

**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 Копек  
© Перевод, оформление, издание,  
Books.kz, 2026

# СОДЕРЖАНИЕ

<i>От издательства</i> .....	10
<i>Благодарности</i> .....	12
<i>Введение</i> .....	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
<b>Часть II ИСКУССТВО И ВЫЧИСЛЕНИЯ.....</b>	<b>84</b>
<b>Глава 3. Ретрообработка изображений .....</b>	<b>85</b>
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
<b>Глава 4. Стохастический алгоритм живописи .....</b>	<b>114</b>
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
<b>Часть III ЭМУЛЯТОРЫ .....</b>	<b>141</b>
<b>Глава 5. Создание виртуальной машины CHIP-8 .....</b>	<b>142</b>
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
<b>Глава 6. Эмуляция игровой консоли NES.....</b>	<b>172</b>
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
<b>Часть IV  ОЧЕНЬ ПРОСТОЕ МАШИННОЕ ОБУЧЕНИЕ.....</b>	<b>242</b>
<b>Глава 7. Классификация с помощью k-ближайших соседей.....</b>	<b>243</b>
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
<b>Глава 8. Регрессия с помощью k ближайших соседей .....</b>	<b>260</b>
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
<b>Послесловие</b> .....	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
<b>Приложение. Побитовые операции</b> .....	276
Обзор двоичной системы счисления .....	276
Распространенные побитовые операции .....	278
Сдвиг влево (<<) .....	278
Сдвиг вправо (>>) .....	279
ИЛИ ( ) .....	280
И (&) .....	281
XOR (^) .....	282
Дополнение (~) .....	283
<b>Предметный указатель</b> .....	284

## Об авторе

**Дэвид Копек** (David Kores) – 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.

# 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 сопровождающем 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](mailto:csfromscratch@oaksnow.com) или через X [@davekopes](https://twitter.com/davekopes). 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**



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<sup>1</sup>. 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.

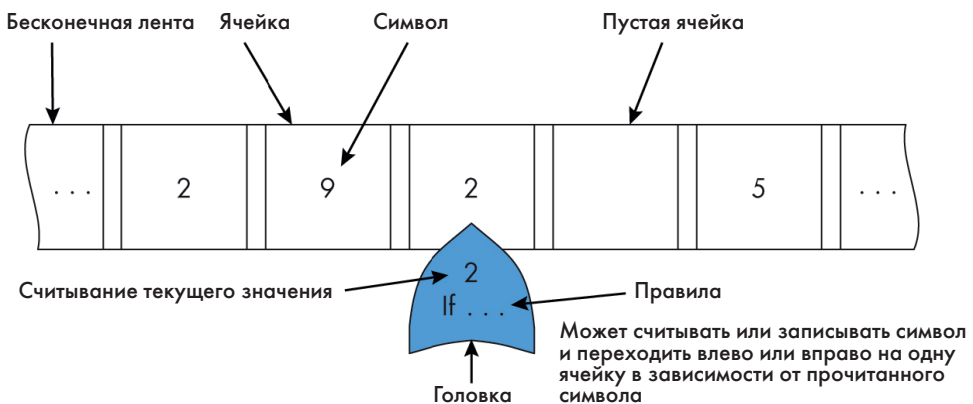


Рис. 1.1. Гипотетическая машина Тьюринга с бесконечной лентой, ячейками и следующей правилам головкой

<sup>1</sup> 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 Нуан (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<sup>1</sup>.

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,

---

<sup>1</sup> 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`.

### С в сравнении с Python

В реализации интерпретатора Brainfuck на языке С 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 С, но я включаю этот момент для иллюстрации различий между языками программирования. В языке С 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 С 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 С также будет работать намного быстрее, чем интерпретатор Brainfuck на Python. Основной интерпретатор для самого Python, CPython, написан на С.

В табл. 1.1 приведены данные из доклада Мюллера 2017 года, где описаны команды с обозначением каждого символа<sup>1</sup>. Я перевел описания на С на Python с использованием только что описанных имен переменных.

<sup>1</sup> 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 пройти 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 предв-  
 ать 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<sup>1</sup>. 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 иницирует 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.

---

<sup>1</sup> «bf core», Esolangs.org, посещение 22 мая 2024 г., [https://esolangs.org/wiki/Bf\\_core](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 состояниями. 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 `clamp_0_255_wgrayscale()`. 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 местопо-

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_wgaraound()`, 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_wgaraound()` с помощью нескольких простых условий:

---

```
# Имитация 1-байтового целого числа без знака
def clamp0_255_wgaraound(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 часть 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 транспилатор – `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.