спрайта-ноль* (sprite-zero hit). Такая простая форма обнаружения столкновений реализована в данном случае:

```
    # Это не прозрачно. Значит ли это, что произошло столкновение спрайта-ноль?
    # Проверьте, что обрезка левых 8 пикселей не отключена.
    if (i == 0) and (not background_transparent) and (not (x < 8 and (
            not self.left_8_sprite_show or not self.left_8_background_show))
                and self.show_background and self.show_sprites):
        self.status |= 0b01000000
    # Необходимо сделать это после проверки спрайта-ноль, чтобы мы по-прежнему
    # учитывали фоновые
    # спрайты для проверок спрайта-ноль.
    if background_sprite and not background_transparent:
        continue # фоновый спрайт не должен рисоваться поверх непрозрачных пикселей
```

Когда происходит столкновение спрайта-ноль, мы отмечаем его в регистре состояния. В этом фрагменте кода мы также пропускаем рисование фоновых спрайтов, если фон не прозрачный. В PPU можно установить флаги, чтобы обрезать левые 8 пикселей фонового тайла или спрайта. Этот флаг также проверяется в данном разделе для гарантии, что при его включении не будет ошибочных столкновений спрайта-ноль.

Наконец, мы извлекаем цвет отдельного пикселя:

```
            color = bit3and2 | bit1and0
            color = self.read_memory(0x3F10 + color) # из палитры
            self.display_buffer[x, y] = NES_PALETTE[color]
```

Для получения цвета мы объединяем `bit3and2` с `bit1and0` и считываем из соответствующего места в памяти палитры. Затем мы помещаем этот пиксель на экран. Вместо прямого чтения из палитры мы используем в данном случае `read_memory()`, поскольку необходимо включить зеркальное отображение адресов.

Доступ к регистрам

В PPU есть несколько сопоставленных с памятью регистров, и при их чтении или записи имеются определенные технические особенности и нюансы. Мы решим эти задачи с помощью методов `read_register()` и `write_register()`. В `read_register()` первым делом обрабатываем адрес 0x2002, который предназначен для чтения регистра состояния:

```
def read_register(self, address: int) -> int:
    if address == 0x2002:
        self.addr_write_latch = False
        current = self.status
        self.status &= 0b01111111 # очистка vblank при чтении до 0x2002
        return current
```

Когда регистр состояния считывается, `self.addr_write_latch` устанавливается в `False`, что изменяет способ записи адресов в 0x2006 (будет описано позже). Кроме того, при считывании регистра состояния очищается vblank. Затем текущий `self.spr_address` в OAM можно прочитать через 0x2004:

```
    elif address == 0x2004:
    return self.spr[self.spr_address]
```

Память PPU по адресу `self.addr` может быть прочитана и записана через регистр 0x2007. Но она читается через буфер (`self.buffer2007`), при этом детали зависят от считываемого адреса:

```
elif address == 0x2007:
    if (self.addr % 0x4000) < 0x3F00:
        value = self.buffer2007
        self.buffer2007 = self.read_memory(self.addr)
    else:
        value = self.read_memory(self.addr)
        self.buffer2007 = self.read_memory(self.addr - 0x1000)
    # Каждое чтение в 0x2007 приводит к инкременту
    self.addr += self.address_increment
    return value
else:
    raise LookupError(f"Error: Unrecognized PPU read {address:X}")
```

Обратите внимание, что `self.address_increment` добавляется к `self.addr` после каждого чтения. Это позволяет последующим чтениям автоматически получать следующую запись либо на 1 байт, либо на 32 байта дальше.

В `write_register()` мы изменяем функционирование PPU посредством записи в различные регистры, отображаемые в памяти. Прежде всего это регистры 0x2000 и 0x2001, называемые *регистрами управления* (control registers). Они используются для изменения различных внутренних значений, которые мы уже видели в остальной части реализации PPU:

```
def write_register(self, address: int, value: int):
    if address == 0x2000: # Control1
        self.nametable_address = (0x2000 + (value & 0b00000011) * 0x400)
```

```
        self.address_increment = 32 if (value & 0b00000100) else 1
        self.spr_pattern_table_address = (((value & 0b00001000) >> 3) * 0x1000)
        self.background_pattern_table_address = (((value & 0b00010000) >> 4) * 0x1000)
        self.generate_nmi = bool(value & 0b10000000)
    elif address == 0x2001: # Control2
        self.show_background = bool(value & 0b00001000)
        self.show_sprites = bool(value & 0b00010000)
        self.left_8_background_show = bool(value & 0b00000010)
        self.left_8_sprite_show = bool(value & 0b00000100)
```

Далее мы обрабатываем регистры с 0x2003 по 0x2007:

```
        elif address == 0x2003:
            self.spr_address = value
        elif address == 0x2004:
            self.spr[self.spr_address] = value
            self.spr_address += 1
        elif address == 0x2005: # прокрутка
            pass
        elif address == 0x2006:
            # На основе https://wiki.nesdev.org/w/index.php/PPU_scrolling
            if not self.addr_write_latch: # первая запись
                self.addr = (self.addr & 0x00FF) | ((value & 0xFF) << 8)
            else: # вторая запись
                self.addr = (self.addr & 0xFF00) | (value & 0xFF)
            self.addr_write_latch = not self.addr_write_latch
        elif address == 0x2007:
            self.write_memory(self.addr, value)
            self.addr += self.address_increment
        else:
            raise LookupError(f"Error: Unrecognized PPU write {address:X}")
```

Регистр 0x2003 устанавливает `self.spr_address`. Регистр 0x2004 устанавливает значение в `self.spr_address`, а затем увеличивает `self.spr_address` на 1. Регистр 0x2005 является регистром прокрутки; мы не реализовали его в нашем простом PPU, но для полной реализации он необходим. В существующем виде наш эмулятор не будет работать с играми, требующими прокрутки. Регистр 0x2006 предназначен для изменения `self.addr`. Здесь вступает в игру `self.addr_write_latch`: нам нужен защелкивающий регистр, потому что `self .addr` имеет размер 16 бит (2 байта), но может записываться только по 1 байту за раз. Наконец, 0x2007 предназначен для записи в `self.addr`.

Доступ к памяти

Наша реализация PPU в основном готова, но нам нужны вспомогательные методы для чтения и записи в память PPU. Эти методы, `read_memory()` и `write_memory()`, очень похожи на их аналоги для центрального процессора:

```
def read_memory(self, address: int) -> int:
    address = address % 0x4000 # отображение >0x4000
    if address < 0x2000: # таблицы шаблонов
        return self.rom.read_cartridge(address)
    elif address < 0x3F00: # таблицы имен
        address = (address - 0x2000) % 0x1000 # 3000-3EFF is a mirror
        if self.rom.vertical_mirroring:
            address = address % 0x0800
        else: # горизонтальное отображение
            if (address >= 0x400) and (address < 0xC00):
                address = address - 0x400
            elif address >= 0xC00:
                address = address - 0x800
        return self.nametables[address]
    elif address < 0x4000: # память палитры
        address = (address - 0x3F00) % 0x20
        if (address > 0x0F) and ((address % 0x04) == 0):
            address = address - 0x10
        return self.palette[address]
    else:
        raise LookupError(f"Error: Unrecognized PPU read at {address:X}")

def write_memory(self, address: int, value: int):
    address = address % 0x4000 # отображение >0x4000
    if address < 0x2000: # таблицы шаблонов
        return self.rom.write_cartridge(address, value)
    elif address < 0x3F00: # таблицы имен
        address = (address - 0x2000) % 0x1000 # 3000-3EFF is a mirror
        if self.rom.vertical_mirroring:
            address = address % 0x0800
        else: # горизонтальное отображение
            if (address >= 0x400) and (address < 0xC00):
                address = address - 0x400
            elif address >= 0xC00:
                address = address - 0x800
        self.nametables[address] = value
    elif address < 0x4000: # память палитры
        address = (address - 0x3F00) % 0x20
        if (address > 0x0F) and ((address % 0x04) == 0):
            address = address - 0x10
        self.palette[address] = value
    else:
        raise LookupError(f"Error: Unrecognized PPU write at {address:X}")
```

В этих методах различные области памяти сопоставляются с соответствующими областями – таблицами шаблонов (которые фактически находятся на картридже), таблицами имен и т. п. Единственная сложность заключается в том, что таблицы имен и память палитры могут быть зеркально отражены. Мы решаем эту проблему с помощью оператора mod (%), как и в случае с центральным процессором.

Тестирование эмулятора

Для разработчиков эмуляторов NES было создано множество тестовых ПЗУ. Некоторые из них включены в репозиторий к этой книге. С их помощью можно протестировать как процессор 6502, так и PPU. Благодарю Шей Грина (Shay Green) и Кевина Хортона (Kevin Horton) за разработку этих тестов.

Наши 10 модульных тестов запускают эти ПЗУ, а затем проводят проверку правильности установки определенных значений в памяти виртуального NES, как указано создателями тестовых ПЗУ. Как и все тесты для книги, файл для этих модульных тестов находится в каталоге *tests* в корне репозитория исходного кода:

```
# tests/test_nesemulator.py
import unittest
from pathlib import Path
from NESEmulator.cpu import CPU
from NESEmulator.ppu import PPU
from NESEmulator.rom import ROM

class CPUTestCase(unittest.TestCase):
    def setUp(self) -> None:
        self.test_folder = (Path(__file__).resolve().parent.parent
                            / 'NESEmulator' / 'Tests')

    def test_nes_test(self):
        # Создание тестируемого оборудования
        rom = ROM(self.test_folder / "nestest" / "nestest.nes")
        ppu = PPU(rom)
        cpu = CPU(ppu, rom)
        # Установка тестов
        cpu.PC = 0xC000 # специальная стартовая локация для тестов
        with open(self.test_folder / "nestest" / "nestest.log") as f:
            correct_lines = f.readlines()
        log_line = 1
        # Проверка каждой строки лога на соответствие нашим собственным логам
        while log_line < 5260: # переход к первому неофициальному тесту операционного кода
            our_line = cpu.log()
            correct_line = correct_lines[log_line - 1]
            self.assertEqual(correct_line[0:14], our_line[0:14],
                             f"PC/Opcode doesn't match at line {log_line}")
            self.assertEqual(correct_line[48:73], our_line[48:73],
                             f"Registers don't match at line {log_line}")
            cpu.step()
            log_line += 1

    def test_blargg_instr_test_v5_basics(self):
        # Создание тестируемого оборудования
        rom = ROM(self.test_folder / "instr_test-v5" / "rom_singles" / "01-basics.nes")
        ppu = PPU(rom)
```

```
        cpu = CPU(ppu, rom)
        # Тесты выполняются до тех пор, пока 0x6000 равно 80, а затем 0x6000 является
        # кодом результата; 0 означает успех.
        rom.prg_ram[0] = 0x80
        while rom.prg_ram[0] == 0x80: # выполняется до первого теста неофициального
                                      # операционного кода
            cpu.step()
        self.assertEqual(0, rom.prg_ram[0],
                         f"Result code of basics test is {rom.prg_ram[0]} not 0")
        message = bytes(rom.prg_ram[4:]).decode("utf-8")
        print(message[0:message.index("\0")]) # сообщение заканчивается нулевым терминатором

    def test_blargg_instr_test_v5_implied(self):
        # Создание тестируемого оборудования
        rom = ROM(self.test_folder / "instr_test-v5" / "rom_singles" / "02-implied.nes")
        ppu = PPU(rom)
        cpu = CPU(ppu, rom)
        # Тесты выполняются до тех пор, пока 0x6000 равно 80, а затем 0x6000 является
        # кодом результата; 0 означает успех.
        rom.prg_ram[0] = 0x80
        while rom.prg_ram[0] == 0x80: # выполняется до первого теста неофициального
                                      # операционного кода
            cpu.step()
        self.assertEqual(0, rom.prg_ram[0],
                         f"Result code of implied test is {rom.prg_ram[0]} not 0")
        message = bytes(rom.prg_ram[4:]).decode("utf-8")
        print(message[0:message.index("\0")]) # сообщение заканчивается нулевым терминатором

    def test_blargg_instr_test_v5_branches(self):
        # Создание тестируемого оборудования
        rom = ROM(self.test_folder / "instr_test-v5" / "rom_singles" / "10-branches.nes")
        ppu = PPU(rom)
        cpu = CPU(ppu, rom)
        # Тесты выполняются до тех пор, пока 0x6000 равно 80, а затем 0x6000 является
        # кодом результата; 0 означает успех.
        rom.prg_ram[0] = 0x80
        while rom.prg_ram[0] == 0x80: # выполняется до первого теста неофициального
                                      # операционного кода
            cpu.step()
        self.assertEqual(0, rom.prg_ram[0],
                         f"Result code of branches test is {rom.prg_ram[0]} not 0")
        message = bytes(rom.prg_ram[4:]).decode("utf-8")
        print(message[0:message.index("\0")]) # сообщение заканчивается нулевым терминатором

    def test_blargg_instr_test_v5_stack(self):
        # Создание тестируемого оборудования
        rom = ROM(self.test_folder / "instr_test-v5" / "rom_singles" / "11-stack.nes")
        ppu = PPU(rom)
        cpu = CPU(ppu, rom)
        # Тесты выполняются до тех пор, пока 0x6000 равно 80, а затем 0x6000 является
        # кодом результата; 0 означает успех.
        rom.prg_ram[0] = 0x80
```

```
        while rom.prg_ram[0] == 0x80: # выполняется до первого теста неофициального
                                      # операционного кода
            cpu.step()
        self.assertEqual(0, rom.prg_ram[0],
                         f"Result code of stack test is {rom.prg_ram[0]} not 0")
        message = bytes(rom.prg_ram[4:]).decode("utf-8")
        print(message[0:message.index("\0")]) # сообщение заканчивается нулевым терминатором

    def test_blargg_instr_test_v5_jmp_jsr(self):
        # Создание тестируемого оборудования
        rom = ROM(self.test_folder / "instr_test-v5" / "rom_singles" / "12-jmp_jsr.nes")
        ppu = PPU(rom)
        cpu = CPU(ppu, rom)
        # Тесты выполняются до тех пор, пока 0x6000 равно 80, а затем 0x6000 является
        # кодом результата; 0 означает успех.
        rom.prg_ram[0] = 0x80
        while rom.prg_ram[0] == 0x80: # выполняется до первого теста неофициального
                                      # операционного кода
            cpu.step()
        self.assertEqual(0, rom.prg_ram[0],
                         f"Result code of jmp_jsr test is {rom.prg_ram[0]} not 0")
        message = bytes(rom.prg_ram[4:]).decode("utf-8")
        print(message[0:message.index("\0")]) # сообщение заканчивается нулевым терминатором

    def test_blargg_instr_test_v5_rts(self):
        # Создание тестируемого оборудования
        rom = ROM(self.test_folder / "instr_test-v5" / "rom_singles" / "13-rts.nes")
        ppu = PPU(rom)
        cpu = CPU(ppu, rom)
        # Тесты выполняются до тех пор, пока 0x6000 равно 80, а затем 0x6000 является
        # кодом результата; 0 означает успех.
        rom.prg_ram[0] = 0x80
        while rom.prg_ram[0] == 0x80: # выполняется до первого теста неофициального
                                      # операционного кода
            cpu.step()
        self.assertEqual(0, rom.prg_ram[0],
                         f"Result code of rts test is {rom.prg_ram[0]} not 0")
        message = bytes(rom.prg_ram[4:]).decode("utf-8")
        print(message[0:message.index("\0")]) # сообщение заканчивается нулевым терминатором

    def test_blargg_instr_test_v5_rti(self):
        # Создание тестируемого оборудования
        rom = ROM(self.test_folder / "instr_test-v5" / "rom_singles" / "14-rti.nes")
        ppu = PPU(rom)
        cpu = CPU(ppu, rom)
        # Тесты выполняются до тех пор, пока 0x6000 равно 80, а затем 0x6000 является
        # кодом результата; 0 означает успех.
        rom.prg_ram[0] = 0x80
        while rom.prg_ram[0] == 0x80: # выполняется до первого теста неофициального
                                      # операционного кода
            cpu.step()
        self.assertEqual(0, rom.prg_ram[0],
                         f"Result code of rti test is {rom.prg_ram[0]} not 0")
```

```
        message = bytes(rom.prg_ram[4:]).decode("utf-8")
        print(message[0:message.index("\0")]) # сообщение заканчивается нулевым терминатором

    def test_blargg_instr_test_v5_brk(self):
        # Создание тестируемого оборудования
        rom = ROM(self.test_folder / "instr_test-v5" / "rom_singles" / "15-brk.nes")
        ppu = PPU(rom)
        cpu = CPU(ppu, rom)
        # Тесты выполняются до тех пор, пока 0x6000 равно 80, а затем 0x6000 является
        # кодом результата; 0 означает успех.
        rom.prg_ram[0] = 0x80
        while rom.prg_ram[0] == 0x80: # выполняется до первого теста неофициального
                                      # операционного кода
            cpu.step()
        message = bytes(rom.prg_ram[4:]).decode("utf-8")
        print(message[0:message.index("\0")]) # сообщение заканчивается нулевым терминатором
        self.assertEqual(0, rom.prg_ram[0],
                         f"Result code of brk test is {rom.prg_ram[0]} not 0")

    def test_blargg_instr_test_v5_special(self):
        # Создание тестируемого оборудования
        rom = ROM(self.test_folder / "instr_test-v5" / "rom_singles" / "16-special.nes")
        ppu = PPU(rom)
        cpu = CPU(ppu, rom)
        # Тесты выполняются до тех пор, пока 0x6000 равно 80, а затем 0x6000 является
        # кодом результата; 0 означает успех.
        rom.prg_ram[0] = 0x80
        while rom.prg_ram[0] == 0x80: # выполняется до первого теста неофициального
                                      # операционного кода
            cpu.step()
        message = bytes(rom.prg_ram[4:]).decode("utf-8")
        print(message[0:message.index("\0")]) # сообщение заканчивается нулевым терминатором
        self.assertEqual(0, rom.prg_ram[0],
                         f"Result code of special test is {rom.prg_ram[0]} not 0")

if __name__ == "__main__":
    unittest.main()
```

При разработке эмулятора важно иметь такие автоматизированные тесты. Даже если вы считаете, что ваш процессор идеален, вы могли пропустить небольшую ошибку, которая выводит из строя всю программу. Вы также должны быть уверены, что изменение одной части эмулятора не повредит другую его часть.

Игровой процесс

Модульные тесты – это, конечно, хорошо, но настоящий тест нашего эмулятора – это возможность запускать на нем реальное программное обеспечение NES. По юридическим причинам мы не будем тести-

ровать коммерческое программное обеспечение в нашем эмуляторе NES. Из-за своей простоты наш эмулятор все равно не сможет запускать большую часть библиотеки NES. Вместо этого в репозитории исходного кода книги есть несколько игр с открытым исходным кодом или общественным доступом, которые наш эмулятор может запускать. Это настоящие игры в том смысле, что они могут запускаться на настоящей консоли NES.

Начнем с *BrickBreaker*, игры в стиле *Breakout* от Aleff Correa, о которой я упоминал ранее. Если у вас установлены Pygame и NumPy, вы можете запустить эту игру простой командой из домашнего каталога репозитория:

```
% python3 -m NESEmulator NESEmulator/Games/brix.nes
```

Выглядит неплохо (см. рис. 6.8).

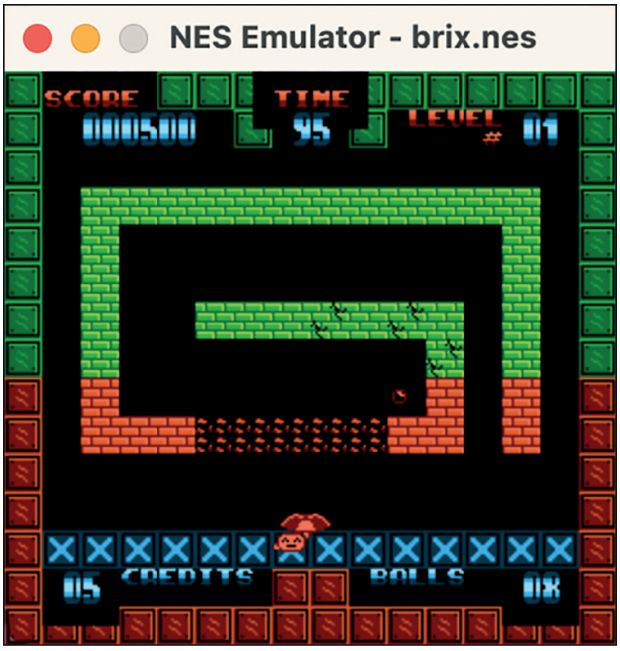


Рис. 6.8. BrickBreaker от Aleff Correa

Далее попробуем *Chase* от Shiru:

```
% python3 -m NESEmulator NESEmulator/Games/Chase.nes
```

Игра показана на рис. 6.9.



Рис. 6.9. Chase от Shiru

Наконец, давайте попробуем *Lan Master* от Shiru:

```
% python3 -m NESEmulator NESEmulator/Games/LanMaster.nes
```

Это головоломка, представленная на рис. 6.10.

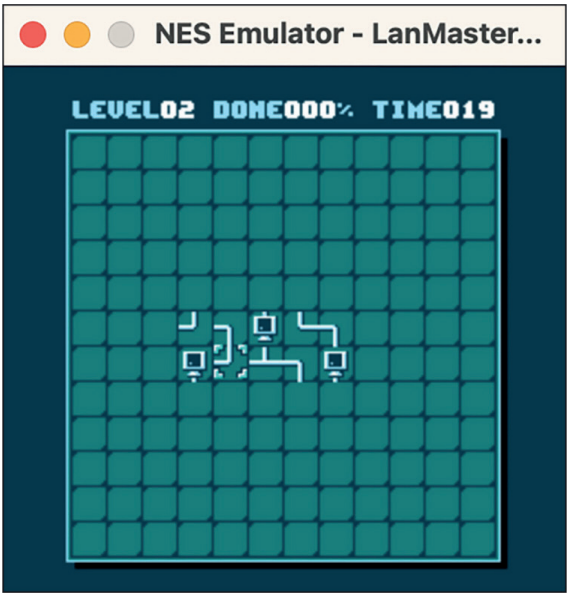


Рис. 6.10. Lan Master от Shiru

Игра *Lan Master* отлично работает на нашем эмуляторе, а две другие игры – нет. Почему? Возможно, вы заметили, что все они работают довольно медленно. Например, на моем M1 MacBook Air с CPython 3.13 они работают со скоростью где-то 14 кадров в секунду. Это примерно четверть скорости реальной NES. Наш эмулятор правильно запускает эти игры, просто очень медленно.

Какой из этого можно извлечь урок? Python, и в частности основная его версия CPython, работает медленно. В последние годы были предприняты попытки улучшить производительность CPython, однако по сравнению с большинством других реализаций языков программирования он по-прежнему работает очень медленно и не подходит для написания низкоуровневых программ, таких как эмуляторы. Чтобы писать более производительные программы, нужно пройти через несколько этапов: можно использовать определенные библиотеки, реализованные на языке более низкого уровня, использовать Cython, написать расширение на C или применить альтернативный интерпретатор Python, такой как PyPy.

Ускорение эмулятора с помощью чего-то вроде Cython я оставлю в качестве упражнения для читателя. Я уверен, что с помощью правильного решения вы сможете довести этот эмулятор NES до реальных 60 FPS NES.

Код в реальной жизни

Когда я начал свое путешествие в мир программирования эмуляторов, написав виртуальную машину CHIP-8 на Swift, моей настоящей мечтой было написать эмулятор для первой игровой консоли, которая у меня была в детстве, – NES. Два года спустя я достиг своей цели, написав базовый эмулятор NES без звука на языке C. В то время как на написание виртуальной машины CHIP-8 у меня ушло день или два, на эмулятор NES ушло около 30 дней, распределенных на год периодических исследований и программирования. Я обнаружил, что написание части, отвечающей за работу процессора, было довольно простым, но написание пиксельно безупречного рендерера фона оказалось гораздо более сложной задачей. В конце концов, я перенес рендерер фона PPU из отличного эмулятора NES Майкла Фоглемана (Michael Fogleman)[1] из Go в C и объединил его с моим собственным кодом рендеринга спрайтов.

Если на написание эмулятора NES у меня ушло около 30 дней, а на CHIP-8 VM – около двух, значит ли это, что проект был в 15 раз сложнее? Я так не думаю. Моя задача при написании PPU заключалась в том, чтобы одновременно держать в голове

[1] См. *https://github.com/fogleman/nes*.

все технические детали. Я был слишком сосредоточен на написании рендерера с идеальной точностью до пикселя, тогда как мне следовало начать с рендерера по фреймам, как мы делали в этой главе. В то время также не было хороших учебных пособий, хотя документация на NesDev была неоценимой. В подростковом возрасте я впервые захотел написать эмулятор NES, поэтому его завершение стало сбывшейся мечтой, даже если он был довольно простым и не имел звука (позже я добавил звук, а также больше мапперов, чем только NROM).

В этой главе я постарался предоставить учебник, который мне бы хотелось иметь во время разработки своего эмулятора NES. Я знал, что пиксельно совершенная визуализация будет слишком сложной и переполненной различными внутренними регистрами для начинающего разработчика эмуляторов, поэтому я вернулся и переписал PPU своего эмулятора NES на C так просто, как только было возможно. Именно этот рендер я перенес на Python для данной главы.

После завершения эмулятора NES я приступил к написанию эмулятора для оригинального IBM PC. Это был значительно более сложный проект, в основном потому, что Intel 8088/8086 гораздо сложнее, чем MOS 6502. В этом проекте меня подвело то, что я не написал автоматические тесты на самом раннем этапе. В конце концов, я заставил его работать на базовом уровне, но мне следовало написать тесты гораздо раньше. Чем сложнее микропроцессор, который вы эмулируете, тем больше вам нужны автоматические тесты на как можно более раннем этапе.

Практические приложения

Эмуляторы, вероятно, чаще всего используются для запуска видеоигр для систем, которые больше не производятся, но они также широко применялись на важных этапах развития компьютерной техники. Например, когда Билл Гейтс (Bill Gates) и Пол Аллен (Paul Allen) основали компанию Microsoft в 1975 году, написав интерпретатор BASIC для Altair 8800, как упоминалось в главе 2, у них на самом деле не было в распоряжении Altair 8800. Вместо этого они написали эмулятор для микропроцессора Intel 8080 Altair на одном из мини-компьютеров в Гарварде, где Гейтс учился в колледже[1].

Apple трижды меняла семейство микропроцессоров, используемых в линейке компьютеров Macintosh: с Motorola 68K на Motorola/IBM PowerPC, с PowerPC на Intel x86 и, наконец, с Intel x86 на собственный

[1] James Wallace and Jim Erickson, Hard Drive: Bill Gates and the Making of the Microsoft Empire (HarperBusiness, 1993).

процессор Apple Silicon, основанный на архитектуре ARM. Означает ли это, что Apple приходилось несколько раз перекомпилировать или переписывать все свое программное обеспечение? В долгосрочной перспективе – да, но в краткосрочной перспективе, во время перехода, Apple предоставляла эмуляторы. Mac с процессором PowerPC могли запускать программное обеспечение для Mac с процессором 68K, Mac с процессором Intel (вначале) могли запускать программное обеспечение для Mac с процессором PowerPC, а Mac с процессором Apple Silicon также могут запускать программное обеспечение для Mac с процессором Intel. Apple – потрясающий разработчик эмуляторов. Фактически PowerPC Mac могли запускать программное обеспечение 68K быстрее, чем 68K Mac. То же самое иногда было верно и для Apple Silicon Mac, запускающих программное обеспечение Intel.

Эмуляторы также важны для сохранения программного обеспечения. Что происходит, когда очень сложно получить оригинальное оборудование, для которого было написано программное обеспечение? В таких случаях эмулятор может быть единственным вариантом. С другой стороны, эмуляторы иногда используются на этапе проектирования новой вычислительной платформы. До того, как платформа появится, разработчики могут использовать эмулятор, чтобы смоделировать ее работу и помочь разобраться в ее функциях в реалистичной среде.

Наконец, написание эмуляторов очень познавательно. Это один из лучших способов обучения тому, как работают компьютеры на низком уровне, как, я надеюсь, вы обнаружили в данной главе.

Упражнения

1. Попытайтесь довести производительность нашего эмулятора до уровня реального NES, то есть до 60 FPS. Вероятно, вам понадобится что-нибудь вроде Cython или Python в сочетании с низкоуровневым языком, таким как C или Rust, через расширение. Начиная с CPython 3.13 и микропроцессоров эпохи 2025 года будет практически невозможно заставить чистый Python работать со скоростью 60 FPS.
2. Добавьте поддержку прокрутки в наш эмулятор, используя документацию по адресу *https://nesdev.org*.
3. Реализуйте другой маппер. В настоящее время наш эмулятор реализует только NROM, самый базовый маппер. Двумя другими популярными мапперами являются MMC1 и UxROM.
4. И теперь самое сложное задание: попробуйте написать APU для нашего эмулятора, чтобы можно было играть в игры со звуком.

ЧАСТЬ IV

ОЧЕНЬ ПРОСТОЕ МАШИННОЕ ОБУЧЕНИЕ

7

КЛАССИФИКАЦИЯ С ПОМОЩЬЮ К-БЛИЖАЙШИХ СОСЕДЕЙ



В этой главе рассматривается принцип *k-ближайших соседей* (k-nearest neighbors, *KNN*) – чрезвычайно простой алгоритм машинного обучения, который может оказаться достаточно эффективным для некоторых приложений. Впервые разработанный в 1950-х и 1960-х годах[1], KNN применяется как для *классификации* (определения категории, к которой относится объект), так и для *регрессии* (прогнозирования значения). Для читателей, которых отпугивает сложность машинного обучения, алгоритм KNN открывает доступную, но в то же время реальную отправную точку в этой области. В этой главе мы будем использовать KNN для решения двух задач классификации с высокой степенью точности: распознавания различных видов рыб и распознавания рукописных цифр. Затем, в главе 8, мы рассмотрим применение KNN для решения некоторых соответствующих задач регрессии.

Становление машинного обучения

Еще в 1950-х годах традиционные исследования в области искусственного интеллекта (ИИ) были в основном сосредоточены на использовании алгоритмов для моделирования человеческого интеллекта.

[1] Thomas Cover and Peter E. Hart, «Nearest Neighbor Pattern Classification», IEEE Transactions on Information Theory 13, № 1 (January 1967): 21–27.

Однако с 1990-х годов, и особенно с появлением нейронных сетей, подкрепленных вычислениями на графических процессорах в 2000-х годах, большая часть исследований и продуктов в области ИИ переключилась на область машинного обучения, которая использует большие наборы данных для обучения моделей, способных принимать решения без необходимости обращаться к человеческим методам решения проблем. Чтобы понять эту перемену, представьте себе разницу между шахматной программой, которая оценивает позицию на основе хорошо понятных эвристических правил, разработанных гроссмейстерами, и шахматной программой, которая оценивает позицию на основе автоматически настроенных весов, полученных в результате статистического анализа миллионов партий. И действительно, машинное обучение в значительной степени основано на статистике.

Все известные нам интересные приложения машинного обучения, включая LLM, распознавание изображений и цифровые помощники, построены с использованием сложных многослойных нейронных сетей, обученных посредством графических процессоров или специализированных нейронных процессоров. Это называется *глубоким обучением* (deep learning). Чтобы запрограммировать среду глубокого обучения с нуля, необходимо хорошо разбираться как в математическом анализе, так и в статистике. Даже если вы избежите части этой работы с помощью библиотеки, вам все равно понадобится огромный набор данных, который зачастую не так просто получить, и мощное аппаратное оборудование.

Из-за этих препятствий программисты, заинтересованные в освоении машинного обучения, иногда испытывают чувство испуга. Они боятся, что математика будет слишком сложной или что у них не хватит ресурсов для разработки интересующего их приложения. Более того, некоторые начинающие программисты предпочитают создавать свои проекты с нуля; они не хотят просто `pip install` для получения решения, которое не дает им понимания глубинных процессов.

Тем не менее, как вы увидите в этой главе, машинное обучение может быть очень доступным, если не погружаться сразу в пучину глубокого обучения. Вы можете запрограммировать алгоритм KNN с нуля и использовать его для решения реальных задач, и при этом вам не понадобятся математические знания выше средней школы для понимания того, что вы делаете. Единственное статистическое определение, необходимое для реализации KNN, – это представление о среднем (mean) значении (или среднем – average), а единственная формула, которая вам понадобится, – это теорема Пифагора для нахождения евклидова расстояния между двумя точками. Это не так уж и много, верно?

Примечание *Если вы хотите узнать больше о нейронных сетях, можете ознакомиться с главой 7 «Fairly Simple Neural Networks» моей предыдущей книги «Classic Computer Science Problems in Python» (Manning, 2019).*

Как работает KNN

Алгоритм KNN исходит из простого предположения: соседними точками данных, скорее всего, будут те точки данных, которые имеют с ней больше всего общего. Например, если я пытаюсь определить заболевание пациента, то, возможно, лучшими подсказками будут другие пациенты с такими же симптомами и такими же показателями жизненных функций. Кстати, именно поэтому вы, вероятно, не захотите, чтобы вас лечил врач, только что закончивший медицинский институт. Более опытные врачи могут использовать очевидную эвристику «Я уже видел других пациентов с такими симптомами» и дать вам более точные первоначальные рекомендации.

Другими словами, данные, которые наиболее близки к неизвестному значению, с наибольшей вероятностью могут подсказать нам это неизвестное значение. Допустим, вы являетесь автодилером и хотите определить, стоит ли тратить дополнительные средства на маркетинг для привлечения потенциального постоянного клиента путем отправки дополнительных рекламных писем. Клиент заполнил анкету удовлетворенности, оценив различные аспекты вашей деятельности. У вас есть много данных от предыдущих клиентов, которые заполняли ту же анкету, и вы в курсе, купили ли они в итоге у вас еще одну машину. Вы можете сравнить оценки этого потенциального постоянного клиента с оценками предыдущих клиентов, чтобы найти клиентов, которые наиболее похожи на него по характеру. Если предыдущие клиенты с похожими оценками в конечном итоге купили еще одну машину, вы знаете, что, скорее всего, стоит потратить деньги на отправку дополнительных маркетинговых материалов. По сути, это и есть анализ, который выполняет KNN: он позволяет предыдущим данным «голосовать» за вероятное значение, связанное с некоторыми новыми данными.

Давайте рассмотрим последний пример визуально в двух измерениях. На рис. 7.1 представлен вымышленный набор данных с оценками респондентов по результатам опроса дилеров на предмет «Удовлетворенность автомобилем» и «Удовлетворенность дилером». Крестики обозначают респондентов, которые приобрели другой автомобиль у дилера, треугольники – респондентов, которые этого не сделали, а круглые точки – новых респондентов.

Как мы классифицируем нового респондента опроса? Скорее всего, он будет покупать другой автомобиль или нет? Используя KNN, нам сначала нужно выбрать значение *k*, *метаэвристику* (meta-heuristic). Это просто количество соседей, которые будут рассматриваться. Если мы установим *k* равным 1, то будем рассматривать только ту точку данных, которая ближе всего к нашему неизвестному значению. На рис. 7.1, где наш респондент находится в точке (6, 8), следующая ближайшая точка данных для сравнения находится в точке (7, 8), и это

человек, который действительно купил еще один автомобиль. При *k*, равном 1, мы сделаем вывод, что стоит потратить деньги на отправку нашему новому респонденту дополнительных маркетинговых материалов.

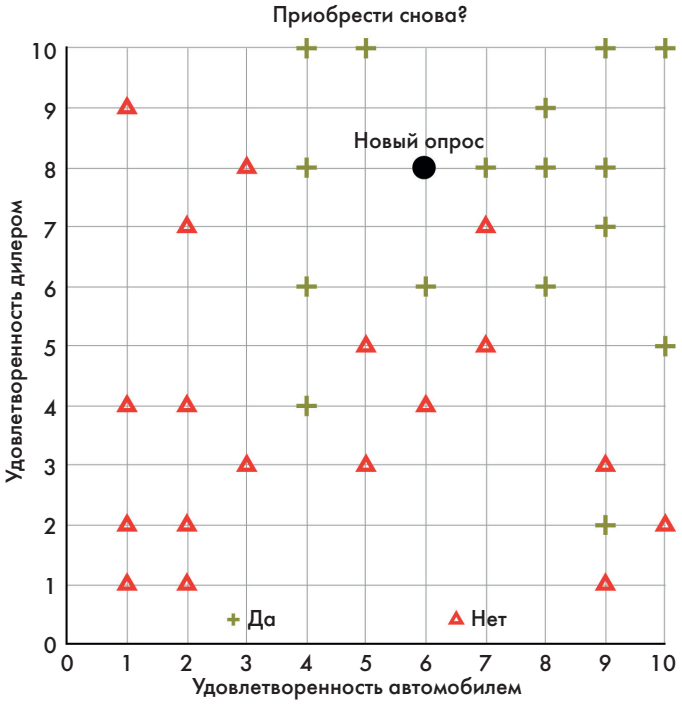


Рис. 7.1. Данные опроса респондентов автосалона

Но что, если *k* задано как число больше 1? Типичным решением в алгоритме KNN является «голосование» (vote). Например, если *k* в нашем сценарии равно 3, то можно видеть, что тремя ближайшими точками данных будут двое, которые купили новую машину, и один, который этого не сделал. Поскольку большинство купило еще одну машину, мы все равно пришли бы к выводу, что стоит потратить деньги на маркетинг для нового респондента опроса. Если *k* установлено на 2, у нас возникла бы проблема, поскольку один решил купить еще одну машину, а другой – нет. Тогда нам понадобился бы какой-то критерий для разрешения этой патовой ситуации.

Вот и все. Это весь алгоритм KNN для классификации. Мы рассматриваем *k* точек данных, близких к рассматриваемой точке данных, и позволяем им проголосовать за определение нашего неизвестного значения. Таким образом, мы можем обобщить алгоритм KNN для классификации следующим образом:

1. Выберите *k*, количество соседей для сравнения с неклассифицированной точкой данных.

2. Найдите *k* ближайших соседей точки данных.
3. Проведите голосование по классификации точки данных на основе классов *k* ближайших соседей.

Этот простой алгоритм требует трех пояснений: что значит быть «ближайшим» соседом? Как определяется победитель голосования? И какое значение *k* является правильным? На все эти вопросы могут быть разные ответы для разных случаев применения KNN.

«Ближайший» чаще всего определяется с помощью евклидова расстояния. Другими словами, если мы нарисуем прямые линии на рис. 7.1 между нашей неклассифицированной точкой (круглым кружком) и всеми другими точками данных на графике, то самые короткие линии определят ближайших соседей. Однако для некоторых приложений, которые не имеют числовых точек данных, может применяться что-то вроде *расстояния Хэмминга* (Hamming distance, заключается в подсчете различий). Тема функций определения расстояния, отличных от евклидова расстояния, выходит за рамки этой главы.

Голосование обычно означает определение класса, к которому принадлежит большинство ближайших соседей. Однако вам понадобится какой-то критерий для разрешения равенства голосов, даже если вы будете придерживаться нечетных чисел для *k*. Это связано с тем, что во многих приложениях имеется более двух классов. Например, что делать, если у вас есть три класса, а *k* установлено на 5? В результате вы можете получить двух ближайших соседей, принадлежащих к классу A, двух, принадлежащих к классу B, и одного, принадлежащего к классу C. Должна ли рассматриваемая точка данных быть классифицирована как A или B?

Определение правильного значения *k* на самом деле довольно просто. Если у нас нет специальных знаний в данной области, которые говорят об обратном, мы должны использовать значение, которое оказалось наиболее точным при тестировании. Мы можем протестировать KNN с несколькими различными значениями *k* с конкретным набором данных и набором тестовых точек и посмотреть, какое значение является наиболее правильным.

Это основные принципы классификации с помощью KNN. Конечно, помимо этого простого описания, существует множество вариантов и усовершенствований, но вы уже знаете достаточно, чтобы реализовать KNN!

Реализация классификации с помощью KNN

Как и сам алгоритм, наш код для реализации KNN будет довольно простым. Но прежде чем перейти к алгоритму, нам нужен общий тип для представления точки данных. Мы создадим класс `DataPoint`, который является протоколом, то есть у этого типа не будет экземпляров, а будут только его подклассы. Это абстрактный шаблон для описания

функциональности, которой должен обладать более конкретный тип точки данных:

KNN/knn.py

```
from pathlib import Path
import csv
from typing import Protocol, Self
from collections import Counter
import numpy as np

class DataPoint(Protocol):
    kind: str

    @classmethod
    def from_string_data(cls, data: list[str]) -> Self: ...

    def distance(self, other: Self) -> float: ...
```

Наш протокол определяет, что точка данных должна иметь атрибут `kind` (или *класс*, если говорить языком классификации), метод `from_string_data()` для преобразования строки из файла CSV (значения, разделенные запятыми) в экземпляр класса, а также метод `distance()` для нахождения расстояния между двумя точками данных одного и того же типа. Мы создадим подклассы `DataPoint` для двух конкретных наборов данных, с которыми будем работать в этой главе.

Наша основная реализация KNN осуществляется через класс, который, что неудивительно, называется `KNN`:

```
class KNN[DP: DataPoint]:
    def __init__(self, data_point_type: type[DP], file_path: str | Path,
                 has_header: bool = True) -> None:
        self.data_point_type = data_point_type
        self.data_points = []
        self._read_csv(file_path, has_header)

    # Считывание файла CSV и возврат списка точек данных
    def _read_csv(self, file_path: str | Path, has_header: bool) -> None:
        with open(file_path, 'r') as f:
            reader = csv.reader(f)
            if has_header:
                _ = next(reader)
            for row in reader:
                self.data_points.append(
                    self.data_point_type.from_string_data(row))
```

Синтаксис подсказки типа `class KNN[DP: DataPoint]:` говорит о том, что общий тип `DP` связан с `KNN` и что `DP` должен быть подклассом `DataPoint`. Мы будем загружать все наши наборы данных из CSV-файлов. Метод `_read_csv()` нашего класса `KNN` использует встроенный модуль

Python csv для загрузки этих файлов. Каждая строка в CSV-файле используется для инициализации одного из наших подклассов DataPoint с помощью метода класса from_string_data(). Мы вернемся к особенностям CSV-файлов чуть позже, когда рассмотрим наш первый набор данных.

Теперь, когда у нас есть набор данных, загруженный в класс KNN, мы готовы к реализации фактического алгоритма KNN. С чего начать? Имея точку, которую мы хотим классифицировать, первым делом нам нужно определить ее *k* ближайших соседей. Все наши точки данных имеют встроенные методы distance(), поэтому мы можем просто вычислить расстояние от нашей неклассифицированной точки данных до каждой точки данных в наборе данных и найти *k* ближайших:

```
    def nearest(self, k: int, data_point: DP) -> list[DP]:
        return sorted(self.data_points, key=data_point.distance)[:k]
```

Да, просто одна строчка. Мы просто используем результаты метода distance() для сортировки всех точек данных, а затем сохраняем *k* наименьших значений (точки с наименьшим расстоянием от data_point). Суть алгоритма KNN действительно настолько проста.

Является ли это наиболее эффективным способом поиска ближайших соседей? Нет. Сортировка – это операция $O(n \log n)$, где n – это количество точек данных в наборе данных. Если набор данных очень большой, это может стать серьезным препятствием. Мы могли бы немного улучшить ситуацию, написав код для ручной оценки расстояний всех точек данных с использованием вспомогательной структуры данных, чтобы сохранить только *k* наименьших. Чтобы еще больше повысить производительность, нам может понадобиться более сложная структура данных, чем несортированный список для хранения набора данных. Таким образом, здесь существует компромисс между производительностью алгоритма и сложностью структуры данных.

Другим вариантом было бы не проводить каждый раз поиск соседей по всему набору данных. Существуют различные подходы к ограничению поиска, либо путем предварительного вычисления более ограниченного подмножества точек данных, которые являются репрезентативными для всего набора, либо путем выборки набора данных во время поиска с помощью так называемого *приближенного поиска* (approximate search)[1]. Тем не менее наша простая техника сортировки достаточно оперативна для наших задач и вполне соответствует названию этой части книги «Очень простое машинное обучение».

Далее нам нужно провести «голосование». Для этого необходимо подсчитать количество представителей каждого класса (или kind) в ближайших соседях и вернуть класс, в котором их больше всего:

[1] Shichao Zhang, «Challenges in KNN Classification», IEEE Transactions on Knowledge and Data Engineering 34, № 10 (октябрь 2022): 4663–4675, *https://doi.org/10.1109/tkde.2021.3049250*.

```
    def classify(self, k: int, data_point: DP) -> str:
        neighbors = self.nearest(k, data_point)
        return Counter(neighbor.kind for neighbor in neighbors).most_common(1)[0][0]
```

Сначала мы находим `neighbors`. Затем используем встроенный в Python тип коллекции `Counter`, чтобы найти наиболее распространенный вид среди соседей. Вызов `most_common(1)` возвращает единственный наиболее распространенный элемент в `Counter`, а `[0][0]` указывает извлечь первый элемент из коллекции и взять его метку `kind`. Внутренняя структура `Counter` будет представлять собой пары ключ-значение, которые будут выглядеть примерно так: `[("amphibian", 3), ("reptile", 4), ("mammal", 1)]`. В этом примере строка найдет пару ключ-значение с наибольшим значением, `("reptile", 4)`, и вернет только ее ключ, `"reptile"`. Здесь мы не обрабатываем равные значения каким-либо систематическим образом – мы просто оставляем это на усмотрение счетчика. Опять же, вспомните название раздела.

Вот и все. Благодаря некоторым удобным процедурам стандартной библиотеки Python фактическая алгоритмическая работа, лежащая в основе KNN, на деле сводится к трем строкам кода между `nearest()` и `classify()`. Я же говорил, что это будет «очень просто». Теперь, когда алгоритм готов, давайте применим KNN к двум задачам классификации.

Классификация рыб

Предположим, вы работаете программистом в компании, которая производит устройства поиска рыбы для рыбаков. Оно состоит из камеры на конце шеста, который движется под лодкой. В него встроена функция распознавания изображений, поэтому когда рыба проплывает мимо подводной камеры, она может автоматически сфотографировать ее и распознать прямоугольник на фотографии, в котором находится рыба. Она также может оценить размеры рыбы на фотографии. Ваша задача – написать программный слой поверх этой системы распознавания изображений, который будет сообщать рыбаку, что это за рыба. Ведь не все виды рыб разрешены к вылову.

К счастью, у нас есть общедоступный набор данных, который поможет нам в этой задаче по классификации рыб. Изначально он был опубликован в финской статье Пекки Брофельдта (Pekka Brofeldt) 1917 года, которая на английском языке называлась Contribution to the Knowledge of Fish Stocks in Dangerous Lakes («Вклад в изучение рыбных запасов в опасных озерах»)[1]. В ней приведены размеры и вес

[1] Pekka Brofeldt, «Bidrag till kaennedom on fiskbestondet i vaara sjoear Laengelmaevesi», T. H. Jaervi: Finlands Fiskeriet Band 4, Meddelanden utgivna av fiskerifoereningen i Finland, Helsingfors, 1917.

159 видов озерной рыбы, классифицированных по видам (поэтому наша программа может работать только для этого одного озера). На рис. 7.2 показаны рыбы из нашего набора данных в двух измерениях: по высоте и ширине.

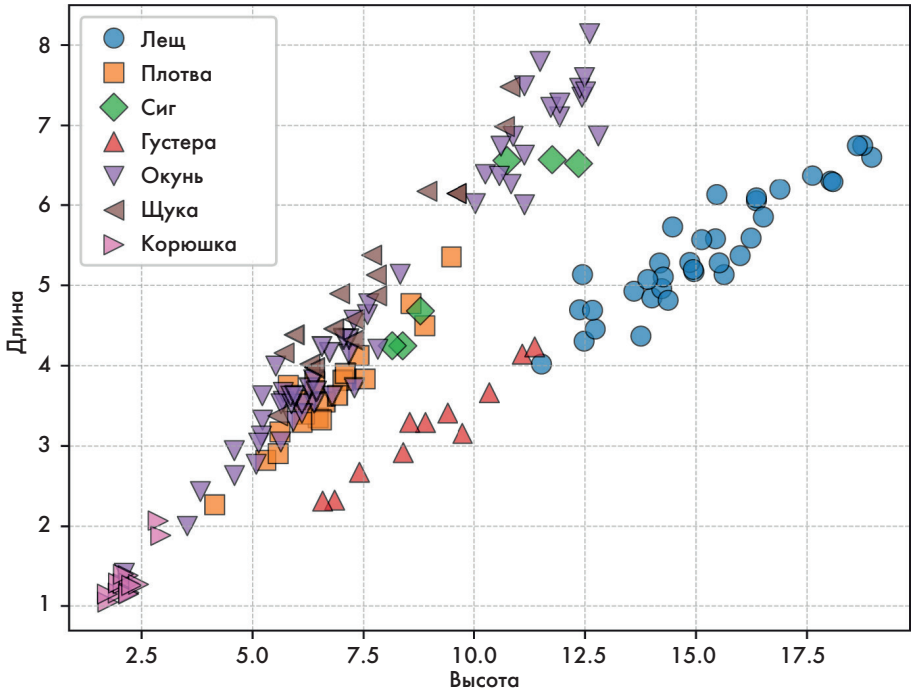


Рис. 7.2. Рыбы, классифицированные по видам и отображенные в двух измерениях: по высоте и ширине

Как и следовало ожидать, рыбы одного вида, как правило, имеют схожие размеры по высоте и ширине. Это хороший показатель того, что алгоритм KNN может быть полезен для решения данной задачи.

Давайте посмотрим, как выглядят исходные данные. Вот пример первых нескольких строк файла *fish.csv*:

```
Species,Weight,Length1,Length2,Length3,Height,Width
Bream,242,23.2,25.4,30,11.52,4.02
Bream,290,24,26.3,31.2,12.48,4.3056
Bream,340,23.9,26.5,31.1,12.3778,4.6961
Bream,363,26.3,29,33.5,12.73,4.4555
```

Первая строка – это заголовок, описывающий каждую колонку. Три размера длины, которые представляют расстояния от носа каждой рыбы до различных частей тела, указаны в сантиметрах, а вес – в граммах. В источниках есть некоторая неясность относительно единиц измерения высоты и ширины. Однако пока единицы измерения

остаются одинаковыми для всех образцов, данные по-прежнему полезны, даже если мы не знаем, идет ли речь о сантиметрах, каких-то процентах или чем-то еще.

Класс Fish

Для реализации нашего классификатора рыб необходимо создать подкласс `DataPoint`, который будет представлять рыбу:

```
# KNN/fish.py
from dataclasses import dataclass
from KNN.knn import DataPoint
from typing import Self

@dataclass
class Fish(DataPoint):
    kind: str
    weight: float
    length1: float
    length2: float
    length3: float
    height: float
    width: float

    @classmethod
    def from_string_data(cls, data: list[str]) -> Self:
        return cls(kind=data[0], weight=float(data[1]), length1=float(data[2]),
                   length2=float(data[3]), length3=float(data[4]),
                   height=float(data[5]), width=float(data[6]))

    def distance(self, other: Self) -> float:
        return ((self.length1 - other.length1) ** 2 +
                (self.length2 - other.length2) ** 2 +
                (self.length3 - other.length3) ** 2 +
                (self.height - other.height) ** 2 +
                (self.width - other.width) ** 2) ** 0.5
```

С технической точки зрения, нам не нужно явно делать `Fish` подклассом `DataPoint` для соответствия протоколу в подсказках типов Python. Выполняя все требования протокола `DataPoint`, `Fish` может заменить `DataPoint`, даже не создавая его подкласс, благодаря концепции неявных подтипов[1]. Тем не менее мы явно объявляем `Fish` подклассом `DataPoint`, потому что это обеспечивает ясность для читателя нашего кода и помогает в проверке типов.

Каждая строка CSV предоставляется из класса `KNN` в виде списка строк, которые класс `Fish` преобразует в экземпляр `Fish`. Это преобразование выполняется методом `from_string_data()`. Метод `distance()`

[1] «Protocols», typing Documentation, доступ 8 мая 2024 г., *https://typing.readthedocs.io/en/latest/spec/protocol.html#explicitlydeclaringimplementation*.

вычисляет евклидово расстояние между текущим экземпляром и другим Fish. Мы находим различия между каждым измерением в наборе данных, возводим эти различия в квадрат и суммируем их. Затем мы возвращаем квадратный корень (** 0,5) из этой суммы. Обратите внимание, что мы не учитываем атрибут веса из набора данных как часть сравнения. Это связано с тем, что в следующей главе мы будем использовать размеры рыбы для прогнозирования ее веса, поэтому вес рассматриваемой рыбы будет неизвестен.

Модульное тестирование

У нас есть модульные тесты для проверки того, что мы получаем ожидаемые результаты от нашего детектора рыбы. Мы начинаем с теста, который определяет, являются ли рыбы, ближайшие к выборке рыбы, ожидаемыми:

```
# tests/test_knn.py
import unittest
from pathlib import Path
import csv
from KNN.knn import KNN
from KNN.fish import Fish
from KNN.digit import Digit

class FishTestCase(unittest.TestCase):
    def setUp(self) -> None:
        self.data_file = (Path(__file__).resolve().parent.parent
                          / "KNN" / "datasets" / "fish" / "fish.csv")

    def test_nearest(self):
        k: int = 3
        fish_knn = KNN(Fish, self.data_file)
        test_fish: Fish = Fish("", 0.0, 30.0, 32.5, 38.0, 12.0, 5.0)
        nearest_fish: list[Fish] = fish_knn.nearest(k, test_fish)
        self.assertEqual(len(nearest_fish), k)
        expected_fish = [Fish('Bream', 340.0, 29.5, 32.0, 37.3, 13.9129, 5.0728),
                         Fish('Bream', 500.0, 29.1, 31.5, 36.4, 13.7592, 4.368),
                         Fish('Bream', 700.0, 30.4, 33.0, 38.3, 14.8604, 5.2854)]
        self.assertEqual(nearest_fish, expected_fish)
```

Далее мы попробуем классифицировать выборку рыбы:

```
    def test_classify(self):
        k: int = 5
        fish_knn = KNN(Fish, self.data_file)
        test_fish: Fish = Fish("", 0.0, 20.0, 23.5, 24.0, 10.0, 4.0)
        classify_fish: str = fish_knn.classify(k, test_fish)
        self.assertEqual(classify_fish, "Parkki")
--snip--
```

Для запуска этих модульных тестов мы используем наш стандартный код запуска тестов:

```
if __name__ == "__main__":
    unittest.main()
```

Обратите внимание, что мы пропустили около 40 строк в файле *test_knn.py*, содержащих дополнительные тесты, которые появятся позже в этой и следующей главах.

Классификация рукописных цифр

Оптическое распознавание символов (optical character recognition, OCR) – это технология, позволяющая компьютерам распознавать символы в изображениях напечатанного или рукописного текста. Например, на почте есть сортировочные машины, которые используют OCR для автоматического считывания адресов на конвертах. Для выполнения OCR успешно применяется широкий спектр методов, в том числе KNN. В этом разделе мы будем использовать KNN для разработки средства распознавания рукописных цифр, которое достигнет 98%-ной точности на образцах в значительном наборе тестов.

Набор данных, который мы будем использовать, разработан Ценком Кайнаком (Cenk Kaynak) и Этемом Алпайдином (Ethem Alpaydin) в 1998 году в Университете Богазичи города Стамбул, Турция, а затем передан в репозиторий машинного обучения Калифорнийского университета в Ирвине (UC Irvine Machine Learning Repository) по лицензии Creative Commons Attribution 4.0 International. Он состоит из 5620 растровых изображений рукописных цифр (0–9), созданных 43 разными людьми[1]. Изображения цифр уменьшены до размера 8×8 пикселей, при этом каждый пиксель представлен в файле CSV в виде целого числа от 0 до 16, обозначающего его уровень оттенков серого. Каждая строка в CSV состоит из 64 целых чисел, представляющих 64 пикселя в изображении рукописной цифры, плюс 65-е целое число, которое указывает, к какой цифре (0–9) следует отнести изображение. На рис. 7.3 показаны образцы этих цифр.



Рис. 7.3. Некоторые изображения рукописных цифр размером 8×8 из набора данных OCR

Цифры на рисунке несколько искусственно увеличены, но видно, что при уменьшении размера до 8×8 происходит некоторая потеря

[1] Ethem Alpaydin and Cenk Kaynak, «Optical Recognition of Handwritten Digits», UCI Machine Learning Repository, доступ 10 декабря 2024 г., *https://doi.org/10.24432/C50P49*.

деталей. Более низкий уровень детализации и, следовательно, более низкая размерность набора данных ускоряют выполнение нашей программы. Сравнение 64 пикселей изображения, очевидно, происходит гораздо быстрее, чем сравнение 1024 пикселей (до уменьшения размера они были 32×32).

Класс Digit

Для представления каждой цифры мы определяем еще один подкласс `DataPoint`:

KNN/digit.py

```
from dataclasses import dataclass
from KNN.knn import DataPoint
from typing import Self
import numpy as np

@dataclass
class Digit(DataPoint):
    kind: str
    pixels: np.ndarray

    @classmethod
    def from_string_data(cls, data: list[str]) -> Self:
        return cls(kind=data[64],
                   pixels=np.array(data[:64], dtype=np.uint32))

    def distance(self, other: Self) -> float:
        tmp = self.pixels - other.pixels
        return np.sqrt(np.dot(tmp.T, tmp))
```

Мы храним данные пикселей в виде массива NumPy. Это удобно, потому что в следующей главе мы будем использовать Pygame для работы с нашими собственными рукописными цифрами, а Pygame может напрямую взаимодействовать с массивами NumPy. Поскольку метод `distance()` выполняет вычисления по массивам NumPy, мы используем встроенные функции NumPy для реализации вида евклидова расстояния.

Модульное тестирование

Набор данных разделен на 3823 изображения цифр в обучающем наборе и 1797 изображений цифр в тестовом наборе. Мы будем использовать обучающий набор в качестве набора данных, на основе которого наша реализация KNN делает прогнозы, и мы проверим, сколько цифр в тестовом наборе могут быть по нему корректно идентифицированы. Давайте определим для этого еще один тестовый сценарий в *test_knn.py*, после тестового сценария `Fish`, но перед строкой `if __name__ == "__main__"` `line`:

tests/test_knn.py

```
class DigitsTestCase(unittest.TestCase):
    def setUp(self) -> None:
        self.data_file = (Path(__file__).resolve().parent.parent
                          / "KNN" / "datasets" / "digits" / "digits.csv")
        self.test_file = (Path(__file__).resolve().parent.parent
                          / "KNN" / "datasets" / "digits" / "digits_test.csv")

    def test_digits_test_set(self):
        k: int = 1
        digits_knn = KNN(Digit, self.data_file, has_header=False)
        test_data_points: list[Digit] = []
        with open(self.test_file, 'r') as f:
            reader = csv.reader(f)
            for row in reader:
                test_data_points.append(Digit.from_string_data(row))
        correct_classifications = 0
        for test_data_point in test_data_points:
            predicted_digit: str = digits_knn.classify(k, test_data_point)
            if predicted_digit == test_data_point.kind:
                correct_classifications += 1
        correct_percentage = (correct_classifications
                              / len(test_data_points) * 100)
        print(f"Correct Classifications: "
              f"{correct_classifications} of {len(test_data_points)} "
              f"or {correct_percentage}%")
        self.assertGreater(correct_percentage, 97.0)
```

Этот тест загружает обучающий набор данных (*digits.csv*) в экземпляр класса `KNN`. Затем он открывает тестовый набор (*digits_test.csv*) и преобразует данные CSV в список точек данных `test_data_points`. Потом он пытается классифицировать каждую из точек данных по очереди и записывает количество успешных классификаций. Наконец, он сообщает этот процент и выдает ошибку в том случае, если точность ниже 97 %.

Давайте запустим все тесты, чтобы оценить результаты. С учетом тестов с рыбами и OCR это займет некоторое время. Классификация 1797 изображений цифр занимает на моем ноутбуке примерно 11 секунд:

```
% python3 -m tests.test_knn
Correct Classifications: 1761 out of 1797 or 97.9966611018364%
....
----------------------------------------------------------------------
Ran 4 tests in 10.826s

OK
```

Из документации, прилагаемой к набору данных OCR (см. нижнюю часть файла *KNN/datasets/digits/readme.txt*), можно узнать, что авторы сами проверили точность набора данных с помощью KNN при различных значениях *k*. Они обнаружили, что наибольшая точность (98 %) достигается при $k = 1$. Результаты тестирования нашего классификатора совпадают с этими данными. Это означает, что он работает!

Код в реальной жизни

Пару лет назад я преподавал вводный курс по искусственному интеллекту группе студентов старших курсов. Я разделил курс на две части. Первая часть была посвящена тому, что мы назвали в начале этой главы «традиционным ИИ», включая алгоритмы A* и MiniMax (оба можно найти в моей предыдущей книге «*Classic Computer Science Problems in Python*») и такие концепции, как экспертные системы. Вторая половина была посвящена машинному обучению. Я использовал KNN в качестве первого примера алгоритма машинного обучения из-за его чрезвычайной простоты. Он послужил отличным переходом в мир машинного обучения, поэтому я пришел к выводу, что он может сыграть ту же роль для читателей этой книги.

С тех пор мой факультет использует KNN в качестве темы демонстрационной презентации для претендентов на должность преподавателей компьютерных дисциплин, приезжающих на собеседование в университет. Им сообщают тему презентации как минимум за неделю до их прибытия в университет, и у них есть возможность как следует подготовиться. Для этой цели KNN подходит идеально, поскольку, несмотря на множество возможных расширений и улучшений основного алгоритма, само объяснение основного алгоритма не занимает много времени, и даже незнакомые с ним преподаватели или студенты первого курса должны быть в состоянии его понять. Это отличный показатель того, готов ли человек стать хорошим преподавателем.

Кстати, вы удивитесь, сколько докторов наук с опытом в области машинного обучения не способны провести хорошую вводную лекцию по такой теме, как KNN. Стоит помнить, что доктор наук – это научная степень, а не педагогическая. Поэтому когда вы советуете своему ребенку, в какой колледж поступить, вам следует рассмотреть педагогический колледж. В крупном исследовательском университете студентов могут учить преподаватели-исследователи, которые не заботятся о преподавании, внештатные сотрудники, для которых преподавание является подработкой, или, в худшем случае, очень неопытные аспиранты. Наличие преподавателей с докторской степенью не имеет большого зна-

чения для студентов на вводном курсе, если эти преподаватели больше заботятся о грантах на исследования, чем о преподавании. Напротив, в педагогическом колледже у вас есть целый штат преподавателей, работающих на полную ставку (в любом случае, большинство из них имеют докторскую степень), которые получили эту работу именно потому, что полностью посвятили себя искусству преподавания и которым зачастую действительно нравится вести вводные курсы. Вы теряете связь с передовыми исследованиями, но для студента бакалавриата эта связь в любом случае не является тем, что окажет наибольшее влияние на его дальнейшую карьеру. Но относитесь к моим словам с долей скептицизма, поскольку я работаю в педагогическом колледже уже девять лет.

Практические приложения

Алгоритм KNN широко используется в реальной жизни в самых разных областях: от оптического распознавания символов до систем рекомендаций, от классификации текстов до финансового моделирования. Благодаря своей простоте и обширным возможностям применения он повсеместно изучается в рамках технологии машинного обучения.

Однако при практическом использовании KNN необходимо преодолеть несколько проблем, о которых уже упоминалось в этой главе. Первая – это поиск правильного значения *k*. Обычно это делается с помощью перекрестной валидации с использованием тестового набора данных. Какое значение *k* лучше всего подходит для тестовых данных? Использование слишком маленького значения может привести к *переобучению* (overfitting), когда модель очень близка к одному конкретному набору данных. В то же время слишком большое значение может привести к *недообучению* (under-fitting), когда модель очень далека от тестовых данных[1].

Следующая проблема – это влияние базового алгоритма на производительность при работе с большими наборами данных высокой размерности. Как упоминалось ранее в данной главе, существует два способа решения этой проблемы: разработка более эффективной структуры данных для хранения набора данных или использование приближенного поиска. Одной из наиболее популярных структур данных для ускорения поиска ближайших соседей является k-d-дерево (k-d tree)[2]. Однако это довольно сложная структура данных, и ее

[1] Stuart Russell and Peter Norvig, Artificial Intelligence: A Modern Approach, 4th ed. (Pearson, 2021), 688.

[2] Russell and Norvig, Artificial Intelligence.

использование оправдано только в том случае, если производительность имеет критическое значение.

Выбор правильной функции расстояния также имеет решающее значение. Евклидово расстояние подходит для многих приложений, но для булевых измерений более подходит расстояние Хэмминга. Кроме того, в научной литературе хорошо изучены и другие функции расстояния. Правильная функция расстояния зависит от конкретного приложения; универсального решения не существует. Обычно также необходимо нормализовать данные, чтобы исключить возможность влияния на результаты отличающихся единиц измерения или величин. В примере с рыбами в этой главе мы не нормализовали данные, а данные в примере с OCR были все в одинаковых единицах измерения и масштабах, поэтому не требовали нормализации.

Хотя мы увидели, что реализация KNN с нуля – дело практически элементарное, многие популярные библиотеки машинного обучения Python все же имеют встроенные хорошо оптимизированные функции KNN. Например, широкое распространение получила библиотека scikit-learn.

Упражнения

1. Найдите другой набор данных, который вас интересует и который наша реализация KNN может точно классифицировать.
2. Попытайтесь ускорить модульные тесты за счет улучшения производительности метода `distance()` класса `Digit`, сохранив при этом 98%-ную точность теста. Если хотите, можете отказаться от использования массивов NumPy. Вы даже можете отказаться от использования чисто евклидова расстояния. Возможно, вам даже не нужно сравнивать каждый пиксель?
3. Перепишите наш классификатор с использованием библиотеки scikit-learn. Сравните производительность нашего классификатора с классификатором KNN, встроенным в scikit-learn.

8

РЕГРЕССИЯ С ПОМОЩЬЮ К БЛИЖАЙШИХ СОСЕДЕЙ



В этой главе мы расширим нашу реализацию KNN для выполнения задач регрессии. Для наших целей регрессия означает просто прогнозирование числового значения. С помощью небольших дополнений к нашему коду из главы 7 мы можем использовать тот же класс KNN не только для классификации, но и для прогнозирования любого числового значения атрибута в наших наборах данных.

Мы применим регрессию к двум примерам KNN из предыдущей главы. Сначала вернемся к набору данных о рыбах и используем регрессию для прогнозирования веса рыбы на основе ее размеров. Затем напишем программу, которая дает пользователю возможность нарисовать часть цифры, а потом прогнозирует, как может выглядеть остальная часть рисунка.

В отличие от прочих глав данной книги, эта глава не является самостоятельной. Она основана на предыдущей главе. Перед тем как приступить к изучению этой главы, убедитесь, что вы проработали главу 7.

Как работает регрессия KNN

В классификации KNN мы пытались предсказать класс или категорию, к которой принадлежит точка данных, выбирая подходящий класс из ограниченного набора вариантов. В регрессии KNN вместо предсказания класса мы пытаемся предсказать значение атрибута. Эти значения атрибутов обычно являются числовыми, а это означает, что существует потенциально бесконечный диапазон возможных

значений. Конечно, было бы логично, если бы значение атрибута отсутствовало, если это то, что мы хотим предсказать.

Например, предположим, что в больнице пациентов распределяют по палатам. Нам может быть интересно узнать вероятную продолжительность пребывания пациента в больнице. Для прогнозирования мы можем использовать данные о пациентах с похожими диагнозами, симптомами и показателями жизненных функций. Рассмотрим этот пример визуально с помощью диаграммы рассеяния по двум измерениям (рис. 8.1).

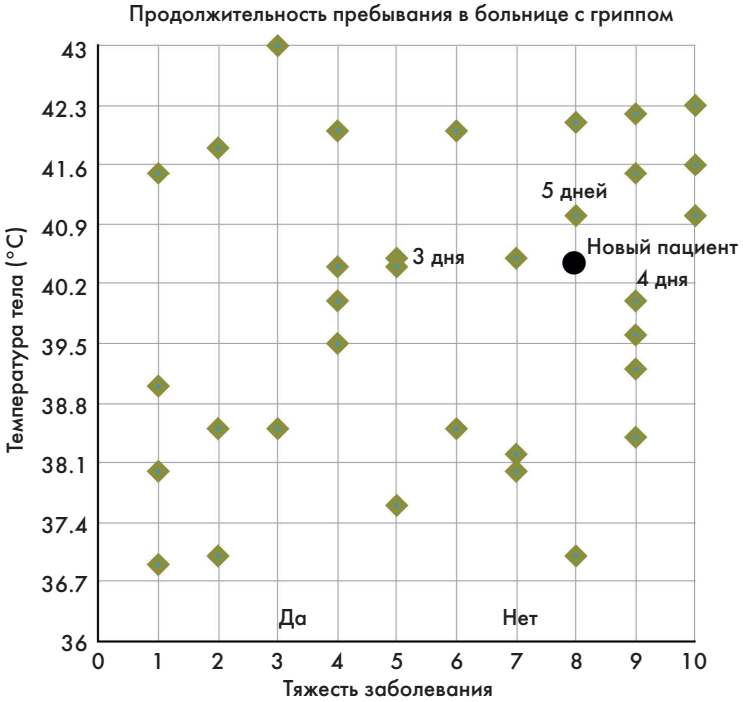


Рис. 8.1. Продолжительность пребывания в больнице при гриппе в зависимости от температуры тела и тяжести заболевания

Предположим, что ромбы на рис. 8.1 представляют прошлых пациентов, которые были госпитализированы с гриппом. Их заболевание было оценено врачом по шкале тяжести от 1 до 10, а их температура была зафиксирована на момент поступления. У нас также есть данные о том, как долго они в конечном итоге находились в больнице.

Круглая точка обозначает только что поступившего пациента. У нас есть оценка тяжести его состояния и температура тела. Необходимо предсказать, как долго он пробудет в больнице, чтобы разместить его в соответствующей палате. Если использовать KNN с евклидовым расстоянием и установить *k* равным 3, то мы рассмотрим трех пациентов из данных прошлых лет, которые на диаграмме рассеяния находятся

ближе всего к точке нового пациента. На рисунке указано, что они пробыли в больнице три, четыре и пять дней соответственно. Оценка продолжительности пребывания нового пациента в больнице с помощью этого метода может быть столь же проста, как вычисление среднего значения трех ближайших соседей. Это среднее значение (среднее или медиана) будет равно четырем, поэтому мы прогнозируем, что новый пациент пробудет в больнице четыре дня.

В более общем плане действия по выполнению регрессии с помощью KNN выглядят следующим образом:

1) выберите *k*, количество соседей для сравнения с точкой данных с отсутствующим атрибутом;
2) найдите *k* ближайших соседей к точке данных;
3) вычислите среднее значение соответствующего атрибута по *k* ближайшим соседям, чтобы предсказать, каким должно быть значение отсутствующего атрибута для данной точки данных.

Как видите, использование KNN для регрессии очень похоже на применение KNN для классификации. Разница заключается только в последнем шаге. У нас есть некоторые вопросы и ответы по поводу этого алгоритма, которые мы уже рассматривали в предыдущей главе: какое значение k является правильным? Как рассчитать расстояние? Ответы на эти вопросы см. в главе 7.

У нас также есть новый вопрос: что означает «взять среднее» (average)? Обычно это либо среднее арифметическое (mean), либо медиана (median). Как и в случае с вопросом о правильной функции расстояния, лучший способ взять среднее может зависеть от конкретного приложения. Обычно для принятия оптимального решения требуются некоторые знания в данной области.

Реализация регрессии с помощью KNN

Для выполнения регрессии нам нужно добавить всего два метода к нашему классу `KNN` из главы 7. Один из них предсказывает скалярный числовой атрибут, а другой прогнозирует атрибут, представляющий собой массив чисел. Мы будем использовать последний для примера с рукописным текстом, чтобы предсказывать пиксели. Вот так выглядят обновления:

```
# KNN/knn.py
# Прогнозирование числового свойства точки данных на основе k ближайших соседей.
# Нахождение среднего значения этого свойства по соседям и его возвращение.
def predict(self, k: int, data_point: DP, property_name: str) -> float:
    neighbors = self.nearest(k, data_point)
    return (sum([getattr(neighbor, property_name) for neighbor in neighbors])
            / len(neighbors))

# Прогнозирование свойства массива NumPy точки данных на основе k ближайших соседей.
```

```
# Нахождение среднего значения этого свойства по соседям и его возвращение.
def predict_array(self, k: int, data_point: DP, property_name: str) -> np.ndarray:
    neighbors = self.nearest(k, data_point)
    return (np.sum([getattr(neighbor, property_name) for neighbor in neighbors], axis=0)
            / len(neighbors))
```

Как и `classify()` из предыдущей главы, эти методы начинают с поиска *k* ближайших соседей. Затем они вычисляют и возвращают среднее значение некоторого свойства среди этих соседей. Здесь мы используем динамическую природу Python, позволяя вызывающему указателю задать свойство в виде строки, а затем использовать `getattr()` для извлечения указанного свойства (или *атрибута*) по имени.

Единственное реальное различие между `predict()` и `predict_array()` заключается в том, что последний использует функцию NumPy `sum()` вместо встроенной функции Python `sum()`. Массивы NumPy также будут работать со встроенной функцией `sum()`, поскольку они реализуют оператор сложения, при этом версия NumPy работает немного быстрее.

Вот и все. По сути, требуется всего две дополнительные строки кода (обе функции примерно одинаковы), и мы уже можем строить прогнозы.

Прогнозирование веса рыб

Давайте добавим модульный тест в `FishTestCase`, чтобы убедиться в работоспособности нашего нового метода `predict()`. Мы хотим ответить на вопрос: «Если мы знаем размеры рыбы, можем ли мы сделать обоснованное предположение о ее весе»? Ответ, конечно же, положительный:

tests/test_knn.py

```
    def test_predict(self):
        k: int = 5
        fish_knn = KNN(Fish, self.data_file)
        test_fish: Fish = Fish("", 0.0, 20.0, 23.5, 24.0, 10.0, 4.0)
        predict_fish: float = fish_knn.predict(k, test_fish, "weight")
        self.assertEqual(predict_fish, 165.0)
```

В этом методе мы создаем `test_fish` без указания веса (`0,0`) и сравниваем его с пятью рыбами, наиболее близкими к нему по анатомическим параметрам, – `length1`, `length2`, `length3`, `width` и `height` (вспомните из предыдущей главы, что в методе `distance()` мы не сравниваем рыб по весу) . Затем мы прогнозируем ее вес посредством усреднения весов этих пяти ближайших соседей. Запустите модульные тесты снова, и вы увидите, что вес рыбы прогнозируется корректно. Чтобы проверить точность этого метода прогнозирования в более общем плане, мы можем попробовать запустить его на всем наборе данных о рыбах.

Поскольку набор данных включает известные веса каждого объекта, мы можем измерить точность результатов KNN по сравнению с фактическим весом рыб.

Прогнозирование недостающей части рукописной цифры

В предыдущей главе мы правильно классифицировали 98 % тестового набора изображений рукописных цифр по сравнению с обучающим набором такого же типа изображений размером 8×8 пикселей с помощью KNN. В этом последнем примере KNN мы классифицируем нарисованную пользователем цифру размером 8×8 и даже прогнозируем, как могут выглядеть остальные пиксели изображения. Вместо того чтобы реализовать это в виде еще одного набора модульных тестов, мы создадим интересную интерактивную программу с помощью Pygame, которая позволит пользователю рисовать в окне сетки 8×8.

Вот предварительный вид того, что мы создаем. На рис. 8.2 показано окно для рисования с неровной цифрой 7, которую я попытался нарисовать. Я нажал клавишу C, и программа правильно классифицировала ее как цифру 7 (это отображается в терминале, на рисунке не показано).

На рис. 8.3 показано окно после нажатия клавиши P, которое запускает прогнозирование внешнего вида остальных пикселей цифры на основе среднего значения пикселей девяти ближайших соседей.

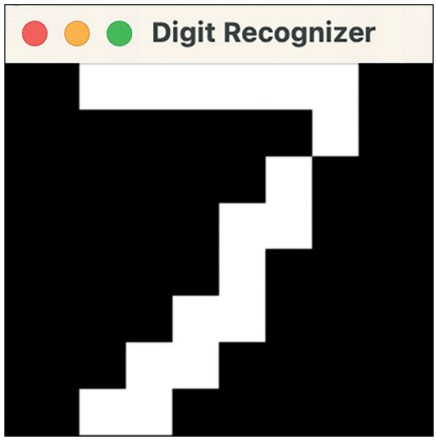


Рис. 8.2. Рисование цифры 7 в программе распознавания цифр

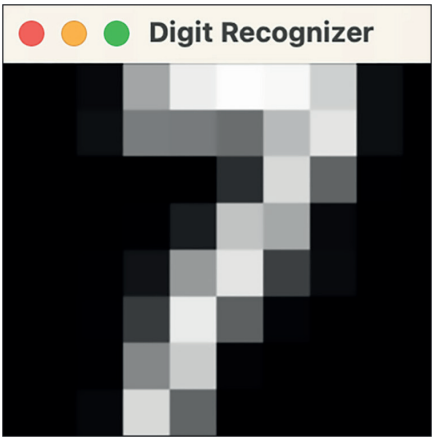


Рис. 8.3. Предсказанные пиксели цифры на основе пикселей ближайших соседей

В нашей простой программе мы можем рисовать только белым цветом. Предсказание дает более красивую цифру 7, поскольку может использовать больше уровней серого. Мы начинаем нашу программу с некоторого импорта и констант:

KNN/__main__.py

```
from KNN.knn import KNN
from KNN.digit import Digit
from pathlib import Path
import sys
import pygame
import numpy as np

PIXEL_WIDTH = 8
PIXEL_HEIGHT = 8
P_TO_D = 16 / 255 # коэффициент масштабирования пикселей в цифры
D_TO_P = 255 / 16 # коэффициент масштабирования цифры в пиксели
K = 9
WHITE = (255, 255, 255)
```

Константы `PIXEL_WIDTH` и `PIXEL_HEIGHT` представляют собой размер одного изображения. Константа `P_TO_D` преобразует 255 оттенков серого в представлении пикселей, с которым мы будем работать, в 16 оттенков серого в исходном наборе данных; `D_TO_P` является ее обратной характеристикой. Мы устанавливаем *K*, количество соседей, которые будет учитывать KNN, равным `9`, а `WHITE` – это константа для белого цвета в формате пикселей RGB.

Это короткая программа. У нас есть только одна центральная функция `run()`, которая тратит столько же времени на обработку пользовательского интерфейса, сколько и на запуск KNN. Вот начало функции:

```
def run():
    # Создание 2D-массива пикселей для представления цифры
    digit_pixels = np.zeros((PIXEL_HEIGHT, PIXEL_WIDTH, 3),
                            dtype=np.uint32)
    # Load the training data
    digits_file = (Path(__file__).resolve().parent
                   / "datasets" / "digits" / "digits.csv")
    digits_knn = KNN(Digit, digits_file, has_header=False)
    # Запуск Pygame, создание окна
    pygame.init()
    screen = pygame.display.set_mode(size=(PIXEL_WIDTH, PIXEL_HEIGHT),
                                     flags=pygame.SCALED | pygame.RESIZABLE)
    pygame.display.set_caption("Digit Recognizer")
```

В этих первых строках мы создаем массив пикселей, загружаем набор данных и инициализируем Pygame. Главное окно инициализируется как «растянутое» 8×8 пикселей. Вы можете изменить размер окна, и оно сохранит свои размеры 8×8. Это устанавливается с помощью `set_mode()` флагов (`flags=pygame.SCALED | pygame .RESIZABLE`).

Далее нам нужно настроить главный цикл:

```
while True:
    pygame.surfarray.blit_array(screen, digit_pixels)
    pygame.display.flip()
```

Поскольку это программа с графическим интерфейсом, использующая Pygame, у нас действительно получился цикл событий. Он прослушивает какие-либо действия пользователя и затем реагирует на них. Этими действиями могут быть события клавиатуры или мыши. Чтобы экран оставался синхронизированным, мы постоянно выводим `digit_pixels` на экран в начале цикла.

Затем мы обрабатываем события клавиатуры:

```
    for event in pygame.event.get():
        if event.type == pygame.KEYDOWN:
            key_name = pygame.key.name(event.key)
            if key_name == "c": # классифицировать цифру
                pixels = digit_pixels.transpose((1, 0, 2))[:, :, 0].flatten() * P_TO_D
                classified_digit = digits_knn.classify(K, Digit("", pixels))
                print(f"Classified as {classified_digit}")
```

Начнем с клавиши C, используемой для классификации. Это похоже на классификацию, которую мы делали в предыдущей главе. Результат выводится на консоль. Единственная сложность заключается в преобразовании пикселей, представляющих изображение, в форму, которую может использовать наш классификатор. По сути, мы переходим от пиксельного формата с 255 оттенками серого и несколькими измерениями к плоскому массиву с 16 оттенками серого. Обработчик клавиатуры продолжает:

```
            elif key_name == "e": # стирание цифры
                digit_pixels.fill(0)
            elif key_name == "p": # прогноз того, как должна выглядеть цифра
                pixels = digit_pixels.transpose((1, 0, 2))[:, :, 0].flatten() * P_TO_D
                predicted_pixels = digits_knn.predict_array(K, Digit("", pixels), "pixels")
                predicted_pixels = predicted_pixels.reshape((
                    PIXEL_HEIGHT, PIXEL_WIDTH)).transpose((1, 0)) * D_TO_P
                digit_pixels = np.stack((predicted_pixels, predicted_pixels,
                                         predicted_pixels), axis=2)
```

Клавиша E просто стирает массив пикселей. Клавиша P отвечает за прогнозирование. Сначала мы вновь преобразуем массив пикселей в форму, которую, как и прежде, может использовать класс `KNN`. Затем мы используем метод `predict_array()` для получения прогнозируемых пикселей `_pixels`, которые являются средним значением

пикселей из девяти ближайших записей в нашем обучающем наборе. Потом преобразуем эти результаты обратно в форму, которую можно отобразить в Pygame. Это включает не только преобразование в двумерный массив, но и изменение формата RGB, где один и тот же уровень серого повторяется в каждом из трех цветовых каналов. Цепочки `reshape()` и `transpose()` переходят от одного к двум измерениям, а вызов `stack()` создает третье измерение, которое имеет одинаковое значение, например уровень серого 128 становится (128, 128, 128) для RGB.

Остальная часть кода рисует белые пиксели в любом месте, где пользователь щелкает мышью, завершает работу, когда пользователь закрывает окно, и вызывает функцию `run()` при выполнении *__main__.py*:

```
            elif ((event.type == pygame.MOUSEBUTTONDOWN) or
                  (event.type == pygame.MOUSEMOTION and pygame.mouse.get_pressed()[0])):
                x, y = event.pos
                if x < PIXEL_WIDTH and y < PIXEL_HEIGHT:
                    digit_pixels[x][y] = WHITE
            elif event.type == pygame.QUIT:
                sys.exit()

if __name__ == "__main__":
    run()
```

Для создания системы распознавания рукописных цифр на основе графического интерфейса пользователя с использованием нашего существующего класса `KNN` требуется всего около 50 строк кода. Python такой лаконичный! Попробуйте сами: он не идеален, но корректно распознает большинство моих каракулей.

Код в реальной жизни

В 2016 году я работал над простым образовательным проектом под названием SwiftSimpleNeuralNetwork[1] в качестве подготовки к главе о построении нейронных сетей с нуля в Swift для моей второй книги «*Classic Computer Science Problems in Swift*». Я разработал распознавание рукописных цифр с помощью этого фреймворка, и оно работало очень медленно. Справедливости ради следует отметить, что оптимизации не было вообще, а приложение было полностью однопоточным и зависимым от процессора. Удивительно, но также неоптимизированная, но гораздо более простая реализация алгоритма KNN в этой главе превосходит его как по скорости, так и по точности.

[1] См. *https://github.com/davecom/SwiftSimpleNeuralNetwork*.

Эта история преподносит два важных урока: более сложный алгоритм не всегда является лучшим алгоритмом для конкретного приложения, и важно провести некоторое исследование, чтобы быть хорошо информированным о том, какие алгоритмы используются для каких приложений. Именно поэтому я включил раздел «Практические приложения» в конце каждой главы этой книги.

Знание алгоритмических возможностей важно во многих областях. В связи с этим одним из преимуществ чтения подобных обзорных книг является то, что они знакомят вас с новыми алгоритмами и методами, о которых вы, возможно, раньше не знали. Таким образом, когда перед вами возникнет проблема, к которой они применимы, вы будете к этому уже подготовлены.

Практические приложения

Алгоритм KNN может быть отличным инструментом для начального этапа изучения регрессии, поскольку он очень прост в использовании. В отличие от нейронных сетей, он практически не требует настройки, а поскольку обучение фактически не требуется, приложение на основе KNN может разработать без особых усилий.

Исследователи действительно использовали KNN для прогнозирования продолжительности пребывания в больнице, как описано в начале главы. Специалисты Pei, Lin и Chen обнаружили, что KNN был примерно столь же точен, как и более сложные методы, такие как логистическая регрессия или алгоритм случайного леса, для прогнозирования продолжительности пребывания пациентов с COVID-19 в больнице[1]. Мне удалось найти несколько других исследований в той же области с применением KNN. Регрессия KNN также использовалась для применения в текстовом майнинге, сельском хозяйстве и на финансовых рынках[2]. Это интуитивно понятно – события или точки данных из прошлого, которые наиболее похожи на то, что происходит в настоящее время, скорее всего, будут наиболее полезны для прогнозирования. Из-за проблем с производительностью KNN не очень

---

[1] Jianing Pei, Xin Lin and Qixuan Chen, «Prediction of Patients' Length of Stay at Hospital During COVID-19 Pandemic» (Прогнозирование продолжительности пребывания пациентов в больнице во время пандемии COVID-19), Journal of Physics: Conference Series 1802 (март 2021 г.): *https://doi.org/10.1088/1742-6596/1802/3/032038*.

[2] Sadegh Bafandeh Imandoust и Mohammad Bolandraftar, «Application of K-Nearest Neighbor (KNN) Approach for Predicting Economic Events: Theoretical Background» (Применение подхода K-ближайших соседей (KNN) для прогнозирования экономических событий: теоретические основы), Journal of Engineering Research and Applications 3, № 5 (2013): 605–610, *https://www.ijera.com/papers/Vol3_issue5/DI35605610.pdf*.

хорошо работает в случаях, когда данные зашумлены или набор данных слишком велик с точки зрения количества точек и размерности. Однако для большинства приложений это разумная отправная точка для применения.

Упражнения

1. Докажите (или опровергните), что KNN эффективен для прогнозирования веса рыбы, запустив его на всем наборе данных о рыбе и сравнив результаты KNN с известными весами из набора данных. Насколько точны прогнозы KNN в среднем?
2. Измените нашу программу распознавания цифр, чтобы использовать более крупную сетку, например 64×64 вместо 8×8. Это даст пользователю возможность рисовать более плавные цифры. Вам нужно будет найти способ точно уменьшить масштаб рисунков 64×64 до 8×8, чтобы использовать их с обучающим набором данных.
3. Используйте либо нашу реализацию регрессии KNN, либо реализацию из библиотеки, такой как scikit-learn, чтобы попробовать сделать прогнозы с применением интересующего вас набора данных.

ПОСЛЕСЛОВИЕ



Спасибо, что прочитали эту книгу. В этом послесловии вы найдете дополнительные ресурсы по четырем темам книги (интерпретаторы, компьютерное искусство, эмуляторы и машинное обучение). Я решил выделить неакадемические, но высоко ценимые ресурсы, которые лично я нашел полезными – здесь нет никакой «башни из слоновой кости». Но прежде чем мы перейдем к этому, я хотел бы поделиться с вами несколькими мыслями о том, чего мы уже достигли.

Что мы сделали и что будет дальше

Работая над проектами, представленными в этой книге, вы получили широкую обзорную информацию о нескольких направлениях компьютерных технологий. Стали ли вы теперь экспертом в этих областях? Конечно, нет. Но вы знаете достаточно, чтобы приступить к реализации собственного проекта в любой из этих четырех областей. Что еще более важно, вы находитесь в хорошем положении, чтобы узнать больше об этих темах.

Интерпретаторы, которые мы создали в части I, были простыми, но NanoBASIC имел все составные части любого реального интерпретатора (токенизатор, парсер, среду выполнения). Теперь вы можете создать интерпретатор для более сложного языка без дополнительного обучения. Сложности могут возникнуть в том случае, если вы захотите повысить производительность этого языка. Для этого могут потребоваться более продвинутые методики, такие как реализация виртуальной машины или компилятора, либо добавление встроенных оптимизаций среды выполнения.

Позже в этом послесловии я поделюсь более подробными ресурсами по интерпретаторам, но суть в том, что вы можете немедленно приступить к работе. Вы когда-нибудь хотели создать свой собственный язык программирования? Теперь у вас есть такая возможность. Я не говорю, что вам обязательно нужно это делать, но вам это по силам.

Программы для создания произведений компьютерного искусства, которые мы разработали в части II, содержат множество интересных алгоритмических техник, но, по сути, не имеют общей идеи, кроме как работа с пикселями. Теперь вы знаете достаточно, чтобы обраба-

тывать пиксели. Если у вас есть идея относительно того, как вы хотите изменить пиксели, вы, вероятно, сможете это сделать. Компьютерная графика в целом наполнена гораздо более широкими идеями и станет вашей следующей остановкой, если вы решите продолжить изучение методов обработки пикселей.

Эмулятор NES в части III был, безусловно, самым большим и сложным проектом в книге. Отличным следующим шагом будет либо добавить больше совместимости к эмулятору (как описано в упражнениях главы 6), либо попробовать сделать эмулятор другой системы. Игровая консоль Game Boy имеет схожую сложность, как и Sega Master System. Как обсуждалось в этой главе, написание эмуляторов на Python является сложной задачей в плане производительности. Если вы не знаете C или C++, написание следующего эмулятора может стать отличной возможностью для их изучения.

В части IV мы сделали первые шаги в мире машинного обучения. Возможно, KNN – это самый простой алгоритм во всех областях практического применения искусственного интеллекта. Он хорошо работает для определенных задач, но вам понадобится изучить несколько других методов, чтобы разобраться в том, какой из них лучше использовать в конкретной ситуации. Надеюсь, что простота части IV помогла вам понять эту область, которая зачастую может казаться сложной. Не нужно быть экспертом в области машинного обучения, чтобы использовать его методы. Сегодня все доступно через вызов библиотеки. Однако вам нужно понимать, какой именно вызов библиотеки необходимо выполнить.

Надеюсь, что выполнение проектов из этой книги помогло вам сделать отличный старт во всех четырех областях. Теперь вам решать, хотите ли вы погрузиться в свои собственные проекты или пройти дополнительное обучение. В остальной части данного послесловия я предложил несколько дополнительных ресурсов по каждой из этих областей, которые, на мой взгляд, хорошо дополняют эту книгу. Все они были проверены лично мной. Я действительно прочитал книги, которые рекомендую вам.

Об изучении компьютерных технологий

Для изучения компьютерных технологий не нужно получать формальное университетское образование. Эта книга – отличное начало. Как и почти во всех других дисциплинах, все необходимое можно найти бесплатно в библиотеке и в интернете. Требуется только настойчивость (время для учебы, время для проектов).

Я отношусь к типу людей, которые предпочитают по возможности разбираться в том, как устроены вещи на фундаментальном уровне. Я встречал других таких людей, и, возможно, это просто черта характера. Даже если вы не относитесь к их числу, позвольте мне попытать-

ся убедить вас в том, что изучение компьютерных технологий дает реальную пользу, позволяя понять, как устроены вещи «под капотом».

Во-первых, для программистов компьютерные знания являются основой для понимания техник, которые могут быть использованы для решения поставленных перед программами задач. Конечно, если мы просто создаем общие приложения по принципу CRUD (create, read, update, delete – создание, чтение, обновление, удаление четыре базовые функции работы с данными. – *Прим. перев.*), это может оказаться не очень полезным, но если вы хотите сделать что-то новое или сложное, компьютерные знания действительно придутся кстати.

Во-вторых, даже если мы можем придумать способ решения проблемы, будет ли он наиболее эффективным? Не создает ли он проблемы с производительностью? Понимание некоторых основ компьютерной грамотности может действительно помочь вам улучшить производительность вашего кода.

И наконец, понимание основ компьютерных технологий поможет вам в вашей карьере. Вы будете понимать, о чем говорят ваши коллеги. Вы «поймете» программные технологии на гораздо более фундаментальном уровне. Вы станете лучшим техническим экспертом. И это поможет вам на технических собеседованиях. К сожалению, слишком многие компании по-прежнему требуют от кандидатов решения задач по структуре данных и алгоритмам на доске. Я не согласен с этой практикой, но нет сомнений, что те, кто изучал компьютерные дисциплины, получают преимущество на таких собеседованиях.

Компьютерные дисциплины – обширная тема. Не стоит этого пугаться. Вот несколько доступных книг, которые, на мой взгляд, хорошо дополняют эту книгу, если вы заинтересованы в дальнейшем изучении компьютерных дисциплин в целом.

Grokking Algorithms, 2nd Edition, by Aditya Y. Bhargava

Это чрезвычайно увлекательная книга об алгоритмах. Она воспринимается намного легче, чем обычные учебники по алгоритмам. В ней меньше математики и есть примеры на Python, иллюстрирующие каждую концепцию. Я сам использую ее вместо традиционного учебника, когда преподаю курс по структурам данных и алгоритмам в колледже.

Classic Computer Science Problems in Python by David Kopec

Я написал книгу «*Classic Computer Science Problems*» (Классические проблемы информатики на Python) в качестве общего обзора интересных алгоритмических тем, которые преподаются в формате «код прежде всего» в стиле учебного пособия. Она охватывает все, от алгоритмов поиска до алгоритмов графов и даже некоторых вводных материалов по искусственному интеллекту. Она идеально дополняет эту книгу, и между ними нет никакого дублирования содержания, поскольку они охватывают разные области компьютерных технологий.

В то время как эта книга состоит из более крупных, увлекательных проектов, книга «Классические проблемы компьютерных технологий» больше посвящена самим алгоритмам и конкретным проблемам, для решения которых они подходят.

Интерпретаторы

Еще десять лет назад было очень мало книг неакадемического характера, которые можно было бы порекомендовать широкой аудитории программистов по теме написания интерпретаторов. Сегодня мы имеем возможность выбрать из нескольких отличных изданий, в числе которых два, которые я особенно рекомендую:

Crafting Interpreters by Robert Nystrom
Это абсолютно замечательная книга как с точки зрения методики изложения, так и с точки зрения содержания кода. Старые классические книги по интерпретаторам и компиляторам чрезвычайно академичны и даже несколько скучны, как, например, так называемая «*Книга дракона*» (Dragon Book). *Crafting Interpreters* – обязательная книга в этой области, если вы хотите научиться создавать полноценные практические интерпретаторы после того, как вас заинтересовали представленные в этой книге проекты Brainfuck и NanoBASIC.

Writing an Interpreter in Go by Thorsten Ball
Эту книгу я прочитал в процессе подготовки к написанию интерпретируемого языка программирования SeaTurtle, предназначенного для обучения детей написанию кода (см. вставку «Код в реальной жизни» в главе 2). Она менее обширна, чем *Crafting Interpreters*, и немного более нишевая, поскольку написана на Go. Тем не менее если вас интересует что-то более лаконичное или вы являетесь программистом Go, эта книга станет отличным выбором. Она хорошо написана, а сопровождающий ее код великолепен.

Компьютерное изобразительное искусство

О методах, описанных в двух главах книги на тему компьютерного изобразительного искусства, я по большей части узнал из коротких онлайн-статей, поэтому, к сожалению, не могу поделиться какими-либо исчерпывающими ресурсами, но хочу упомянуть проект Майкла Фоглемана (Michael Fogleman) под названием Primitive. Он послужил источником вдохновения для проекта «Импрессионист» (см. главу 4), хотя в этой программе используется другой алгоритм. В репозитории GitHub для Primitive (*https://github.com/fogleman/primitive*) есть несколько простых для чтения исходных кодов на языке Go, а у Майкла также есть веб-сайт, посвященный этому проекту (*https://www.michaelfogleman.com/#primitive*).

Эмуляторы

Возможно, существуют хорошие тексты по написанию эмуляторов, но лично я о них не знаю. Вместо этого я поделюсь парой онлайн-ресурсов, которые я нахожу для себя весьма полезными:

EmuDev
Этот подраздел Reddit (*https://www.reddit.com/r/EmuDev*) – одно из самых активных сообществ по разработке эмуляторов среди всех, с которыми я сталкивался. Здесь можно найти людей, которые создают всевозможные эмуляторы и делятся своими знаниями. Есть также сопутствующий Discord, который может быть полезен для получения ответов на вопросы в режиме реального времени.

NesDev
Вики-документация и форумы на *https://www.nesdev.org* оказали мне неоценимую помощь как при разработке моего первого эмулятора NES, так и при написании эмулятора для этой книги. К сожалению, несмотря на то что эмуляция NES является популярным проектом, существует очень мало хороших учебных пособий или ресурсов по этой теме (что и стало одной из причин, по которой я решил написать эту книгу). NesDev – лучший из существующих ресурсов, и если вы хотите продолжить разработку NES за пределами того, что мы сделали в главе 6 (например, добавить больше мапперов или более точный PPU), этот сайт станет для вас незаменимым источником информации.

Машинное обучение

Существует так много ресурсов по машинному обучению, что трудно даже решить, с какого из них начать. Однако если вам понравился простой алгоритм, представленный в главах 7 и 8, и вам понравился способ, которым мы разработали его с нуля, то у меня есть два особенно понятных ресурса для изучения других алгоритмов машинного обучения, которые вы можете применить:

The Hundred-Page Machine Learning Book by Andriy Burkov
Эта книга рассказывает только о самом главном. Вы изучите алгоритм с помощью достаточного количества теории и другой информации, необходимой для его реализации, без каких-либо излишних слов, характерных для некоторых технических книг. Хотя в ней не так много кода, она отлично объясняет материал, который можно дополнить с помощью хороших онлайн-курсов или каналов YouTube.

Classic Computer Science Problems in Python by David Kopec
Да, я во второй раз рекомендую свою книгу. Это немного эгоистично, но я бы не стал ее писать, если бы не считал ее действительно отличным ресурсом. По сути, пять из девяти глав книги посвящены искус-

ственному интеллекту, а две главы – конкретно машинному обучению. Хотите научиться писать нейронные сети с нуля на Python (без библиотек)? Ознакомьтесь с главой 7 этой книги. В ней вы узнаете, как написать простой классификатор и регрессор с использованием KNN. В главе 6 вы также узнаете, как создать программу кластеризации с помощью другого простого алгоритма, k-means.

Ресурсы автора в интернете:

X: *https://x.com/davekopec*

GitHub: *https://github.com/davecom*

LinkedIn: *https://www.linkedin.com/in/dkopec*

YouTube: *https://www.youtube.com/c/DavidKopec09*

Веб-сайт: *https://davekopec.com*

Подкаст *Kopec Explains Software*: *http://kopec.live*

ПРИЛОЖЕНИЕ
ПОБИТОВЫЕ ОПЕРАЦИИ



Работа с битами на низком уровне имеет огромное значение для половины проектов, описанных в этой книге. Если вы не имеете опыта работы с битовыми операциями, в этом приложении представлен обзор, включая описание наиболее важных битовых операций, способы их использования в Python и несколько примеров их применения.

Обзор двоичной системы счисления

Я предполагаю, что большинство читателей, как программисты среднего или продвинутого уровня, знакомы с двоичной системой счисления. Если это ваш случай и вы просто хотите быстро освежить в памяти знания о побитовых операциях, можете пропустить этот раздел. Однако если вы незнакомы с двоичной системой счисления, этот раздел поможет вам лучше понять ее, хотя и не даст исчерпывающих знаний.

Вся информация в компьютерах хранится в виде единиц и нулей. Это удобно, потому что тип оборудования, используемого для создания компьютеров, позволяет представлять единицы и нули на физическом уровне. Например, если присутствует электричество (или «сигнал»), мы можем сказать, что оно представляет единицу, а отсутствие электрического сигнала представляет ноль. Двоичная система также проявляется на физическом уровне в ныне устаревшей технологии CD и DVD. Их считывающие устройства имеют лазер, который проходит по поверхности диска. Когда лазер не отражается обратно из-за микроскопической ямки на диске, это означает 0. Если лазер отражается обратно, потому что ямки нет, это означает 1. Последний физический пример – QR-коды. Наличие черной точки может означать 1, а отсутствие – 0. Существует много других понятных физических проявлений двоичной системы.

Как все эти единицы и нули преобразуются в информацию? Последовательность единиц и нулей представляет собой число в двоичной системе счисления. А когда у нас есть числа, мы можем представлять любую другую информацию. Конкретное число может представлять конкретную букву в электронном документе. Или оно может пред-

ставлять конкретный цвет. Другое число может представлять место на экране, где следует разместить этот цвет. Вскоре у нас появляются пиксели.

Но как последовательность единиц и нулей представляет число, отличное от 1 или 0? Здесь в игру вступает двоичная система счисления, также называемая системой счисления с *основанием 2* (base 2).

Типичные числа, которые мы используем в повседневной жизни, имеют основание 10, то есть *десятичную* (decimal) систему счисления. Это означает, что каждая цифра в числе может иметь 10 различных значений (0–9), и каждая цифра сама по себе представляет степень 10. Например, число 427 на самом деле равно $(4 \times 10^2) + (2 \times 10^1) + (7 \times 10^0)$. Точно так же каждая цифра в двоичном числе может иметь два разных значения (0 или 1), и каждая цифра сама по себе представляет степень числа 2. Число 427 в двоичной системе записывается как 110101011, что соответствует $(1 \times 2^8) + (1 \times 2^7) + (0 \times 2^6) + (1 \times 2^5) + (0 \times 2^4) + (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)$. Цифры 1 – это степени числа 2, которые «включены», а цифры 0 – это степени числа 2, которые «выключены».

Чтобы проверить свое понимание, попробуйте преобразовать несколько чисел из десятичной системы в двоичную и наоборот. Сколько будет 73 в двоичной системе? Сколько будет 11000 в десятичной системе? Попробуйте еще несколько чисел, которые вы выберете сами. Вы можете проверить свои результаты с помощью Python, где двоичные числа представлены в виде литералов с префиксом `0b`, например `0b11` для десятичного числа 3:

```
>>> value = 0b11
>>> value
3
```

Между тем функция `bin()` в Python принимает целое число и возвращает строку, отформатированную как двоичный эквивалент:

```
>>> bin(3)
'0b11'
```

Каждое сохраненное двоичное значение 1 или 0 в вычислительной технике называется *битом* (bit). Стандартный *байт* (byte) состоит из 8 бит. Все современные компьютеры используют 8-битный байт в качестве стандартной единицы хранения. Максимальное значение, которое может содержать байт (при представлении целого числа без знака), равно 255, поскольку восемь единиц, или 11111111 в двоичной системе, равно 255 в десятичной системе. Это также означает, что байт может содержать 256 различных возможных значений (все значения от 0 до 255).

При записи байта *младший бит* (least-significant bit), то есть бит, представляющий наименьшую степень 2, обычно записывается в са-

мом конце справа (бит 0, представляющий 2^0). *Старший бит* (most-significant bit), представляющий наибольшую степень 2, обычно записывается в самом конце слева (бит 7, представляющий 2^7). Например, в 10000000 цифра 1 представляет 2^7, поэтому байт имеет десятичное значение 128. Это предполагает, что мы работаем с целыми числами *без знака* (unsigned) – целыми числами, которые не могут быть отрицательными. Знаки выходят за рамки этого базового введения в двоичную систему счисления.

Некоторые типы данных представлены с использованием более одного байта. Например, на 64-разрядном микропроцессоре, подобном тому, который, вероятно, используется в вашем компьютере, целые числа обычно хранятся с использованием 64 бит (8 байт). Так выглядит максимальное значение 64-разрядного числа в двоичной системе:

```
1111111111111111111111111111111111111111111111111111111111111111
```

Это соответствует 18 446 744 073 709 551 615 в десятичной системе счисления.

Для нескольких проектов в данной книге нам необходимо работать с байтами на битовом уровне. В остальной части данного приложения рассматриваются некоторые распространенные операции для выполнения этих операций.

Распространенные побитовые операции

Побитовые операции (bitwise operations) позволяют работать со значениями на уровне отдельных двоичных цифр. Это означает работу с единицами и нулями. Все микропроцессоры содержат инструкции для выполнения побитовых операций, а Python имеет операторы для использования этих инструкций микропроцессора. *Таблицы истинности* (truth tables), которые показывают истинный/ложный результат логических функций на основе различных комбинаций входов, могут пригодиться при изучении побитовых операций. Чтобы сделать это более практичным и применимым к использованию таких операций в Python, в сопровождающих каждую операцию таблицах вместо истинного и ложного будут показаны двоичные значения.

Сдвиг влево (<<)

Вместо того чтобы рассматривать наши двоичные данные как числа, на минуту представьте их как набор единиц и нулей. Представьте, что нули – это пустые пробелы, а единицы – заполненные пробелы. Что, если мы хотим сдвинуть все единицы на одно место влево? Это задача *сдвига влево* (left shift), который в Python обозначается оператором <<.

Сдвиг влево оставляет пробел в младшем значащем бите (битовая позиция 0). При сдвиге влево в Python мы заполняем этот пробел нулем. Поскольку целые числа в Python имеют произвольную длину (нет ограничения по максимальной длине), мы не можем сдвинуть единицы с конца, переместив их влево. Число просто увеличивается на одну цифру. Например, `1010`, сдвинутое влево на 1, становится `10100`, а не `0100`. Мы также можем сдвигать влево более чем на одну позицию, поэтому `1010`, сдвинутое влево на три, становится `1010000`. В табл. А.1 представлены результаты некоторых сдвигов влево.

A	A << 1	A << 3
0	0	0
1	10	1000
1010	10100	1010000

Таблица А.1 Примеры сдвига влево

В первой строке таблицы, где происходит сдвиг `0`, можно подумать, что он станет `00` и `0000`, но на самом деле это то же самое, что `0`. Вместо этого сдвиг влево можно рассматривать как простое перемещение единиц. Если единиц нет, то сдвиг фактически ничего не делает.

Оператор сдвига влево в Python предваряется сдвигаемым элементом и следует за целым числом, указывающим количество сдвигаемых позиций. Вот краткий пример его использования:

```
>>> bin(0b1010 << 3)
'0b1010000'
```

Операторы сдвига обычно используются для выравнивания бита или битов с другим двоичным значением в сочетании с другими побитовыми операторами, о которых будет рассказано далее.

Сдвиг вправо (>>)

Сдвиг вправо (right shift) очень похож на сдвиг влево, за исключением того, что 1 перемещаются вправо, а не влево. Если 1 выходит за пределы (за позицию 0-бита), то он «теряется». Никакого переноса нет. Например, `1001`, сдвинутый вправо на одну позицию, становится `100`, а не `1100`. Мы также можем сдвигать вправо более чем на одну позицию, поэтому `1001`, сдвинутый вправо на три позиции, становится 1.

В Python для выполнения сдвига вправо используется оператор >>. В табл. А.2 приведены некоторые примеры сдвига вправо.

A	A >> 1	A >> 3
0	0	0
1	0	0
1010	101	1

Таблица А.2 Примеры сдвига вправо

Во второй строке таблицы 1 сдвигается «за пределы» и 1 больше не остается, поэтому результат равен 0.

ИЛИ (|)

При *побитовой* операции *ИЛИ* (OR), выполняемой над двумя значениями, если одно из значений равно 1, то результат будет равен 1. Если ни одно из значений не равно 1, то результат будет равен 0. Операция выполняется над двоичными цифрами, которые находятся в одних и тех же разрядах между двумя значениями, по одному разряду за раз. Представьте, что числа выстроены в ряд, одно под другим. Затем операция OR выполняется в каждой колонке, по одной колонке за раз, как показано ниже:

```
1010
0110
----
1110
```

Попробуйте самостоятельно вычислить последнюю строку внизу, следуя правилам из предыдущего абзаца. Эти правила также обобщены в табл. А.3.

A	B	A|B
0	0	0
0	1	1
1	0	1
1	1	1
1010	0110	1110

Таблица А.3 Примеры ИЛИ

В Python есть оператор | для выполнения побитового ИЛИ. Вот простой пример его использования с двумя двоичными числами в Python:

```
>>> bin(0b1010 | 0b0110)
'0b1110'
```

Одним из распространенных применений операции побитового ИЛИ в низкоуровневом программировании является объединение двух значений в сочетании с операторами сдвига. Например, предположим, что у нас есть один ниббл (*ниббл* – это 4 бита), который представляет половину байта, и другой ниббл, который представляет вторую половину байта. Мы хотим объединить их, чтобы получить полный байт. Предположим, что ниббл A равен `1001`, а ниббл B равен `0110`, и мы хотим, чтобы ниббл A был первой половиной, а ниббл B – второй половиной результирующего байта. Код для их объединения может выглядеть следующим образом:

```
>>> a = 0b1001
>>> b = 0b0110
>>> c = (a << 4) | b
>>> bin(c)
'0b10010110'
```

С помощью `(a << 4) | b` мы сдвигаем `a` влево на 4 позиции, а затем выполняем операцию ИЛИ с его значениями и `b`. Помните, что при сдвиге влево справа добавляются нули, поэтому `a`, сдвинутый влево на 4 позиции, становится `10010000`. Затем, если мы выравниваем `a` и `b` и выполняем операцию ИЛИ, мы получаем:

```
10010000
    0110
--------
10010110
```

Полученный байт, `10010110`, имеет исходное значение `a` в левых четырех разрядах и исходное значение `b` в правых четырех разрядах.

Если вы видите это впервые, это может показаться немного абстрактным. Например, вы можете задаться вопросом, почему числа хранятся только в 4 битах. Чтобы привести только одну из нескольких причин, в проектах этой книги мы видим несколько сценариев, в которых хотим сэкономить место, объединив значения, которым требуется менее 8 бит, в одном байте. Например, 8-битные микропроцессоры нередко имели 1-байтовый регистр флагов, где каждый отдельный бит представлял отдельный флаг, который мог быть включен или выключен. Это гораздо экономичнее, чем использовать отдельный байт для каждого флага. Точно так же некоторые форматы файлов хранят значения, которым требуется менее 1 байта, в одном и том же байте для экономии дискового пространства.

И (&)

Побитовое И (AND) возвращает 1, если оба операнда равны 1; в противном случае оно возвращает 0. Вот пример с использованием тех же операндов, которые мы рассматривали с побитовым OR:

```
1010
0110
----
0010
```

Только бит, который оказался в одной линии с двумя единицами, дал результат 1. В табл. А.4 приведены примеры работы побитового оператора И.

A	B	A&B
0	0	0
0	1	0
1	0	0
1	1	1
1010	0110	0010

Таблица A.4 Примеры использования оператора И

В Python есть оператор & для выполнения побитового И. Вот простой пример его использования с двумя двоичными числами:

```
>>> bin(0b1010 & 0b0110)
'0b10'
```

Обратите внимание, что в выводе отсекаются ведущие нули, поскольку они не нужны для представления результирующего числа (2 в десятичной системе). Другими словами, 0010 в двоичной системе соответствует 2 в десятичной системе, так же как 10 в двоичной системе соответствует 2 в десятичной системе.

Одним из распространенных применений операции побитового И в низкоуровневом программировании является обеспечение того, чтобы конечный результат включал только некоторые биты из предыдущего результата. Например, предположим, что нас интересуют только 4 правых бита в байте 10011110. Мы можем выполнить операцию И с байтом 1111, чтобы гарантировать, что в конечном результате будут только 4 правых бита. Код может выглядеть следующим образом:

```
>>> a = 0b10011110
>>> b = 0b1111
>>> c = a & b
>>> bin(c)
'0b1110'
```

Давайте посмотрим на эту операцию с выровненными битами:

```
10011110
    1111
--------
    1110
```

Мы нередко используем эту технику с одним битом в наших проектах эмуляторов при работе с флагами. Нам просто нужно определить один флаг и узнать, равен ли он 1 или 0 (истина или ложь).

XOR (^)

Побитовая операция *исключающего ИЛИ* (XOR) возвращает 1, если операнды различны (один 1 и один 0); в противном случае возвраща-

ется 0. Вот пример с использованием тех же операндов, что мы рассматривали с побитовыми операциями ИЛИ и И:

```
1010
0110
----
1100
```

В табл. А.5 приведены результаты работы операции XOR.

A	B	A^B
0	0	0
0	1	1
1	0	1
1	1	0
1010	0110	1100

Таблица А.5 Примеры XOR

В Python есть оператор ^ для выполнения побитового XOR. Вот простой пример применения этого оператора к двум двоичным числам в Python:

```
>>> bin(0b1010 ^ 0b0110)
'0b1100'
```

XOR – это удивительно мощная операция. Она лежит в основе неразрывной схемы шифрования, известной как *блокнот одноразового назначения* (one-time pad). Вы также можете перевернуть биты с помощью XOR с 1: если вы выполните XOR 1 с 1, он станет 0, но если вы выполните XOR 0 с 1, он станет 1. Любой бит, который вы выполните XOR с 1, станет противоположным тому, что был раньше. Так работает рисование на экране в проекте CHIP-8 в главе 5.

Дополнение (~)

Дополнение (complement) – это самая простая из всех битовых операций (также известна как «*побитовое НЕ*»): она заменяет все 1 на 0, а все 0 на 1, как показано в табл. А.6.

A	~A
1	0
0	1
1010	0101
011010	100101

Таблица А.6 Примеры дополнения

В Python есть оператор ~ для вычисления двоичного дополнения. В этой книге дополнение используется нечасто, за исключением одного небольшого момента в эмуляторе NES в главе 6.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Символы
^ (оператор XOR), 283
<< (оператор сдвига влево), 278
>> (оператор сдвига вправо), 279
| (оператор ИЛИ), 280
~ (оператор дополнения), 283
& (оператор AND), 282

A
ADC, инструкция, 204
ArgumentParser, 29, 31, 89, 120, 154

B
BASIC, 40, 41
  диалект, 41
boolean_expr, 75
Brainfuck
  запуск интерпретатора, 33
  интерпретатор, 27
  команды, 23
  определение, 20
  принцип работы, 22
  реализация на Python, 28
  создание интерпретатора, 29
  тестирование интерпретатора, 34
BrickBreaker, игра, 214, 237

C
C в сравнении с Python, 23
Chase, игра, 237
CHIP-8
  виртуальная машина, 144
  инструкция, 146
  платформа, 142
  реализация, 150
  регистр и память, 145
CHR RAM, 213
CHR ROM, 185, 213, 214
Clojure, 36
Common Language Runtime (CLR), 170
consume(), 64
COSMAC VIP, 144
Counter (Python), 250
CSV, 252

D
DataPoint, 247, 252, 255
DEFLATE, алгоритм, 113

F
FPS, 154, 221

G
getattr(), 263
getcolors(), 126
getpixel(), 93
GOSUB, 42
GOTO, 42

H
hblank (NES), 179, 222

I
IBM PC, 41
IfStatement, 58
iNES, 181, 182
InterpreterError, 62

J
Java, 37, 143, 161
JIT, 83, 170
Joypad, 188
JVM, 143

K
k-means, 275

L
Lan Master, игра, 238
LetStatement, 61

Lisp, 36
LLM, 244
Logo, 81

M
MacBinary, 96, 105
Macintosh, 85, 95, 98, 112, 240
Mac OS, 96, 105
MacPaint, 95
MemMode, 187, 205
Microsoft, 240
MOS, 174

N
NanoBASIC
  запуск, 77
  комментарии, 43
  ошибка, 62
  парадигма, синтаксис
  и семантика, 41
  пример программы, 47
  реализация, 53
  синтаксис, 43
  стиль и особенность, 46
  тестирование, 78
  формализация синтаксиса, 50
nearest(), 250
NES, тестирование эмулятора, 233
NesDev, 173, 182, 240
NROM, 182
NumPy, 151, 227, 255

P
P-код, 170
PackBits, алгоритм, 99
Panic Playdate, 86
ParserError, 62
Pillow, 88, 89, 90, 93, 125
play_sound, 157
PrintStatement, 67
putpixel(), 93
Pygame, 151, 157, 177, 255, 264

Q
QR-код, 276

R
Retro Dither, 89
ReturnStatement, 75
Ricoh, 174

S
SCHIP, 170
SeaTurtle, 81
setZN(), 211
step(), 162, 190, 204
struct, 183
SVG, 121
switch, 30, 161
System 1, 110

T
take_same(), 101
Thwaite, игра, 218
Tiny BASIC, 42

U
unpack(), 183

V
vblank (NES), 180

W
write_memory(), 210

X
XML, 122

А
Абстрактная модель, 21
Абстрактное искусство, 137
Абстракция, 42, 144, 176
Адресация памяти (6502), 207
Алгоритм
  Аткинсона, 90
  дизеринга, 85, 90
  поиска восхождением
  к вершине, 131, 132
  рекурсивного спуска, 63
  сортировочной станции, 72
  стохастический, 114
  стохастического распределения
  ошибки, 90
  Флойда–Стейнберга, 90
  k-ближайших соседей (KNN), 243
  KNN, 243, 245, 258, 268
Аллен Пол, 240
Анимированный GIF, 85, 112
Аткинсон Билл, 90, 113
Аудиопроцессор (APU), 174, 208, 241

Б
Битовая плоскость, 215
Брофельдт Пекка, 250

В
Вайсбекер Джозеф, 143
Векторная графика, 118
Виртуальная машина, 142

Г
Гейтс Билл, 240
Генератор парсера, 63
Глубокое обучение, 244
Гомоиконичность, 36
Горизонтальное зеркалирование, 217

Д
Двоично-десятичные числа (BCD), 150
Дейкстра Эдсгер, 46
Декларативный язык, 42
Дерево
  абстрактного синтаксического анализа, 27, 53
  k-d, 258
Дизеринг, 85
Динамическая перекомпиляция, 161

Е
Евклидово расстояние, 247

И
Императивный язык, 42
Импрессионист, 120
Интеграционный тест, 34
Интерпретатор, 37
Искусственный интеллект (ИИ), 243

К
Картридж (NES), 151, 174, 181, 208
Кемени Джон, 40, 80
Классификация, 243
  рыб, 250
Кодирование по длине серии, 95
Компилятор, 37
Компьютерное изобразительное искусство, 273
Контекстно-свободная грамматика, 50
Копек Дэнни, 80
Курц Томас, 40

Л
Локальный максимум, 132

М
Метаэвристика, 245
Мюллер Урбан, 20

Н
Набор шрифтов, 159
Наиболее распространенный цвет, 116
Недообучение, 258
Нейронная сеть, 244
Нейронный процессор, 244
Немаскируемое прерывание, 180
Ниббл, 156, 280
Нисходящий парсинг, 64

О
Объединенный тип, 56
Объектно-ориентированное программирование, 42
Ограничивающая рамка, 129
Оператор моржа, 102
Оптическое распознавание символов (OCR), 254
Отображаемый в памяти аппаратный регистр, 175
Оттенок серого, 86
Ошибка
  интерпретатора, 62
  парсера, 62

П
Память палитры, 217
Парсер (синтаксический анализатор), 27, 63
Переключение банков, 175, 181
Перекрестная валидация, 258
Переобучение, 258
Побитовая операция, 278
Полутон, 86
Порядок
  big-endian, 106
  little-endian, 107
Последовательность Фибоначчи, 47
Проба, 129

Прогнозирование веса рыб, 263
Прогнозирование недостающей части рукописной цифры, 264
Продукционное правило, 48
Процедурный язык, 42
Процессор обработки изображений (PPU), 174, 212
Прямой доступ к памяти (DMA), 190

Р

Развилка
  данных, 105
  ресурсов, 105
Разрешение экрана, 88
Распределитель (маппер), 181
Расстояние Хэмминга, 247
Регистр флага, 160
Регрессия, 243
Рекурсия, 64

С

Сникернет, 111
Спрайт, 158, 212
Среда выполнения, 27, 72
Стек, 32, 75
Столкновение спрайта-ноль, 229
Стохастическая оптимизация, 138
Страница памяти, 188
Структурированное программирование, 46

Т

Таблица
  атрибутов, 218
  имен, 216
  переходов, 161
  шаблонов, 214
Телевизор с кинескопом (CRT), 179
Токенизатор, 27, 54
Торвальдс Линус, 41
Точка данных, 245
Транспайлер (транспилятор), 38

Ф

Файзуллин Марат, 182
Фон, 212
Форма Бэкуса–Наура, 48
Фрейм, 152

Ш

Шестнадцатеричная система счисления, 147

Э

Эллисон Деннис, 50
Эмулятор, 274
  определение, 142
Эмуляция центрального процессора, 185

Я

Язык, полный по Тьюрингу, 21
Янг Джеффри, 113

Дэвид Копек

Python для профи

Главный редактор *Мовчан Д. А.*
Зам. главного редактора *Яценков В. С.*
editor@dmkpress.com
Перевод *Бахур В. И.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Гарнитура PT Serif. Печать цифровая.
Усл. печ. л. 23,4. Тираж 200 экз.



Python для профи

Если вы имеете опыт программирования, то наверняка задавались вопросом о базовых принципах, лежащих в основе кода. Книга восполняет пробелы в знаниях читателей с помощью тщательно разработанных примеров, которые оживляют основные концепции информатики. Глубокое понимание ключевых подходов сделает вас более разносторонним и эффективным программистом.

Каждая глава описывает целенаправленный практический проект, обучающий фундаментальной идее в области информатики, и завершается примерами из реальной жизни, продуманными дополнениями и упражнениями для закрепления знаний.

Основные темы книги:

- *интерпретаторы – понимание синтаксического анализа и оценки кода путем написания интерпретатора языка BASIC;*
- *эмуляторы – изучение архитектуры компьютера путем создания эмулятора NES с нуля;*
- *графика – изучение обработки изображений и алгоритмического искусства с помощью проектов компьютерной графики;*
- *машинное обучение – понимание принципа классификации на примере простой и понятной модели KNN.*

Дэвид Копек – доцент кафедры информатики в колледже Олбрайт, Пенсильвания. Автор четырех технических книг, включая «Классические задачи Computer Science на языке Python», которая была переведена на восемь языков и опубликована по всему миру. Копек – активный разработчик приложений и подкастер.

ISBN 978-6-01140-652-9

no starch press

BOOKS.KZ