PRINSIP DASAR ILMU KOMPUTER :

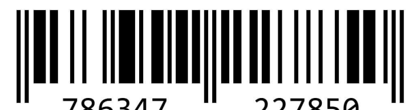
Interpreter, Seni Komputasi, Algoritma, Emulator, Pembelajaran Mesin (ML in Python)

Dr. Ir. Agus Wibowo, M.Kom, M.Si, MM

PRINSIP DASAR ILMU KOMPUTER :

Interpreter, Seni Komputasi, Algoritma, Emulator, Pembelajaran Mesin (ML in Python)

PENERBIT :
YAYASAN PRIMA AGUS TEKNIK
Jl. Majapahit No. 605 Semarang
Telp. (024) 6723456. Fax. 024-6710144
Email : penerbit_ypat@stekom.ac.id


ISBN 978-634-7227-85-0 (PDF)
9 786347 227850

PRINSIP DASAR ILMU KOMPUTER :
Interpreter, Seni Komputasi, Algoritma, Emulator, Pembelajaran Mesin (ML in Python)

Penulis :
Dr. Ir. Agus Wibowo, M.Kom, M.Si, MM

ISBN : 978-634-7227-85-0 (PDF)

Editor :
Dr. Joseph Teguh Santoso, S.Kom., M.Kom.

Penyunting :
Dr. Mars Caroline Wibowo. S.T., M.Mm.Tech

Desain Sampul dan Tata Letak :
Irdha Yunianto, S.Ds., M.Kom

Penebit :
Yayasan Prima Agus Teknik Bekerja sama dengan
Universitas Sains & Teknologi Komputer (Universitas STEKOM)

Anggota IKAPI No: 279 / ALB / JTE / 2023

Redaksi :
Jl. Majapahit no 605 Semarang
Telp. 08122925000
Fax. 024-6710144
Email : penerbit_ypat@stekom.ac.id

Distributor Tunggal :
Universitas STEKOM
Jl. Majapahit no 605 Semarang
Telp. 08122925000
Fax. 024-6710144
Email : info@stekom.ac.id

Hak cipta dilindungi undang-undang
Dilarang memperbanyak karya tulis ini dalam bentuk dan dengan cara apapun tanpa ijin dari penulis

KATA PENGANTAR

Dalam era digital yang terus berkembang pesat, pemahaman mendalam tentang prinsip dasar ilmu komputer menjadi fondasi esensial bagi siapa pun yang ingin berkontribusi dalam inovasi teknologi. Buku *PRINSIP DASAR ILMU KOMPUTER: Interpreter, Seni Komputasi, Algoritma, Emulator, Pembelajaran Mesin (ML in Python)* ini hadir sebagai panduan praktis yang dirancang untuk membawa pembaca menyelami inti komputasi melalui pendekatan hands-on, mulai dari bahasa pemrograman terkecil hingga aplikasi pembelajaran mesin modern.

Buku ini lahir dari pengamatan bahwa banyak materi ilmu komputer cenderung abstrak dan teoritis, padahal esensinya justru terletak pada implementasi nyata. Melalui delapan bab yang terstruktur secara bertahap, pembaca akan diajak membangun interpreter Brainfuck dan Nanobasic, mengeksplorasi seni komputasi seperti dithering citra retro dan algoritma melukis stokastik, serta mengemulasi mesin virtual Chip-8 dan konsol NES. Puncaknya, kita akan menerapkan algoritma K-Nearest Neighbors (KNN) untuk klasifikasi dan regresi, dilengkapi operasi bitwise sebagai pondasi logika biner. Semua ini diimplementasikan menggunakan Python, bahasa yang powerful namun mudah diakses, sehingga cocok untuk mahasiswa, pengembang pemula, maupun praktisi yang ingin memperdalam keterampilan komputasional. Tujuan utama buku ini adalah membekali pembaca dengan kemampuan berpikir komputasional yang tajam: dari memahami struktur interpreter hingga menciptakan emulator game retro dan model ML sederhana.

Setiap bab dilengkapi kode lengkap, pengujian, dan contoh nyata, sehingga pembaca tidak hanya memahami *mengapa* sesuatu bekerja, tapi juga *bagaimana* membangunnya sendiri. Ini bukan sekadar teori; ini adalah perjalanan membangun fondasi ilmu komputer yang kokoh di era Industry 4.0.

Buku ini terbagi menjadi sembilan bab yang saling terkait, membangun pemahaman bertahap. Bab 1 Bahasa Pemrograman Terkecil yang Mungkin memperkenalkan Brainfuck sebagai esensi komputasi minimalis, membahas struktur interpreter, dan panduan implementasinya di Python—fondasi untuk memahami bagaimana kode diterjemahkan menjadi aksi.

Bab 2 memformalkan sintaksnya, mengimplementasikan parser dan executor, serta pengujiannya, sehingga pembaca mampu membangun interpreter fungsional sendiri. Selanjutnya Bab 3 tetang Pemrosesan Citra Retro mengeksplorasi dithering, format MacPaint, pengkodean byte-bit, dan Run-Length Encoding (RLE), dengan pengujian praktis untuk seni komputasi visual era awal.

Bab 4 membahas opsi baris perintah, format SVG, dan algoritma stokastik untuk menghasilkan seni digital acak tapi indah, lengkap dengan penggunaan nyata. Bab 5 menguraikan konsep mesin virtual, spesifikasi Chip-8, argumen CLI, dan penggunaannya—langkah awal menuju emulasi sistem retro. Bab 6 memberikan gambaran arsitektur NES, pengembangan emulator lengkap (termasuk tabel lompatan, step(), pola ubin, palet warna), serta pengujian game asli.

Bab 7 menjelaskan kebangkitan ML (*Machine Learning*), mekanisme KNN, implementasinya, dan aplikasi klasifikasi angka tulisan tangan—pengenalan intuitif ke pembelajaran mesin. Bab 8 melanjutkan dengan prinsip regresi KNN dan implementasinya praktis, memperluas toolkit ML dasar. Bab 9 menutup dengan tinjauan biner dan operasi bitwise umum, sebagai pondasi logika rendah-level yang esensial untuk semua topik sebelumnya.

Melalui pendekatan ini, buku tidak hanya menyampaikan teori, tapi membekali keterampilan langsung: dari membangun interpreter hingga emulator NES dan model ML. Ideal untuk mahasiswa ilmu komputer, pengembang IoT, atau siapa pun yang haus akan komputasi kreatif di era AI. Terima kasih kepada komunitas programmer open-source atas inspirasi tak ternilai. Semoga buku ini memicu gelombang inovasi baru bagi para pembaca.

Semarang, Februari 2026

Penulis

Dr. Ir. Agus Wibowo, M.Kom, M.Si, MM

DAFTAR ISI

KATA PENGANTAR ..... **ii**
DAFTAR ISI ..... **iv**
BAB 1 BAHASA PEMROGRAMAN TERKECIL YANG MUNGKIN **1**
1.1 Apa Itu Brainfuck? 1
1.2 Struktur Interpreter 6
1.3 Mengimplementasikan Brainfuck Dalam Python 7
BAB 2 MENULIS INTERPRETER DASAR **18**
2.1 Memahami Nanobasic 18
2.2 Memformalkan Sintaks Nanobasic 25
2.3 Implementasi Nanobasic 29
2.4 Menjalankan Program 52
2.5 Pengujian Nanobasic 53
BAB 3 PEMROSESAN CITRA RETRO **59**
3.1 Apa Itu Dithering? 59
3.2 Algoritma Dithering 63
3.3 Format File MacPaint 67
3.4 Menerjemahkan Byte Ke Bit 68
3.5 Menerapkan Pengkodean Panjang Berurutan (*Run-Length Encoding*) 70
3.6 Pengujian Pengkodean Panjang Berurutan 74
BAB 4 ALGORITMA MELUKIS STOKASTIK **85**
4.1 Pendahuluan 85
4.2 Opsi Baris Perintah 89
4.3 Format SVG 91
4.4 Algoritma Stochastic 92
4.5 Penggunaan Algoritma Melukis Stokastik 100
BAB 5 MEMBANGUN MESIN VIRTUAL CHIP-8 ..... **107**
5.1 Mesin Virtual 107
5.2 Mesin Virtual Chip-8 108
5.3 Argumen Baris Perintah 117
5.4 Penggunaan VM 128
BAB 6 MENGEMULSI KONSOL GAME NES **133**
6.1 Gambaran Umum NES 134
6.2 Pengembangan Emulator 136
6.3 Tabel Lompatan 148
6.4 Metode `Step()` 162
6.5 Metode Pembantu 168
6.6 Tabel Pola Dan Ubin 171

6.7 Palet Warna Dan Tabel Atribut 174
6.8 Pengujian Emulator Game Nes 187
BAB 7 KLASIFIKASI DENGAN K-NEAREST TETANGGA 196
7.1 Kebangkitan Pembelajaran Mesin 196
7.2 Cara Kerja Knn 197
7.3 Mengimplementasikan Klasifikasi Dengan Knn 199
7.4 Klasifikasi Angka Tulisan Tangan 205
BAB 8 REGRESI DENGAN KNN 211
8.1 Cara Kerja Regresi Knn 211
8.2 Menerapkan Regresi Dengan Knn 213
BAB 9 OPERASI BITWISE 219
9.1 TINJAUAN BINER 219
9.2 OPERASI BITWISE UMUM 221
DAFTAR PUSTAKA 227

BAB 1
BAHASA PEMROGRAMAN TERKECIL YANG MUNGKIN

Apa bahasa pemrograman terkecil yang masih dapat digunakan untuk memecahkan masalah nyata? Tentu saja, kandidat untuk penghargaan itu adalah Brainfuck, bahasa pemrograman esoterik yang dikembangkan oleh Urban Müller pada tahun 1993.

Dalam bab ini, kita akan mengembangkan interpreter Brainfuck. Brainfuck mungkin merupakan bahasa pemrograman termudah untuk ditulis interpreter-nya—Anda akan kagum betapa ringkasnya interpreter kita. Pada akhir bab ini, Anda tidak hanya akan mempelajari cara kerja Brainfuck tetapi juga prinsip-prinsip inti dari interpreter apa pun.

1.1 APA ITU BRAINFUCK?

Brainfuck hanya memiliki delapan perintah (+, -, ., ,, >, <, [,]), dan setiap perintah adalah satu karakter. Berikut adalah "Hello World!" di Brainfuck:

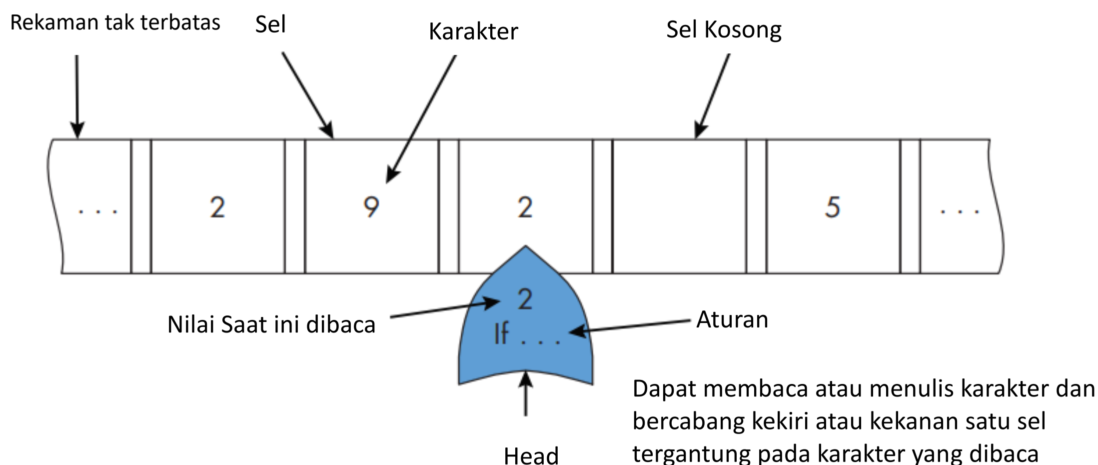
```
++++++++[>++++[>++>+++>+++>+<<<<-]>+>+>->>+[<]<-]>>.>---.+++++++..+++.>>.<-.<.+++.-
-----.-----\---.>>+.>++.
```

Mungkin terlihat aneh, tetapi itu adalah program yang sebenarnya. Sintaks eksotis Brainfuck dan fitur minimalnya membuatnya tidak cocok untuk tujuan praktis apa pun. Sebaliknya, ini adalah mainan yang berguna sebagai model pendidikan. Tetapi ini adalah mainan yang Turing-lengkap!

Apa yang Membuat Suatu Bahasa Turing-Lengkap?

Suatu bahasa pemrograman dianggap Turing-lengkap jika dapat mensimulasikan mesin Turing, model abstrak dari mesin yang dapat mengimplementasikan algoritma komputer apa pun. Untuk membayangkan mesin Turing, bayangkan sebuah pita dengan panjang tak terbatas, dibagi menjadi sel-sel yang kosong atau memiliki karakter di dalamnya. Kemudian, bayangkan sebuah kepala yang dapat membaca atau menulis karakter di dalam sel, termasuk menghapusnya. Bayangkan bahwa kepala dapat bergerak ke kiri atau ke kanan satu sel pada satu waktu. Terakhir, bayangkan bahwa ia dapat menulis atau bergerak berdasarkan nilai yang dibaca—yaitu, bahwa ia dapat bercabang. Dengan kata lain, kepala mesin mengikuti beberapa aturan sederhana, yang dapat dianggap sebagai program yang pada dasarnya mengatakan, "Jika nilai ini dibaca, tulis nilai lain ini. Jika nilai itu dibaca, pindah ke kiri satu sel."

Itu saja. Itu sudah cukup untuk dapat mengimplementasikan algoritma komputasi apa pun. Lebih jauh lagi, bahasa pemrograman apa pun yang dapat mensimulasikan fungsi ini—bahkan bahasa sederhana seperti Brainfuck—dapat digunakan untuk memecahkan masalah nyata. Gambar 1-1 mengilustrasikan mesin Turing hipotetis.



Gambar 1.1. Mesin Turing hipotetis, termasuk pita tak terbatas, sel, dan kepala yang mengikuti aturan

Fitur apa yang dibutuhkan bahasa pemrograman agar menjadi Turing-lengkap? Seperti yang dijelaskan Allen Tucker dan Robert Noonan dalam buku mereka Programming Languages: Principles and Paradigms, tidak dibutuhkan banyak hal:

> *Suatu bahasa pemrograman dikatakan Turing-lengkap jika mengandung variabel, nilai, dan operasi bilangan bulat serta memiliki pernyataan penugasan dan konstruksi kontrol pengurutan pernyataan, kondisional, dan pernyataan percabangan. Semua bentuk pernyataan lainnya (loop while dan for, pemilihan kasus, deklarasi dan panggilan prosedur, dll.) dan tipe data (string, nilai floating point, dll.) disediakan dalam bahasa modern hanya untuk meningkatkan kemudahan pemrograman berbagai aplikasi kompleks.*

Izinkan penulis menyederhanakan deskripsi tersebut lebih lanjut menggunakan bahasa yang lebih sederhana. Suatu bahasa pemrograman membutuhkan variabel bilangan bulat, cara untuk mengubah nilai yang terkait dengan variabel tersebut, sesuatu seperti pernyataan if, dan sesuatu seperti pernyataan goto ("lompatan") agar menjadi Turing-lengkap. Itu tidak banyak. Dan jika Anda memikirkannya, Anda dapat membayangkan bagaimana konstruksi sederhana itu dapat dipetakan ke elemen-elemen mesin Turing. Variabel integer adalah karakter pada sel, mengubah variabel adalah kepala yang menulis pada sel, dan pernyataan if dan goto mewakili percabangan kepala.

Bahasa pemrograman apa pun yang Turing-lengkap dapat mengimplementasikan algoritma yang sama dengan bahasa pemrograman lain yang Turing-lengkap. Anda dapat mengimplementasikan Quicksort dalam C, tetapi Anda juga dapat mengimplementasikan Quicksort dalam Brainfuck. Anda dapat menulis parser JSON dalam Python, tetapi Anda juga dapat menulis parser JSON dalam Brainfuck. Dalam pengertian ini, meskipun Brainfuck adalah bahasa pemrograman "mainan", ia juga merupakan bahasa pemrograman "nyata".

Cara Kerja Brainfuck

Keadaan utama dalam program Brainfuck adalah array integer. Setiap slot dalam array disebut sel. Array sel dapat dianggap analog dengan pita dalam mesin Turing. Alih-alih karakter yang dibaca atau ditulis pada sel, yang digunakan adalah bilangan bulat. Perintah dalam Brainfuck memungkinkan programmer untuk bergerak maju satu sel (>), bergerak mundur satu sel (<), menambah nilai sel (+), mengurangi nilai sel (-), mengeluarkan sel (.), memasukkan data ke sel (,), dan melakukan perulangan selama sel tertentu bukan nol ([dan]).

Beberapa operasi ini dipetakan langsung ke operasi mesin Turing, sehingga Brainfuck bersifat Turing-lengkap. Dalam Python, kita akan membutuhkan dua variabel untuk menyimpan status sel: daftar semua sel (cells) dan bilangan bulat yang mewakili indeks sel saat ini (cell_index). Selain itu, kita perlu melacak posisi kita dalam file sumber Brainfuck. Kita akan menangani ini dengan bilangan bulat lain, yang disebut instruction_index.

C VS. PYTHON

Dalam implementasi interpreter Brainfuck berbasis C, sel-sel dapat dikelola oleh satu pointer ke array bilangan bulat. Pointer dapat digunakan untuk menginisialisasi memori untuk sel dan untuk berpindah dari sel ke sel, dan dapat di-dereferensi untuk mengubah nilai sel.

Beberapa dari Anda mungkin tidak mengenal C, tetapi penulis menyertakan poin ini untuk mengilustrasikan perbedaan antara bahasa pemrograman. Dalam C, berkat kekuatan pointer, kita memiliki konstruksi variabel tunggal, sedangkan kita membutuhkan banyak variabel dalam interpreter Brainfuck Python kita untuk fungsi yang sama.

Pointer C sangat ampuh, tetapi juga berbahaya dan sulit dipahami bagi programmer baru. Tentu saja, bahasa pemrograman yang berbeda memiliki kompromi yang berbeda untuk mengimplementasikan interpreter. Interpreter Brainfuck berbasis C juga akan jauh lebih cepat daripada interpreter Brainfuck berbasis Python. Interpreter utama untuk Python itu sendiri, CPython, ditulis dalam C.

Tabel 1.1 didasarkan pada tabel dari presentasi tahun 2017 oleh Müller yang menjelaskan perintah yang diwakili oleh setiap karakter. penulis telah menerjemahkan deskripsi C ke Python menggunakan nama variabel yang baru saja dibahas.

Tabel 1.1. Perintah Brainfuck

Perintah	Ekuivalen Python	Keterangan
>	`cell_index += 1`	Geser satu sel ke kanan.
<	`cell_index -= 1`	Geser satu sel ke kiri.
+	`cells[cell_index] += 1`	Tingkatkan sel saat ini.
–	`cells[cell_index] -= 1`	Kurangi sel saat ini.

.	`print(chr(cells[cell_index]), end='', flush=True)`	Cetak nilai ASCII dari sel saat ini.
,	`cells[cell_index] = int(input())`	Nilai baca untuk sel saat ini.
[	`if cells[cell_index] == 0: instruction_index = self.find_bracket_match(instruction_index,True)`	Jika sel tersebut bernilai nol, pindah ke tanda kurung tutup yang sesuai.
]	`if cells[cell_index] != 0: instruction_index = self.find_bracket_match(instruction_index,False)`	Jika sel tersebut bukan nol, pindah ke tanda kurung buka yang sesuai.

Dengan menggunakan Tabel 1-1, kita memiliki cukup informasi untuk menelusuri program Brainfuck dan memahami apa yang dilakukannya. Kita akan mempertimbangkan program sederhana yang mengeluarkan satu karakter yang ditentukan pengguna sebanyak jumlah yang ditentukan pengguna. Berikut adalah keseluruhan program, dengan setiap perintah diberi label indeks sehingga kita dapat merujuknya nanti:

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

Mari kita bahas program ini satu perintah demi satu. Kita akan menjelaskan cara kerja setiap perintah dan mengilustrasikan pengaruhnya terhadap status program menggunakan tabel. Program ini hanya membutuhkan dua sel, ditambah indeks sel dan indeks instruksi. Tabel awalnya terlihat seperti ini:

Cell 0	Cell 1	Cell Index	Instruksi Index
0	0	0	0

Berikut perintah pertama (untuk kejelasan, kita akan mendahului setiap perintah dengan indeks instruksinya dan titik dua):

```
0; ,
```

Input pengguna diambil dan disimpan di sel 0 karena indeks sel awalnya adalah 0. Untuk contoh kita, bayangkan pengguna memasukkan 88 (kode karakter ASCII untuk huruf kapital X). Setelah perintah dijalankan, indeks instruksi akan bertambah.

Cell 0	Cell 1	Cell Index	Instruksi Index
88	0	0	1

```
1: >
```

Indeks sel dinaikkan, lalu indeks instruksi juga dinaikkan.

Cell 0	Cell 1	Cell Index	Instruksi Index
88	10	1	2

```
2: ,
```

Input pengguna dimasukkan ke dalam sel 1. Misalnya, pengguna memasukkan 10. Kemudian, indeks instruksi akan bertambah.

Cell 0	Cell 1	Cell Index	Instruksi Index
88	10	1	3

```
3; [
```

Sebuah perulangan berpotensi dimulai. Karena nilai pada indeks sel saat ini bukan 0 (melainkan 10), alih-alih melompat ke tanda kurung penutup yang sesuai, kita cukup menambah indeks instruksi sebesar 1 untuk melanjutkan ke perintah berikutnya.

Cell 0	Cell 1	Cell Index	Instruksi Index
88	10	1	4

```
4: <
```

Indeks sel dikurangi. Kemudian indeks instruksi ditambah.

Cell 0	Cell 1	Cell Index	Instruksi Index
88	10	0	5

```
5: .
```

Nilai ASCII dari sel pada indeks sel saat ini akan ditampilkan ke konsol—dalam hal ini, sebuah X. Kemudian, indeks instruksi akan ditambah.

Cell 0	Cell 1	Cell Index	Instruksi Index
88	10	0	6

```
6: >
```

Indeks sel dinaikkan. Kemudian, indeks instruksi dinaikkan.

Cell 0	Cell 1	Cell Index	Instruksi Index

88	10	1	7

```
7: -
```

Nilai pada indeks sel saat ini dikurangi. Indeks instruksi ditambah.

Cell 0	Cell 1	Cell Index	Instruksi Index
88	9	1	8

```
8: ]
```

Jika nilai pada indeks sel saat ini bukan nol, kita melompat ke tanda kurung buka yang sesuai. Dalam kasus ini, sel 1 adalah 9, jadi kita melompat, yang berarti indeks instruksi menjadi 3.

Cell 0	Cell 1	Cell Index	Instruksi Index
88	9	1	3

Sekarang kita telah menyelesaikan iterasi pertama dari sebuah loop yang akan diulang sembilan kali untuk mencetak total 10 huruf X. Instruksi 3 hingga 8 akan mengulanginya sembilan kali hingga sel 1 bernilai 0, dalam hal ini pemeriksaan pada tanda kurung penutup (indeks 8) akan mengakhiri pengulangan dan program akan selesai.

Untuk memberikan penutup pada program ini, mari kita langsung menjalankannya melalui interpreter kita:

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

Interpreter Brainfuck kami hanya menerima bilangan bulat sebagai input. Bilangan bulat ini dapat sesuai dengan kode karakter ASCII, dan 88 adalah kode karakter ASCII untuk X. Oleh karena itu, output yang diharapkan adalah 10 X. Setelah Anda menyelesaikan bab ini, Anda akan dapat menjalankan program sendiri, serta program Brainfuck lainnya.

1.2 STRUKTUR INTERPRETER

Interpreter umumnya memiliki setidaknya tiga bagian:

- Tokenizer (kadang-kadang dikenal sebagai lexer) yang mengambil kode sumber asli dan membaginya menjadi konstruksi terkecil yang dapat dikenali yang diizinkan dalam bahasa pemrograman. Ini dikenal sebagai token. Untuk kode `a + 2`, tokennya mungkin `a`, `+`, dan 2.

- Parser yang mengambil token yang berdekatan dan mencari tahu artinya (yaitu, ekspresi atau pernyataan yang dibentuknya). Parser biasanya menghasilkan pohon node yang mewakili hubungan relatif antara ekspresi, pernyataan, dan nilai literal. Pohon ini disebut pohon sintaks abstrak (AST). Misalnya, jika interpreter Python melihat token a diikuti oleh token + diikuti oleh token 2, ia dapat membangun node ekspresi aritmatika dan menghubungkannya ke node untuk a dan 2.
- Lingkungan runtime yang menelusuri node AST dan menjalankan operasi yang sesuai untuk mengeksekusi makna yang terkandung di dalamnya. Untuk node ekspresi aritmatika a + 2 kita, ini berarti mencari nilai yang diwakili oleh a dan menambahkan 2 ke dalamnya.

Hal yang indah tentang Brainfuck adalah setiap pernyataan hanyalah simbol tunggal, jadi yang perlu kita lakukan untuk mendapatkan token hanyalah membaca satu karakter dari file sumber. Dan setiap token itu sendiri sudah mewakili node makna. Itu membuat penulisan interpreter untuk Brainfuck lebih mudah daripada menulis interpreter untuk hampir semua bahasa pemrograman lainnya. Kita dapat menggabungkan tokenizer, parser, dan runtime ke dalam satu loop yang menggabungkan ketiga konsep tersebut.

Kita akan kembali membahas ide tokenizer, parser, dan runtime yang terpisah di Bab 2, di mana kita akan membangun interpreter untuk bahasa yang sedikit lebih rumit yang disebut NanoBASIC. Akan sangat ilustratif untuk melihat bagaimana bagian-bagian yang digabungkan dalam interpreter Brainfuck kita dipisahkan untuk interpreter NanoBASIC kita.

Interpreter Brainfuck tidak hanya mudah ditulis tetapi juga sangat ringkas.

Terinspirasi oleh bahasa esoterik lain, yang disebut FALSE, tujuan Müller dengan Brainfuck adalah untuk menghasilkan bahasa minimal, dan dia tentu saja berhasil melakukannya. Interpreter aslinya untuk bahasanya dengan hanya delapan perintah hanya berukuran 240 byte. Lebih mengesankan lagi, ada interpreter Brainfuck yang ditulis dalam assembly x86 yang hanya berukuran 69 byte. Kita tidak akan sampai pada 69 byte, tetapi inti dari interpreter Python Brainfuck kita hanya akan terdiri dari 25 baris kode dan beberapa fungsi pembantu. Dan 25 baris tersebut akan mampu menjalankan program Brainfuck apa pun, dan karenanya algoritma komputer apa pun.

Implementasi Brainfuck asli Müller memiliki beberapa keterbatasan yang akan kita ulangi dalam interpreter kita. Alih-alih pita tak terbatas, seperti pada mesin Turing, Brainfuck asli dibatasi hingga 30.000 sel. Dan setiap sel tersebut hanya dapat menampung bilangan bulat tak bertanda 8-bit.

1.3 MENGIMPLEMENTASIKAN BRAINFUCK DALAM PYTHON

Sebelum kita masuk ke implementasi interpreter utama, mari kita lakukan sedikit persiapan. Setiap proyek yang kita kerjakan dalam buku ini akan terstruktur sebagai paket Python. Setiap paket akan berada di foldernya sendiri dengan file __main__.py yang memulai eksekusi saat proyek dijalankan dari baris perintah. Anda akan menemukan semua kode yang

diperlukan langsung di buku ini, kecuali dinyatakan lain. Setiap daftar kode muncul dengan nama file Python yang terkait dengannya, sehingga Anda dapat menemukan kode tersebut di repositori buku.

Mendapatkan File Sumber

File *__main__.py* utama kami bertanggung jawab untuk menerima argumen baris perintah yang berisi jalur file sumber Brainfuck dan meneruskannya ke interpreter utama:

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

if __name__ == "__main__":
   # Parse the file argument
   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()
```

Kelas pustaka standar `ArgumentParser` memudahkan penanganan argumen baris perintah. Kita akan menggunakannya di setiap proyek dalam buku ini. Dalam cuplikan ini, kita membuat satu argumen baris perintah, `brainfuck_file`, yang mewakili jalur file yang ingin kita muat sumber Brainfuck-nya. Tipe default argumen adalah string, jadi pada akhirnya kita akan meneruskan string jalur ke kelas Brainfuck kita, yang akan bertanggung jawab untuk membaca isinya. Untuk mempelajari lebih lanjut tentang ArgumentParser, lihat dokumentasi resmi argparse di *https://docs.python.org/3/library/argparse.html.*

Menulis Interpreter

Interpreter kita bertanggung jawab untuk mempertahankan status Brainfuck `(sel`, `cell_index`, dan `instruction_index)`. Ia juga perlu membaca setiap perintah Brainfuck yang valid dalam file sumber dan mengubah status atau menyelesaikan operasi input/output berdasarkan perintah tersebut. Karena perintah Brainfuck hanya berupa satu karakter, membacanya sangat mudah. Dan "apa yang harus dilakukan" dengan setiap karakter sebenarnya hampir identik dengan Tabel 1-1. Itulah mengapa kita akhirnya hanya memiliki satu fungsi `(execute())` dengan 25 baris kode.

```
# Brainfuck/brainfuck.py
from pathlib import Path

class Brainfuck:
   def __init__(self, file_name: str | Path):
      # Open text file and store in instance variable
      with open(file_name, "r") as text_file:
         self.source_code: str = text_file.read()
```

```
def execute(self):
    # Setup state
    cells: list[int] = [0] * 30000
    cell_index = 0
    instruction_index = 0
    # Keep going as long as there are potential instructions left
    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
```

Implementasi setiap perintah diambil langsung dari Tabel 1-1 dan terdiri dari manipulasi sederhana dari tiga variabel status. Bagi Anda yang belum mengikuti versi Python terbaru, pernyataan match ditambahkan di Python 3.10 dan dapat dianggap sebagai versi yang lebih canggih dari pernyataan switch dari bahasa lain. Pernyataan ini mengeksekusi bagian kode (atau case) yang cocok dengan nilai dalam variabel yang dicocokkan. Dalam program kita, case tersebut sesuai dengan nilai-nilai yang mungkin dari instruksi.

PETUNJUK TIPE

Anda mungkin memperhatikan penggunaan petunjuk tipe pada beberapa variabel lokal. penulis akan melakukannya di buku ini di mana penulis

pikir itu menambah kejelasan, tetapi penulis tidak akan terlalu kaku tentang hal itu. Misalnya, penulis pikir mengklarifikasi bahwa `cells` adalah `list[int]` masuk akal karena beberapa orang mungkin tidak ingat sintaks inisialisasi daftar yang penulis gunakan. Tetapi jelas bahwa `instruction` akan berupa `str` berdasarkan konteksnya, jadi penulis tidak memberikan petunjuk tipe untuk itu. Hal lain yang memungkinkan petunjuk tipe adalah menjalankan pemeriksa tipe statis untuk membantu memverifikasi kebenaran semua kode dalam buku ini. penulis selalu merasa itu bermanfaat.

Catatan singkat tentang sintaks petunjuk tipe `file_name: str | Path` yang digunakan dalam tanda tangan __init__(): sintaks tersebut berarti bahwa argumen yang diberikan diharapkan berupa tipe `str` atau tipe `Path`. Kedua tipe tersebut dapat diterima. ArgumentParser kami menyediakan jalur file sebagai string, sementara pengujian unit kami akan menyediakannya sebagai objek `Path`. Fungsi open() yang menggunakan path di __init__() dapat menerima keduanya.

Dua fungsi pembantu yang hilang dari Tabel 1.1 adalah: find_bracket_match() dan clamp0_255_wraparound(). Pertama, mari kita lihat find_bracket_match(), yang membantu melompat dari satu perintah kurung seperti pernyataan if ke pasangannya. Fungsi ini diimplementasikan sebagai metode pada kelas Brainfuck karena perlu mengakses self.source_code:

```
# Find the location of the corresponding bracket to the one at *start*.
# If *forward* is true go to the right looking for a matching "]".
# Otherwise do the reverse.
def find_bracket_match(self, start: int, forward: bool) -> int:
  ❶ in_between_brackets = 0
    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
    # Didn't find a match
    print(f"Error: could not find match for {start_bracket} at {start}.")
    return start
```

Untuk menemukan tanda kurung yang cocok, kita melakukan pencarian linier melalui kode sumber Brainfuck, melihat setiap karakter berikutnya satu per satu. Kita mencari ke kanan jika forward bernilai True atau ke kiri jika forward bernilai False. Variabel arah menjadi proksi untuk forward, baik menambah atau mengurangi lokasi untuk pergi ke kanan atau ke kiri ❶.

Faktor yang membingungkan saat mencari tanda kurung yang cocok adalah tanda kurung di antara, yaitu kumpulan tanda kurung yang muncul di antara tanda kurung awal dan tanda kurung yang kita inginkan. Misalnya, katakanlah kita mencari tanda kurung yang cocok dari tanda kurung pertama dalam cuplikan Brainfuck ini (saya telah memberi label karakter dengan indeks untuk kejelasan):

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

Pasangan untuk tanda kurung buka pada indeks 0 adalah tanda kurung tutup pada indeks 9. Namun, jika kita secara naif menerima tanda kurung tutup pertama yang kita temukan, pencarian kita akan menyimpulkan bahwa pasangan untuk indeks 0 adalah tanda kurung tutup pada indeks 6. Padahal, tanda kurung tutup tersebut cocok dengan tanda kurung buka pada indeks 3.

Solusinya cukup dengan menghitung tanda kurung di antara keduanya. Setiap kali kita menemukan awal dari pasangan tanda kurung di antara keduanya, kita menambah penghitung `in_between_brackets` sebanyak ❸. Setiap kali kita menemukan akhir dari pasangan tanda kurung di antara keduanya, kita mengurangi `in_between_brackets`, kecuali jika `in_between_brackets` adalah 0, yang berarti tidak ada lagi tanda kurung di antara keduanya dan tanda kurung tujuan telah ditemukan ❷.

ALTERNATIF UNTUK MENCARI TANDA KURUNG

Cara lain untuk menyelesaikan masalah tanda kurung di antara adalah dengan menggunakan tumpukan (stack). Setiap kali tanda kurung awal ditemukan, lokasinya dimasukkan ke dalam tumpukan. Setiap kali tanda kurung akhir ditemukan, tumpukan dikeluarkan (pop). Dua lokasi tanda kurung yang dihasilkan (lokasi tanda kurung akhir yang ditemukan dan tanda kurung awal yang dikeluarkan) adalah pasangan.

Dengan menggunakan metode ini, Anda dapat menelusuri seluruh kode sumber sekaligus dan menemukan semua pasangan tanda kurung dengan mudah. Lokasi pasangan tanda kurung tersebut kemudian dapat di-cache untuk meningkatkan kinerja interpreter. Alih-alih menjalankan pencarian linier seperti pada `find_bracket_match()` setiap kali lompatan diperlukan, menemukan tanda kurung lainnya (lokasi lompatan) hanya menjadi pencarian dari cache.

Fungsi pembantu lainnya, `clamp0_255_wraparound()`, mensimulasikan Brainfuck asli dengan membatasi nilai sel ke bilangan bulat tak bertanda 8-bit. Kita membutuhkan fungsi ini karena tipe int Python memiliki presisi arbitrer, artinya dapat mengakomodasi bilangan bulat sebesar apa pun yang Anda inginkan tanpa overflow (sebaliknya, lebih banyak byte diambil sesuai kebutuhan). Bilangan bulat tak bertanda 8-bit yang sebenarnya akan kembali ke 0 setelah melebihi 255 sebanyak 1, dan akan kembali ke 255 jika berada di 0 dan dikurangi 1. Kita mensimulasikan perilaku ini di `clamp0_255_wraparound()` dengan beberapa kondisi sederhana:

```
# Simulate a 1-byte unsigned integer
  def clamp0_255_wraparound(num: int) -> int:
    if num > 255:
      return 0
    elif num < 0:
      return 255
    else:
      return num
```

Karena Brainfuck tidak dapat mengubah sel lebih dari 1 pada satu waktu, kita tidak perlu khawatir tentang kasus di mana kita menambahkan lebih dari 1 ke sel yang bernilai 255 atau mengurangi lebih dari 1 dari sel yang bernilai 0. Oleh karena itu, pengujian `num > 255` dan `num < 0` sudah cukup.

Dengan dua fungsi pembantu ini, implementasi interpreter Brainfuck kita sudah lengkap. Sebenarnya tidak perlu banyak usaha untuk mengimplementasikan bahasa yang Turing-complete.

Menjalankan Interpreter

Mari kita coba menjalankan beberapa kode Brainfuck. Folder Brainfuck di repositori buku ini memiliki subfolder Examples dengan beberapa program contoh untuk diinterpretasikan, termasuk fibonacci.bf untuk menghasilkan beberapa anggota pertama dari deret Fibonacci dan `hello_world_verbose.bf` yang berisi program `"Hello World!"` yang ditunjukkan sebelumnya di bab ini. Di sini, penulis menjalankan program-program tersebut dari direktori utama repositori:

```
% 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!
```

Anda harus menjalankan perintah-perintah ini dengan opsi `-m` yang menunjukkan bahwa Brainfuck harus dipahami sebagai sebuah modul. Jika tidak, Anda akan menerima kesalahan impor. Perhatikan juga bahwa cara Python diakses dari shell akan berbeda tergantung pada sistem operasi dan jenis instalasi Python. Pada sistem saya, interpreter Python memiliki alias

python3 dan jalur menggunakan garis miring. Sistem Anda mungkin menggunakan python dan garis miring terbalik (gaya Windows). Sepertinya interpreter kita berfungsi, tetapi kita harus membuat beberapa pengujian untuk memastikannya.

Menguji Interpreter

Mari kita tulis beberapa pengujian untuk memastikan interpreter kita berfungsi dengan benar. Kita dapat mulai dengan menulis beberapa pengujian unit untuk mengkonfirmasi setiap perintah individual dari interpreter berfungsi seperti yang diharapkan. Apakah + berfungsi dengan benar? Apakah . berfungsi dengan benar? Namun, demi singkatnya (dan karena interpreter sangat sederhana), kita akan menulis beberapa pengujian integrasi. Pengujian ini memeriksa apakah seluruh program Brainfuck berjalan dengan benar melalui interpreter, menghasilkan output yang diharapkan.

Untuk mempermudah pengaturan integrasi berkelanjutan, pengujian untuk seluruh buku berada di folder tersendiri di dalam root repositori utama, yang disebut tests. Pengujian kami untuk Brainfuck akan menjalankan seluruh program Brainfuck melalui interpreter, menangkap output teksnya, dan membandingkan output tersebut dengan output yang diharapkan.

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

# Tokenizes, parses, and interprets a Brainfuck
# program; stores the output in a string and returns it
def run(file_name: str | Path) -> str:
     output_holder = StringIO()
     sys.stdout = output_holder
     Brainfuck(file_name).execute()
     return output_holder.getvalue()
```

Fungsi `run()` menginisialisasi kelas Brainfuck dengan sebuah file yang terletak di `file_name`. Fungsi ini juga menggunakan `output_holder` untuk menangkap dan mengembalikan `stdout`, yang berarti bahwa alih-alih output dari program yang dijalankan masuk ke konsol, output tersebut akan diberikan ke sebuah variabel. Hal ini memungkinkan kita untuk membandingkan output aktual dengan output yang diharapkan secara terprogram setelah memanggil `run()` di setiap pengujian kita:

```
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()
```

Setiap pengujian mengambil program Brainfuck di direktori Examples, menggunakan `run()` untuk mengeksekusinya, dan membandingkan output akhir dengan output yang diharapkan melalui `assertEqual()`. Mari kita coba menjalankan semua pengujian dari direktori utama repositori:

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

Jika interpreter Brainfuck kita berhasil menjalankan empat program yang sangat berbeda satu sama lain, ada kemungkinan besar interpreter tersebut berfungsi. Di repositori online untuk buku ini, penulis telah menyiapkan integrasi berkelanjutan sehingga pengujian ini secara otomatis berjalan setiap kali kode diubah. Sebagian besar bab dalam buku ini memiliki pengujian unit atau integrasi yang juga berjalan secara otomatis.

KODE BERTEMU KEHIDUPAN

Saya pertama kali mendengar tentang Brainfuck sudah lama sekali sebagai rasa ingin tahu, tetapi penulis benar-benar tertarik padanya pada tahun 2018 ketika

mempersiapkan diri untuk mengajar kelas yang disebut Bahasa-Bahasa Baru di Champlain College. Ini adalah kelas teori bahasa pemrograman dengan sentuhan berbeda: kami menggunakan bahasa-bahasa yang, pada tahun 2018, baru mulai relevan di industri—yaitu, Go, Swift, dan Clojure—untuk mengilustrasikan ide-ide pemrograman.

Saya mengembangkan kursus ini bersama seorang kolega bernama Josh Auerbach. penulis membuat bagian Go dan Swift dari kelas tersebut, dan Josh mengembangkan bagian Clojure.
Kami berdua menyukai ide untuk mengerjakan tugas Brainfuck, karena ini adalah alat pendidikan yang hebat untuk memahami cara kerja interpreter sederhana. Josh memiliki ide untuk menggunakan sebagian dari tugas Brainfuck untuk mengajarkan makro Clojure.

Kami menggunakan makro Clojure untuk menjelaskan gagasan homoikonisitas dalam Lisp (Clojure adalah dialek Lisp)—yaitu, konsep bahwa "kode adalah data." Dalam makro Clojure, Anda dapat memanipulasi kode sebelum dijalankan, memperlakukannya seperti data lain dalam program Clojure. Makro yang dikembangkan siswa dalam tugas Josh memungkinkan seseorang untuk menulis kode Brainfuck langsung dalam Clojure dan menjalankannya seolah-olah memang seharusnya ada di sana.

Anda bisa berada di tengah program Clojure Anda dan menulis sesuatu seperti ini:

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

Saya masih menggunakan tugas tersebut saat mengajar di kelas (Josh sudah meninggalkan dunia akademis), tetapi penulis akui terkadang penulis kesulitan mengingat sintaks untuk menulis makro Clojure. Menulis interpreter Brainfuck bahkan lebih mudah daripada menulis makro. Itulah mengapa Brainfuck adalah alat yang hebat untuk para pendidik.

Aplikasi Dunia Nyata

Setiap bab dalam buku ini diakhiri dengan beberapa aplikasi dunia nyata, tetapi sayangnya tidak ada aplikasi dunia nyata dari Brainfuck. Ini adalah bahasa rasa ingin tahu, berguna untuk mempelajari beberapa ide fundamental dalam ilmu komputer. Mungkin, oleh karena itu, kita dapat mengatakan bahwa aplikasi dunia nyata dari Brainfuck adalah dalam bidang pendidikan.

Interpreter secara lebih luas merupakan infrastruktur komputasi penting dengan banyak aplikasi dunia nyata. Seperti yang mungkin Anda ketahui, Python sendiri adalah bahasa yang diinterpretasikan. Ada banyak cara bahasa pemrograman beralih dari file teks ke kode mesin, tetapi kita dapat secara luas mengkategorikan sebagian besar implementasi bahasa pemrograman sebagai diinterpretasikan, dikompilasi ahead-of-time, atau dikompilasi just-in-time. Beberapa bahasa pemrograman bahkan memiliki implementasi dalam ketiga kategori tersebut. Misalnya, ada interpreter Java, compiler Java ahead-of-time, dan implementasi Java yang paling populer adalah yang dikompilasi just-in-time.

Sebagai aturan umum, implementasi bahasa pemrograman yang diinterpretasikan cenderung lebih lambat daripada implementasi bahasa pemrograman yang dikompilasi. Anda mungkin bertanya-tanya, mengapa bahasa pemrograman yang digunakan orang di dunia nyata diimplementasikan sebagai interpreter. Ambil contoh Python: Mengapa diimplementasikan menggunakan interpreter alih-alih compiler? Kita semua tahu Python adalah bahasa yang relatif lambat, dan tentu saja akan lebih cepat jika dikompilasi.

Jawabannya adalah bahwa banyak fitur dinamis runtime Python tidak akan mungkin, atau setidaknya akan sangat sulit untuk diimplementasikan, selain dengan interpreter. Ada upaya untuk melakukannya (PyPy, misalnya), tetapi jauh lebih sulit untuk melakukannya dengan benar.

Selain itu, interpreter jauh lebih mudah diimplementasikan daripada compiler karena tidak memiliki seluruh fase backend compiler yang bertanggung jawab untuk menghasilkan kode mesin. Oleh karena itu, banyak bahasa dimulai sebagai interpreter karena itu adalah cara tercepat untuk membuat implementasi berjalan. Misalnya, versi pertama Java adalah interpreter, dan butuh beberapa tahun sebelum versi kompilasi just-in-time keluar.

Singkatnya, interpreter ada karena lebih mudah diimplementasikan daripada compiler dan karena memungkinkan fitur runtime dinamis yang kuat. Jika Anda berpikir untuk mengimplementasikan bahasa pemrograman baru, terutama yang dinamis, tempat termudah untuk memulai mungkin adalah dengan interpreter.

LATIHAN SOAL

1. Tulis transpiler Brainfuck-ke-Python. Transpiler mirip dengan compiler, tetapi alih-alih mengkonversi kode sumber yang ditulis dalam bahasa tingkat tinggi menjadi kode mesin, ia mengkonversi kode sumber dari satu bahasa tingkat tinggi ke bahasa tingkat tinggi lainnya. Anda dapat menggunakan kembali banyak struktur dari interpreter Brainfuck. Alih-alih mengeksekusi setiap perintah Brainfuck, Anda dapat mengeluarkan beberapa kode Python yang setara ke dalam daftar string. Output akhir program Anda harus berupa kode Python yang setara yang disimpan ke dalam file. Bagian tersulit adalah mencari tahu apa yang harus dilakukan dengan tanda kurung.
2. Tambahkan mode debug ke interpreter yang memungkinkan Anda untuk menjalankan program Brainfuck langkah demi langkah. Setelah setiap perintah, sebuah tabel akan ditampilkan ke konsol yang berisi semua status interpreter Brainfuck, mirip dengan tabel di awal bab ini untuk menelusuri program "repeat".

3. Tulis program Brainfuck yang membaca dua angka, membandingkannya, dan menampilkan angka yang lebih besar. Tulis tes dalam Python yang memverifikasi bahwa program bekerja dengan benar dengan angka yang dihasilkan secara acak. Tip: Anda mungkin perlu memodifikasi `sys.stdin` dengan cara yang sama seperti kita memodifikasi `sys.stdout` dalam fungsi `run()` untuk pengujian.

BAB 2
MENULIS INTERPRETER DASAR

Pada Bab 1, kita telah membangun interpreter sederhana untuk Brainfuck, sebuah bahasa minimalis dan esoteris. Tetapi Brainfuck hanyalah mainan; meskipun kita dapat menyelesaikan masalah nyata di Brainfuck, kita sebenarnya tidak ingin melakukannya. Ada bahasa pemrograman lain yang tidak jauh lebih kompleks daripada Brainfuck namun "nyata" dalam arti bahwa programmer reguler benar-benar menggunakannya untuk pekerjaan sehari-hari mereka. Pada bab ini, kita akan membangun interpreter untuk salah satu bahasa tersebut, NanoBASIC, dan kita akan mempelajari lebih lanjut tentang cara kerja interpreter dalam prosesnya.

Meskipun Brainfuck hanya memiliki delapan perintah, NanoBASIC, dialek BASIC yang disederhanakan, hanya memiliki enam jenis pernyataan. Sejujurnya, setiap pernyataan tersebut memiliki lebih banyak fungsi daripada perintah Brainfuck, tetapi tetap saja tidak banyak. Ini cukup kompleks sehingga memungkinkan kita untuk mengeksplorasi beberapa aspek interpreter yang digabungkan dalam implementasi Brainfuck kita. Secara khusus, kita akan menulis tokenizer, parser, dan runtime secara terpisah, sedangkan interpreter Brainfuck kita menangani ketiga tugas tersebut sekaligus.

Kita akan menggunakan pendekatan yang dapat diskalakan untuk setiap komponen, artinya apa yang kita lakukan di sini dapat diperluas untuk bekerja dengan bahasa yang lebih besar.

2.1 MEMAHAMI NANOBASIC

Dengan sintaksnya yang sederhana dan keberadaannya yang meluas, BASIC (Beginner's All-purpose Symbolic Instruction Code) mendemokratisasi dunia komputasi dan menjadi bahasa standar de facto dari revolusi komputer pribadi. NanoBASIC adalah versi bahasa pemrograman BASIC yang berasal dari dialek populer untuk mikrokomputer tahun 1970-an yang dikenal sebagai Tiny BASIC. NanoBASIC bahkan lebih sederhana (atau lebih kecil, jika Anda mau) daripada Tiny BASIC, oleh karena itu disebut Nano.

NanoBASIC hampir sepenuhnya sama dengan Tiny BASIC, tetapi penulis telah membuat beberapa perubahan: ada beberapa pernyataan yang hilang, dan ada beberapa perbedaan kecil mengenai nama variabel dan lebar bilangan bulat. Saat kita membahas bahasa ini, Anda mungkin bertanya-tanya tentang beberapa sintaks atau keterbatasannya yang esoteris. Keunikan ini disengaja karena NanoBASIC dimaksudkan untuk sebagian besar kompatibel dengan Tiny BASIC. Di akhir bab ini, Anda akan dapat mengambil program Tiny BASIC aktual yang Anda temukan secara online dan menjalankannya di interpreter NanoBASIC kami. Oleh karena itu, Anda akan mengimplementasikan bahasa yang digunakan di dunia nyata.

Anda dapat mempelajari semua yang perlu Anda ketahui tentang NanoBASIC hanya dalam beberapa menit. Apakah Anda ingat berapa lama waktu yang Anda butuhkan untuk

mempelajari Python? Pada akhir bagian ini, Anda akan sepenuhnya mampu menulis program dalam NanoBASIC.

Sejarah BASIC

BASIC awalnya dikembangkan pada tahun 1964 oleh John Kemeny dan Thomas Kurtz di Dartmouth College untuk membuat komputer lebih mudah diakses, termasuk bagi mahasiswa yang tidak mengambil jurusan sains atau matematika. Bahkan, mahasiswa S1-lah yang membangun implementasi BASIC pertama bersama Kemeny dan Kurtz. Ketika revolusi komputer pribadi dimulai pada pertengahan tahun 1970-an, BASIC sangat cocok untuk para penghobi dan orang-orang "biasa" lainnya yang membeli mesin pertama. Akibatnya, BASIC menjadi bahasa pemrograman tingkat tinggi yang paling populer untuk komputer pribadi dari pertengahan tahun 1970-an hingga pertengahan tahun 1980-an. Komputer umum pada era tersebut, seperti Commodore 64 dan Apple II, dilengkapi dengan interpreter BASIC bawaan.

Oleh karena itu, BASIC adalah cara banyak orang berinteraksi dengan komputer pribadi awal. Sebagai contoh, BASIC adalah bahasa pemrograman pertama Linus Torvalds di Commodore VIC-20 pada tahun 1981.

Menariknya, Microsoft memulai kiprahnya pada tahun 1975 ketika Bill Gates dan Paul Allen mengembangkan interpreter BASIC untuk salah satu komputer pribadi pertama, Altair 8800. Perusahaan mereka berkembang pesat ketika mereka memindahkan interpreter mereka ke mesin lain di akhir tahun 1970-an. Microsoft BASIC dikirimkan bersama banyak komputer pribadi, dan menjadi dialek standar de facto dari BASIC. Akhirnya, Microsoft memasuki bisnis sistem operasi pada tahun 1981 dengan DOS pada IBM PC asli, tetapi BASIC adalah awal mula perusahaan tersebut. Sekarang Anda juga akan mengembangkan interpreter BASIC!

Tiny BASIC, yang NanoBASIC merupakan salah satu bentuknya, pada gilirannya dimulai karena Microsoft. Banyak orang yang terlibat dalam pengembangan awal Tiny BASIC sebagian termotivasi oleh tingginya biaya interpreter Microsoft.[4] Beberapa dari mereka juga percaya bahwa orang harus bebas untuk berbagi perangkat lunak sesuai keinginan mereka. Ini adalah bentuk awal dari Gerakan Perangkat Lunak Bebas. Selain menghindari biaya tinggi Microsoft, para pengembang juga menginginkan bahasa yang cukup kecil untuk sesuai dengan keterbatasan memori mikrokomputer pada saat itu (seringkali hanya 4 kilobyte [KB]) dan cukup portabel sehingga program dapat dijalankan di berbagai jenis mesin. Pada akhirnya, Tiny BASIC diporting ke berbagai macam komputer pribadi di berbagai arsitektur mikroprosesor dan digunakan secara luas.

Paradigma, Sintaksis, dan Semantik NanoBASIC

Seperti yang telah Anda baca, BASIC dimaksudkan agar mudah digunakan oleh orang-orang non-teknis, dan banyak BASIC dirancang untuk bekerja di lingkungan dengan keterbatasan memori. Oleh karena itu, BASIC cenderung menjadi bahasa yang disederhanakan dengan fitur yang relatif sedikit, bahkan dibandingkan dengan bahasa lain pada masanya. Dialek yang sedang kami kembangkan, NanoBASIC, bersifat imperatif, tetapi kami hampir tidak akan menyebutnya prosedural. Mari kita tinjau apa arti istilah-istilah tersebut.

Bahasa imperatif adalah bahasa di mana Anda memberikan instruksi terperinci yang memberi tahu komputer bagaimana Anda ingin komputer menyelesaikan suatu tugas. Ini

berbeda dengan bahasa deklaratif, yang berfokus pada "apa" yang ingin Anda lakukan daripada "bagaimana" Anda ingin melakukannya. Katakanlah penulis ingin Anda menggambar persegi di tengah selembar kertas grafik. Cara imperatif untuk melakukannya adalah dengan memberi tahu Anda: "Mulailah dari titik (4, 4) dan gambarlah garis ke atas sejauh lima unit. Kemudian, gambarlah garis ke kanan sejauh lima unit. Kemudian, gambarlah garis ke bawah sejauh lima unit. Kemudian, gambarlah garis ke kiri sejauh lima unit." Cara deklaratif untuk melakukannya adalah dengan mengatakan, "Gambarlah persegi 5×5 di tengah kertas." Dalam dunia deklaratif, penulis menyatakan apa yang penulis inginkan dan membiarkan Anda (atau komputer) mencari tahu detail spesifik tentang cara mencapainya.

Bahasa pemrograman imperatif modern umumnya terbagi menjadi dua sub-paradigma utama: prosedural dan berorientasi objek. Bahasa pemrograman prosedural menggunakan subrutin/prosedur/fungsi (istilah-istilah ini sering, tetapi tidak selalu, digunakan secara bergantian) sebagai titik abstraksi utama. Kode dipecah menjadi beberapa fungsi yang masing-masing memiliki tujuan spesifik dan bekerja bersama untuk membentuk keseluruhan program. Pemrograman berorientasi objek menggunakan objek sebagai titik abstraksi utama, dan karena Anda adalah programmer Python tingkat menengah atau mahir, penulis berasumsi Anda tahu apa artinya itu. Tentu saja, Python dapat diprogram dengan kedua gaya tersebut.

Sub-paradigma pemrograman deklaratif yang paling populer adalah pemrograman fungsional dan logika. Membahas detailnya berada di luar cakupan bab ini. NanoBASIC jelas bukan bahasa pemrograman deklaratif. Ia termasuk dalam kelompok imperatif. Ia juga jelas bukan berorientasi objek.

Tetapi apakah ia prosedural? Meskipun secara teknis memiliki cara untuk memanggil subrutin dengan pernyataan GOSUB-nya, yang akan kita bahas sebentar lagi, tidak ada yang menyerupai fungsi modern dalam arti memiliki parameter dan nilai kembalian. Itulah mengapa penulis menulis bahwa kita "hampir tidak akan menyebutnya prosedural."

Beberapa versi BASIC, seperti Tiny BASIC dan NanoBASIC, tidak hanya tidak memiliki konsep fungsi, tetapi juga tidak memiliki perulangan atau struktur kontrol modern lainnya. Sebaliknya, semua kontrol ditangani dengan GOTO dan GOSUB, yang memicu lompatan langsung ke nomor baris tertentu dalam program. Ini, ditambah dengan pernyataan if, adalah satu-satunya cara untuk mengontrol program Tiny BASIC atau NanoBASIC. BASIC awal terkenal karena mendorong "kode spaghetti" karena lompatan eksplisit dari satu bagian program ke bagian lain dan mekanisme pengorganisasian yang buruk. Kritik ini sepenuhnya adil. Tanpa fungsi atau objek sebagai mekanisme pengorganisasian, bahasa imperatif pasti akan berubah menjadi kode spaghetti. Jangan heran jika Anda melihat beberapa kode spaghetti saat kita mulai membuat NanoBASIC!

Komentar dan Nomor Baris

Komentar di NanoBASIC dimulai dengan penunjukan REM dan dapat diakhiri dengan string apa pun. Komentar tidak akan diproses sama sekali oleh interpreter. Setiap baris non-komentar di NanoBASIC dimulai dengan nomor baris dan diikuti oleh sebuah pernyataan. Programmer dapat memilih nomor baris sembarang, selama semuanya berurutan meningkat dari atas file sumber ke bawah. Misalnya, nomor baris berikut valid:

```
10 PRINT "Hello"
REM This is a comment
20 PRINT "Goodbye"
30 PRINT "WOW"
```

Sebaliknya, nomor baris ini tidak valid, sehingga perilaku program tidak terdefinisi:

```
10 PRINT "Hello"
REM This is a comment
40 PRINT "Goodbye"
30 PRINT "WOW"
```

Program yang memiliki pernyataan `GOTO` atau `GOSUB` dan nomor baris yang tidak berurutan tidak akan berfungsi dengan benar.

Hanya ada enam cara untuk memulai sebuah pernyataan di NanoBASIC: `PRINT`, `IF`, `GOTO`, `GOSUB`, `RETURN`, dan `LET`. Jika Anda mengetahui keenam jenis pernyataan ini, pada dasarnya Anda mengetahui seluruh bahasa tersebut. Inilah mengapa Anda dapat mempelajari NanoBASIC hanya dalam beberapa menit jika Anda sudah mengetahui bahasa pemrograman lain.

LET, Variabel, dan Ekspresi Matematika

Pernyataan `LET` mengikat nilai ke sebuah variabel. Semua variabel mewakili bilangan bulat. Tidak ada tipe variabel lain. Tiny BASIC asli terbatas hanya pada 26 nama variabel satu huruf (A hingga Z). Dalam NanoBASIC, ini diperluas untuk mencakup pengidentifikasi dengan panjang sembarang yang terdiri dari huruf dan garis bawah. Pernyataan berikut menetapkan variabel A menjadi 5:

```
10 LET A = 5
```

Kata kunci `LET` harus diikuti oleh nama variabel dan tanda sama dengan `(=).` Setelah itu, ekspresi matematika apa pun dapat muncul. Ekspresi matematika NanoBASIC dapat terdiri dari variabel; literal bilangan bulat; operator untuk penjumlahan `(+)`, pengurangan `(-)`, perkalian `(*)`, dan pembagian `(/)`; dan tanda kurung `((and)).` Selain itu, Anda dapat meniadakan nilai matematika apa pun di NanoBASIC dengan tanda negatif `(-)`. Semua perhitungan matematika berlangsung dalam ranah bilangan bulat bertanda. Mulai saat ini, kita hanya akan menyebut ekspresi matematika sebagai ekspresi. Berikut ini adalah semua penggunaan `LET` yang valid:

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

Karena keterbatasan mesin, sebagian besar implementasi Tiny BASIC terbatas pada bilangan bulat 16-bit. Variabel kita didukung oleh bilangan bulat Python di balik layar, yang memiliki presisi sembarang, sehingga tidak terbatas pada 16 bit. Hal ini, dan nama variabel dengan panjang sembarang, adalah beberapa dari sedikit area di mana NanoBASIC lebih unggul daripada Tiny BASIC, bukan hanya sekadar subsetnya.

Pernyataan PRINT

Setiap literal string atau ekspresi dapat ditampilkan ke konsol dengan PRINT. Literal string NanoBASIC adalah karakter apa pun yang terletak di antara tanda kutip ganda (""). Sayangnya, tidak ada cara di NanoBASIC untuk menyertakan tanda kutip ganda sebenarnya dalam string Anda. Dengan kata lain, tidak ada mekanisme escape. penulis akan menyerahkan itu sebagai latihan di akhir bab ini. Berikut adalah beberapa pernyataan `PRINT` yang valid dengan literal string:

```
10 PRINT "What a nice program"
20 PRINT "Who said sit down?"
30 PRINT "6734 spells HELP upside down sorta"
```

Seperti yang telah disebutkan, `PRINT` juga dapat mencetak hasil dari ekspresi apa pun:

```
REM This was the first thing Paul Allen ran on the Altair 8800
70 PRINT 2 + 2
```

Anda juga dapat memberikan daftar item yang dipisahkan koma (literal string dan ekspresi) ke `PRINT`. Semua item yang dicetak akan memiliki karakter tab di antaranya, dan `PRINT` selalu menyelesaikan pencetakan dengan menyisipkan karakter baris baru. Misalnya:

```
30 PRINT "2 plus 2 is", 2 + 2, "and 3 times 5 is", 3 * 5
```

Ini akan menghasilkan teks yang terlihat seperti berikut:

```
2 plus 2 is 4 and 3 times 5 is 15
```

Perhatikan bahwa spasi antar ekspresi disebabkan oleh karakter tab. Karena berbagai pengaturan konsol, tampilannya mungkin tidak sama di terminal Anda.

Pernyataan IF dan Ekspresi Boolean

Pernyataan `IF` NanoBASIC mirip dengan pernyataan `if` dalam bahasa lain, tetapi lebih sederhana dan ringkas. Pernyataan ini hanya dapat memiliki satu ekspresi Boolean (tidak ada operator and atau or, misalnya), dan tidak memiliki klausa else. Terakhir, pernyataan ini hanya dapat mengeksekusi satu pernyataan jika bernilai benar. Pernyataan yang akan dieksekusi jika bernilai benar selalu didahului oleh kata THEN literal. Misalnya:

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

Ekspresi Boolean sebagian besar melibatkan perbandingan yang Anda harapkan, tetapi operatornya sedikit berbeda dari operator standar gaya C. Misalnya, tidak sama dengan dapat berupa <> atau >< di NanoBASIC, dan sama dengan adalah `=`, bukan `==`.

Pernyataan GOTO, GOSUB, dan RETURN

Pernyataan GOTO langsung melompat ke nomor baris tanpa cara untuk kembali. Pernyataan GOSUB melompat ke nomor baris, tetapi pernyataan RETURN yang sesuai akan mengirim program kembali ke baris tepat setelah tempat GOSUB awalnya dipanggil. Berikut contohnya:

```
10 GOTO 50
20 LET A = 10
40 RETURN
50 LET A = 5
60 GOSUB 20
REM RETURN returns to here; we expect A to be 10
70 PRINT A
```

Program ini pada akhirnya akan menampilkan 10 ke konsol.

Gaya dan Detail NanoBASIC

Secara umum, menulis kata kunci BASIC dengan huruf kapital semua dianggap sebagai gaya yang baik. Karena NanoBASIC tidak memiliki fasilitas pengorganisasian yang baik, ada baiknya juga untuk menyertakan komentar di seluruh program Anda yang menjelaskan apa yang sedang terjadi.

Sayangnya, kode BASIC biasanya cepat dipenuhi dengan pernyataan GOTO dan menjadi "kode spaghetti". Ini adalah hal yang wajar, dan tidak banyak yang dapat Anda lakukan di NanoBASIC jika Anda ingin menulis program dengan kompleksitas sedang. Akhirnya, orang-orang yang tidak senang dengan gaya pemrograman GOTO menciptakan gerakan pemrograman terstruktur. Misalnya, Edsger Dijkstra terkenal menulis surat berjudul "Pernyataan Go To Dianggap Berbahaya."

Berikut beberapa hal lain yang perlu Anda ketahui tentang NanoBASIC:

- NanoBASIC tidak peka huruf besar/kecil. Ini berarti bahwa `LET A = 5` sama dengan `let a = 5`.
- Perilaku yang tidak dijelaskan dalam bab ini tidak terdefinisi.

Pada akhirnya, NanoBASIC sengaja dibuat sebagai barang cacat karena penulis ingin menjaga interpreter tetap sederhana dan menyediakan analog dunia nyata—Tiny BASIC—untuk membuat karya bab ini terasa lebih "nyata." Saya juga berpikir bahwa fakta bahwa NanoBASIC didasarkan pada bahasa nyata dan mampu menjalankan program Tiny BASIC asli yang ditemukan di internet membuat penulisan interpreter menjadi lebih menyenangkan. Namun,

akan cukup mudah bagi kita untuk membuat bahasa tersebut lebih kuat. Kita akan membahasnya dalam latihan.

Contoh Program NanoBASIC

Beberapa contoh program NanoBASIC disertakan dalam direktori NanoBASIC/Examples di repositori pendamping. Salah satu program tersebut mencetak semua angka dalam deret Fibonacci yang kurang dari 100. Deret Fibonacci adalah deret angka progresif di mana setiap angka (kecuali dua angka pertama yang istimewa) adalah jumlah dari dua angka sebelumnya. Deret ini dimulai dengan angka 0 dan 1. Kemudian, 0 + 1 = 1, sehingga angka berikutnya dalam deret adalah 1. Kemudian, 1 + 1 = 2, sehingga angka berikutnya adalah 2. Deret ini berlanjut 3, 5, 8, 13, dan seterusnya.

Berikut adalah fib.bas, program Fibonacci NanoBASIC:

NanoBASIC/Examples/fib.bas

```
REM Printing the Fibonacci numbers less than 100
REM A is the last number
10 LET A = 0
REM B is the next number
11 LET B = 1
20 PRINT A
21 PRINT B
REM C is last + next
30 LET C = A + B
31 LET A = B
32 LET B = C
40 IF B < 100 THEN GOTO 21
```

Sesuai dengan gaya Tiny BASIC, kita hanya menggunakan huruf kapital tunggal sebagai nama variabel. Ini menyoroti pentingnya komentar yang banyak. Seperti yang telah dibahas sebelumnya, nilai yang dipilih untuk nomor baris bersifat arbitrer selama urutannya meningkat. Pada baris 10 dan 11, kita memulai urutan dengan nilai awal yang dikodekan secara langsung, yaitu 0 dan 1.

Pada baris 30, kita membentuk angka berikutnya dalam urutan, C, dengan menjumlahkan dua angka sebelumnya. Baris 21 hingga 40 membentuk semacam perulangan melalui penggunaan `IF` dan GOTO pada baris 40. Beberapa versi Tiny BASIC yang lebih baru memiliki pernyataan perulangan sebenarnya seperti FOR, tetapi versi paling awal melakukan semua perulangan menggunakan sintaks yang mirip dengan ini, seperti cara kerja perulangan di sebagian besar bahasa assembly. Setelah Anda menyelesaikan bab ini, Anda akan dapat menjalankan program ini sendiri dengan perintah seperti berikut:

```
% python3 -m NanoBASIC NanoBASIC/Examples/fib.bas
0
1
1
2
3
```

```
5
8
13
21
34
55
89
```

Kita hampir siap untuk menulis implementasi NanoBASIC, tetapi sebelum sampai di sana, penting untuk menentukan sintaks bahasa secara lebih formal. Kita dapat langsung menggunakan spesifikasi tersebut untuk menulis implementasi kita.

2.2 MEMFORMALKAN SINTAKS NANOBASIC

Sintaks bahasa pemrograman didefinisikan secara formal oleh tata bahasa. Bentuk Backus–Naur (BNF) adalah cara umum tata bahasa bahasa pemrograman ditentukan. Ada banyak ekstensi dan penambahan BNF; kita akan menggunakan bentuk yang menurut penulis akan sangat jelas bagi programmer tingkat menengah karena mencakup beberapa sintaks seperti ekspresi reguler.

Tata bahasa terdiri dari seperangkat aturan produksi yang mendefinisikan sintaks yang diizinkan dalam bahasa pemrograman. Istilah aturan produksi terdengar mewah, tetapi itu hanyalah cara untuk mengganti satu hal dengan hal lain. Katakanlah penulis membuat tata bahasa untuk bahasa yang hanya dapat terdiri dari huruf A dan B dan angka 1 dan 2. Aturan produksinya mungkin terlihat seperti ini:

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

Sebuah pengidentifikasi yang dibungkus dalam tanda kurung sudut, seperti `<ekspresi>`, adalah non-terminal. Ini adalah item dalam tata bahasa yang, ketika diperluas, digantikan oleh sesuatu yang lain. Apa yang digantikannya ditentukan oleh sisi kanan aturan produksinya. Dalam aturan produksi, simbol ::= memisahkan non-terminal dari penggantinya. Pengganti tersebut dapat terdiri dari non-terminal atau terminal.

Terminal adalah sesuatu yang akan muncul dalam bahasa dalam bentuk literalnya. Terminal tidak diperluas lebih lanjut. Dalam sintaks kita, terminal dibungkus dalam tanda kutip tunggal, seperti 'A'. Sintaks kita juga menggunakan | untuk berarti atau. Atau berarti ada pilihan opsi yang dapat dipilih untuk bagian produksi tersebut. Kita menggunakan tanda kurung untuk pengelompokan, dan * menandakan nol atau lebih pengulangan sesuatu.

Dengan mengingat hal ini, kita dapat membaca tiga aturan produksi yang ditunjukkan sebelumnya sebagai indikasi:

1. Sebuah ekspresi terdiri dari nol atau lebih huruf atau angka.
2. Huruf adalah A atau B.

3. Angka adalah 1 atau 2.

Kita dapat menggunakan tata bahasa untuk memeriksa apakah suatu rangkaian teks tertentu merupakan sintaks yang valid untuk suatu bahasa hanya dengan mengikuti aturan produksinya. Misalnya, tata bahasa kita menetapkan bahwa "AAA21B" adalah sintaks yang valid tetapi "AB123" tidak. Lebih dari satu tata bahasa dapat menentukan bahasa yang sama. Nama non-terminal sebagian besar bersifat arbitrer dan harus dipilih agar masuk akal bagi manusia. Misalnya, tidak ada alasan mengapa huruf dan angka perlu dipisahkan dalam tata bahasa kita. Kita dapat menyederhanakan tata bahasa menjadi:

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

Kita bahkan bisa menghilangkan aturan produksi kedua sama sekali:

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

Secara umum, kita sebaiknya tidak menggunakan aturan produksi yang tidak perlu, karena hal itu akan mempersulit tata bahasa. Namun, jika non-terminal tambahan tersebut mewakili beberapa terminal yang mungkin dan akan digunakan lagi di tempat lain dalam tata bahasa, masuk akal untuk memberikannya aturan produksi tersendiri daripada menduplikasi daftar terminal yang panjang. Hal ini akan menjadi lebih jelas saat kita bekerja dengan tata bahasa yang lebih besar dengan struktur yang lebih kaya. Ini analog dengan pemrograman, di mana gaya penulisan yang lebih baik adalah memiliki banyak fungsi kecil yang kita gunakan kembali daripada hanya beberapa fungsi besar.

Mari kita lihat contoh lain. Katakanlah penulis menentukan aturan produksi untuk daftar bernomor. Mungkin akan terlihat seperti ini:

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

Kami telah memperkenalkan beberapa bentuk khusus lagi. Simbol ... menunjukkan daftar terminal berlanjut dengan cara tersirat (ini tidak terlalu formal, tetapi menghemat ruang), dan . hanya berarti terminal apa pun yang dapat dibayangkan pengguna, seperti dalam ekspresi reguler. Mari kita kembali menyajikan lima aturan tata bahasa ini dalam bentuk yang lebih mirip bahasa Inggris:

1. Sebuah daftar terdiri dari nol atau lebih item.
2. Sebuah item adalah angka diikuti oleh titik, beberapa teks, dan baris baru.
3. Sebuah angka adalah satu atau lebih digit.
4. Sebuah digit adalah salah satu karakter 0 hingga 9.

5. Sebuah teks adalah string sembarang.

Apakah Anda memperhatikan masalah dengan tata bahasa ini mengenai cara penanganannya terhadap angka? Angka dengan angka nol di depan, seperti 0020, akan diizinkan, tetapi itu tidak masuk akal dalam daftar bernomor. Bagaimana cara memperbaikinya? penulis akan menyerahkan itu sebagai latihan bagi pembaca. Jika Anda dapat memperbaikinya, Anda mungkin memiliki pemahaman yang cukup baik tentang terminal dan non-terminal.

Tata bahasa yang dijelaskan dengan BNF dikatakan bebas konteks, artinya setiap aturan produksi dapat berdiri sendiri untuk non-terminal yang muncul dalam string yang lebih besar. Tanpa konteks apa pun tentang sisa string, aturan produksi masih dapat diperluas untuk non-terminal tunggal yang dimaksud. Dengan kata lain, dalam tata bahasa bebas konteks, setiap non-terminal tidak bergantung pada non-terminal lain di sekitarnya untuk diperluas.

Tata bahasa NanoBASIC didasarkan pada tata bahasa Tiny BASIC asli yang diterbitkan oleh Dennis Allison, pencipta implementasi Tiny BASIC pertama, pada tahun 1976. Tampilannya seperti ini:

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

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

Satu-satunya sintaks baru dalam tata bahasa NanoBASIC lengkap adalah karakter epsilon (ε). Artinya, tidak mungkin ada apa pun (“kosong”) di tempat kemunculannya. Karakter ini selalu muncul sebagai bagian dari operator "atau", yang berarti mungkin ada sesuatu, atau mungkin tidak ada apa pun.

Tata bahasa NanoBASIC terlihat jauh lebih canggih daripada dua contoh sebelumnya—bagaimanapun juga, ini adalah bahasa pemrograman utuh—tetapi sebenarnya cukup mudah untuk diuraikan:

1. Sebuah baris adalah angka (nomor baris) diikuti oleh sebuah pernyataan, atau komentar (REM mendahului semua komentar) ❶.
2. Sebuah pernyataan adalah salah satu dari enam pernyataan yang telah kita pelajari (`PRINT, IF, GOTO, LET, GOSUB, RETURN`) ❷. Ini adalah tempat pertama kita melihat semacam rekursi: sebuah pernyataan IF berisi pernyataan lain dalam klausa THEN-nya, sehingga salah satu dari enam pernyataan tersebut dapat muncul setelah THEN.
3. Daftar ekspresi (`expr-list`) adalah daftar string atau ekspresi yang dipisahkan koma ❸. Seperti yang Anda lihat dari bagian `PRINT` pada aturan produksi sebelumnya, daftar ekspresi hanya digunakan untuk pernyataan `PRINT.` Ini juga satu-satunya aturan yang menghubungkan ke string. Oleh karena itu, string hanya dapat digunakan sebagai bagian dari daftar ekspresi pernyataan `PRINT`. Kita menyebutnya daftar, tetapi sebenarnya daftar ekspresi hanya dapat berisi satu ekspresi atau string. Jika ada lebih dari satu, saat itulah kita menggunakan bagian * dari tata bahasa, yang dilekatkan pada koma dan pilihan ekspresi atau string berikutnya.
4. Ekspresi adalah sesuatu yang berhubungan dengan aritmatika. Ini bisa berupa penjumlahan beberapa angka, perkalian beberapa angka, atau hanya mengambil nilai dari suatu variabel. Aturan produksi ekspresi itu sendiri hanya mencakup kemungkinan penjumlahan atau pengurangan ❹.
5. Ekspresi terdiri dari beberapa istilah. Sedangkan aturan produksi ekspresi menangani penjumlahan dan pengurangan, aturan produksi istilah menangani perkalian dan pembagian ❺. Alasannya berkaitan dengan prioritas: semakin "dalam" kita menelusuri non-terminal, semakin tinggi prioritas operator kita ketika kita akhirnya mengubah tata bahasa ini menjadi bahasa kerja. Itulah mengapa Anda menemukan perkalian dan pembagian setelah penjumlahan dan pengurangan.
6. Prioritas juga merupakan alasan mengapa Anda menemukan tanda kurung dalam aturan produksi untuk faktor ❻ dan bukan dalam aturan produksi untuk ekspresi atau istilah. Tanda kurung memiliki prioritas tertinggi dari semua operator aritmatika. Hal lain yang dapat kita gantikan dengan faktor adalah variabel (yang dalam runtime akan mengambil nilainya), literal angka, atau negasi (opsi tanda minus di depan -).
7. Aturan untuk variabel, angka, digit, huruf, operator relasional (relops), dan string sebagian besar mudah dipahami. Perhatikan betapa mudahnya memperluas apa yang diizinkan sebagai pengidentifikasi variabel: kita mengizinkan satu atau lebih garis

bawah atau huruf, berbeda dengan huruf tunggal Tiny BASIC asli. PC awal itu benar-benar terbatas memorinya jika mereka harus membatasi kita hanya pada 26 nama variabel huruf tunggal.

8. Ekspresi Boolean hanyalah dua ekspresi numerik dengan operator relasional di antaranya. ❼. Karena NanoBASIC tidak memiliki operator "dan" atau "atau", tidak perlu bentuk khusus * di sini seperti yang kita butuhkan untuk operasi aritmatika seperti penjumlahan dan perkalian.

Tata bahasa ini menyediakan cetak biru untuk mengimplementasikan tokenizer dan parser interpreter kita. Jika Anda ingat dari Bab 1 apa saja bagian-bagian tersebut, Anda mungkin sekarang dapat melihat bagaimana terminal dalam tata bahasa akan menjadi token yang dibaca oleh tokenizer kita. Dan berikut ini sesuatu yang lebih berguna: aturan produksi untuk non-terminal pada akhirnya akan dipetakan ke fungsi dalam parser penurunan rekursif kita. Kita akan kembali ke sana sebentar lagi. Pada akhirnya, tata bahasa menentukan sintaks untuk bahasa pemrograman, tetapi tidak memberikan makna pada setiap elemen bahasa tersebut. Itulah keajaiban dari interpreter kita.

2.3 IMPLEMENTASI NANOBASIC

Setelah kita membahas NanoBASIC dan bagaimana sintaksnya ditentukan, akhirnya tiba saatnya untuk mulai menulis implementasi kita. Anda mungkin ingat dari Bab 1 bahwa interpreter dasar memiliki setidaknya tiga bagian:

- Tokenizer (kadang-kadang dikenal sebagai lexer) yang mengambil kode sumber asli dan membaginya menjadi konstruksi terkecil yang dapat dikenali yang diizinkan dalam bahasa pemrograman. Ini dikenal sebagai token. Untuk kode a + 2, tokennya mungkin a, +, dan 2.
- Parser yang mengambil token yang berdekatan dan mencari tahu artinya (yaitu, ekspresi atau pernyataan yang dibentuknya). Parser biasanya menghasilkan pohon node yang mewakili hubungan relatif antara ekspresi, pernyataan, dan nilai literal. Pohon ini disebut pohon sintaks abstrak (AST). Misalnya, jika interpreter Python melihat token a diikuti oleh token + diikuti oleh token 2, ia dapat membangun node ekspresi aritmatika dan menghubungkannya ke node untuk a dan 2.
- Lingkungan runtime yang menelusuri node AST dan menjalankan operasi yang sesuai untuk mengeksekusi makna yang terkandung di dalamnya. Untuk node ekspresi aritmatika a + 2 kita, ini berarti mencari nilai yang diwakili oleh a dan menambahkan 2 ke dalamnya.

Kita akan membangun ketiga bagian ini secara berurutan, tetapi sebelum kita dapat sampai ke tokenizer, kita perlu dapat membuka file kode NanoBASIC:

NanoBASIC/__main__.py

```
from argparse import ArgumentParser
from NanoBASIC.executioner import execute
if __name__ == "__main__":
   # Parse the file argument
   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)
```

Kita memuat file kode sumber berdasarkan argumen baris perintah, hampir sama seperti yang kita lakukan di interpreter Brainfuck kita, dan meneruskan fungsi yang disebut `execute()` ke dalamnya. Fungsi tersebut berada dalam file terpisah agar lebih mudah diakses untuk pengujian kita. Fungsi ini menggabungkan komponen tokenizer, parser, dan interpreter (runtime). Output dari satu komponen diberikan sebagai input ke komponen lainnya (kode sumber ➔ tokenizer ➔ parser ➔ interpreter):

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):
   # Load the text file from the argument
   # Tokenize, parse, and execute it
   with open(file_name, "r") as text_file:
          tokens = tokenize(text_file)
          ast = Parser(tokens).parse()
          Interpreter(ast).run()
```

Setiap baris kode dalam `execute()` membawa kita dari satu bagian utama interpreter ke bagian berikutnya. Hasil dari tokenizer masuk ke parser, dan hasil dari parser masuk ke lingkungan runtime. Untuk sisa bab ini, kita akan membangun setiap komponen ini secara berurutan.

Tokenizer

Tokenizer mengambil string kode sumber (isi file teks) dan mengubahnya menjadi token. Token mewakili semua potongan individual terkecil dari sebuah program yang dapat diproses. Token yang valid di NanoBASIC berasal langsung dari terminal dalam tata bahasa NanoBASIC yang dijelaskan di bagian sebelumnya.

Kita akan menggunakan pola ekspresi reguler untuk menemukan token: kita akan mengaitkan pola ekspresi reguler dengan setiap jenis token dan kemudian mencarinya satu per satu. Kesulitan dengan pengaturan ini adalah kita perlu berhati-hati tentang urutan pencarian. Jika dua ekspresi reguler dapat cocok dengan token yang sama, maka urutannya akan penting. Misalnya, dalam tokenizer kita, ekspresi reguler untuk nama variabel juga dapat cocok dengan token `PRINT` (atau nama pernyataan lainnya), jadi pencarian token nama variabel sengaja dilakukan terakhir.

Kita akan memulai tokenizer kita dengan mendefinisikan semua jenis token yang berbeda sebagai kasus enum. Setiap kasus akan dilampirkan ke ekspresi reguler untuk

menemukannya. Beberapa token juga akan memiliki nilai yang ditentukan pengguna yang terkait dengannya, yang ditunjukkan oleh True atau False di akhir setiap kasus enum. Misalnya, token variabel akan memiliki nama variabel sebenarnya yang terhubung dengannya sebagai nilai terkait. Berikut tampilan enum TokenType kita:

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
```

Enum TokenType hanya mendeskripsikan jenis token. Selain jenis token, kita juga ingin mengetahui di mana token tersebut muncul dalam file kode sumber. Ini akan berguna agar kita

dapat menentukan lokasi kesalahan sintaks dan melaporkan informasi tersebut kembali kepada programmer. Kita akan mencakup semua informasi ini menggunakan Token, tipe komposit yang menggabungkan jenis token, lokasi, dan nilai terkait jika ada:

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

Tipe properti `associated_value, str | int | None`, menggunakan sintaks petunjuk tipe yang disempurnakan yang diperkenalkan dengan Python 3.10 melalui PEP 604. penulis telah menyebutkannya secara singkat di Bab 1, dan di sini kita akan membahasnya sedikit lebih formal. Ini adalah cara untuk membuat tipe union. Variabel yang dideklarasikan sebagai tipe union dapat merujuk ke nilai yang merupakan salah satu tipe yang membentuk union tersebut. Dalam versi Python sebelum 3.10, Anda perlu mengimpor Union dari typing dan petunjuk tipenya akan terlihat seperti Union[str, int, None]. Sintaks baru ini jelas jauh lebih ringkas. Singkatnya, ini berarti bahwa associated_value dapat berupa string, integer, atau None.

Kita siap untuk membaca file kode sumber dan memecahnya menjadi token-token penyusunnya menggunakan fungsi `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
```

Fungsi ini menerima objek `TextIO`, sebuah tipe yang mewakili objek yang dapat bertindak sebagai aliran teks. Kita menginisialisasi daftar token, di mana kita akan mengumpulkan semua token di seluruh file. Kemudian, kita mengulangi setiap baris file. Karena kita ingin melaporkan nomor baris dan kolom kepada pengguna seperti yang mungkin mereka harapkan muncul di editor teks mereka, kita membuat keduanya dimulai dari 1. Selanjutnya, kita mengekstrak semua token:

```
while len(line) > 0:
    found: re.Match | None = None
    for possibility in TokenType:
        # Try each pattern from the beginning, case-insensitive
        # If it's found, store the match in *found*
      ❶ found = re.match(possibility.pattern, line, re.IGNORECASE)
        if found:
            col_end: int = col_start + found.end() - 1
            # Store tokens other than comments and whitespace
```

```
        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:
                    # Remove quote characters
                    associated_value = found.group(0)[1:-1]
                ❸ tokens.append(Token(possibility, line_num, col_start,
                        col_end, associated_value))
            # Continue search from place in line after token
            line = line[found.end():]
            col_start = col_end + 1
            break # go around again for next token
        # If we went through all the tokens and none of them were a match
        # then this must be an invalid token
    ❹ if not found:
            print(f"Syntax error on line {line_num} column {col_start}")
            break
❺ return tokens
```

Kami memindai setiap baris file dari kiri ke kanan, mencari kecocokan setiap pola token yang mungkin secara berurutan ❶. Ketika kami menemukan kecocokan yang bukan spasi atau komentar (itu diabaikan) ❷, kami memeriksa apakah itu tipe token dengan nilai terkait. Jika ya, kami menyimpan nilai terkait tersebut.

Kami membuat Token yang berisi TokenType yang cocok, di mana ia ditemukan, dan nilai terkait apa pun, dan kami menambahkannya ke koleksi token kami ❸. Ini benar-benar sesederhana melakukan proses linier tersebut. Jika kami menemukan sepotong teks yang tidak cocok dengan TokenType yang dikenal untuk NanoBASIC, itu adalah kesalahan sintaks dan kami memberi tahu pengguna ❹. Akhirnya, token dikembalikan ❺.

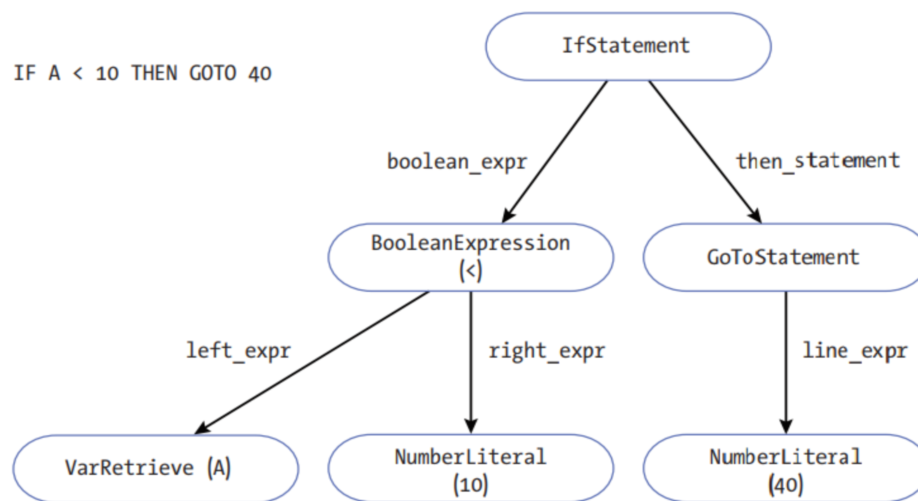
Tokenizer adalah bagian paling sederhana dari interpreter kami. Ia bertanggung jawab untuk mengubah file kode sumber asli menjadi kumpulan token yang valid dari bahasa tersebut. Token-token tersebut selanjutnya diteruskan ke parser. Namun sebelum kita melihat parser, mari kita lihat blok bangunan yang akan dihasilkan parser untuk runtime interpreter: node.

Node

Parser kita pada akhirnya akan menghasilkan AST yang berisi node yang mewakili setiap bagian penting dari program. Misalnya, setiap pernyataan IF akan menjadi node, dan setiap kali nilai variabel diambil, itu juga akan menjadi node. Karena ini adalah pohon, AST menghubungkan semua node bersama-sama menjadi hierarki hubungan. Untuk mengilustrasikan konsep ini, mari kita lihat cabang potensial nyata (menggunakan nama node

sebenarnya) dari AST dari interpreter kita. Cabang ini akan mewakili pernyataan IF `IF A < 10 THEN GOTO 40`.

Node akar cabang akan berupa IfStatement. Node IfStatement tersebut akan dihubungkan ke node `BooleanExpression (A < 10)` dan node `GoToStatement (GOTO 40)`. Node BooleanExpression akan memiliki variabel internal untuk mewakili TokenType dari operatornya `(<)`, tautan ke node VarRetrieve `(A)`, dan tautan ke node NumberLiteral (10). Node GoToStatement akan ditautkan ke satu node NumberLiteral (40). Gambar 2.1 mengilustrasikan struktur ini. Perhatikan bahwa label pada panah mewakili nama sebenarnya dari tautan antar node dalam kode.



Gambar 2.1: Node untuk `IF A < 10 THEN GOTO 40`

Tugas parser kita adalah mengubah kumpulan token yang berdekatan secara bermakna menjadi node AST. Pada fase akhir interpreter kita, node AST akan ditelusuri, yang melibatkan penyelesaian tindakan apa pun yang terhubung dengan setiap node, secara berurutan. Setiap node yang dapat muncul dalam AST kita memiliki kelasnya sendiri di nodes.py. Semua node mewarisi dari kelas Node. Setiap Node melacak lokasinya dalam file kode sumber asli untuk tujuan debugging:

NanoBASIC/ nodes.py

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

# For debug purposes, we'll need to know the locations of all Nodes
@dataclass(frozen=True)
class Node:
    line_num: int
    col_start: int
    col_end: int
```

Sekarang mari kita definisikan node Statement:

```
# All statements in NanoBASIC have a line number identifier
# that the programmer puts in before the statement (*line_id*).
# This is a little confusing because there's also the "physical"
# line number (*line_num*), that actual count of how many lines down
# in the file where the statement occurs.
@dataclass(frozen=True)
class Statement(Node):
  line_id: int
```

Setiap pernyataan dalam NanoBASIC muncul setelah nomor baris yang ditentukan pengguna. Ini untuk panggilan `GOTO` dan `GOSUB`. Kita tidak boleh mengacaukan nomor baris tersebut dengan `line_num` setiap objek Node, tempat Node muncul dalam file kode sumber. Untuk kejelasan, kita menyebut nomor baris yang ditentukan pengguna sebagai `line_id` dalam kelas Statement. Misalnya, jika baris pertama file kode sumber penulis adalah 23 `PRINT "HELLO"`, maka `line_id` adalah 23, tetapi `line_num` adalah 1.

`NumericExpression` adalah tipe Node yang dapat menghasilkan satu bilangan bulat ketika dievaluasi. Ini bisa berupa operasi biner, operasi unary, literal angka, atau pencarian variabel, jadi kita akan mendeklarasikan semua node tersebut sebagai subkelas dari `NumericExpression`:

```
# A numeric expression is something that can be computed into a number.
# This is a superclass of literals, variables & simple arithmetic operations.
@dataclass(frozen=True)
class NumericExpression(Node):
    pass

# A numeric expression with two operands like 2 + 2 or 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}"

# A numeric expression with one operand, like -4
@dataclass(frozen=True)
class UnaryOperation(NumericExpression):
  operator: TokenType
  expr: NumericExpression

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

```
# An integer written out in NanoBASIC code
@dataclass(frozen=True)
class NumberLiteral(NumericExpression):
  number: int

# A variable *name* that will have its value retrieved
@dataclass(frozen=True)
class VarRetrieve(NumericExpression):
  name: str
```

Agar dapat dievaluasi, berbagai jenis ekspresi numerik ini perlu menyimpan beberapa informasi. Misalnya, `VarRetrieve` perlu memiliki nama variabel yang nilainya sedang dicari. Demikian pula, `BinaryOperation`, yang juga dapat dianggap sebagai operasi aritmatika, perlu menyimpan operasi aritmatika aktual yang sedang dilakukan (penjumlahan, pengurangan, perkalian, atau pembagian), jadi kita menyimpan token operator bersamanya.

Sementara `NumericExpression` menghasilkan bilangan bulat, `BooleanExpression` digunakan untuk menghasilkan nilai Boolean. Ia mengambil dua node `NumericExpression` dan membandingkannya menggunakan operator Boolean (disimpan sebagai token):

```
# A Boolean expression can be computed to a true or false value.
# It takes two numeric expressions, *left_expr* and *right_expr*, and compares
# them using a Boolean *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}"
```

Node-node lainnya digunakan untuk merepresentasikan enam jenis pernyataan NanoBASIC:

```
# Represents a LET statement, setting *name* to *expr*
@dataclass(frozen=True)
class LetStatement(Statement):
  name: str
  expr: NumericExpression

# Represents a GOTO statement, transferring control to *line_expr*
@dataclass(frozen=True)
class GoToStatement(Statement):
  line_expr: NumericExpression
```

```
# Represents a GOSUB statement, transferring control to *line_expr*
# Return line_id is not saved here, it will be maintained by a stack
@dataclass(frozen=True)
class GoSubStatement(Statement):
  line_expr: NumericExpression

# Represents a RETURN statement, transferring control to the line after
# the last GOSUB statement
@dataclass(frozen=True)
class ReturnStatement(Statement):
    pass

# A PRINT statement with all that it is meant to print (comma separated)
@dataclass(frozen=True)
class PrintStatement(Statement):
  printables: list[str | NumericExpression]

# An IF statement
# *then_statement* is what statement will be executed if the
# *boolean_expression* is true
@dataclass(frozen=True)
class IfStatement(Statement):
  boolean_expr: BooleanExpression
  then_statement: Statement
```

Properti node-node ini mencerminkan bagian-bagian data yang dibutuhkan oleh setiap jenis pernyataan. Misalnya, karena pernyataan `LET` menetapkan nilai ke sebuah variabel, node `LetStatement` membutuhkan nama variabel string dan `NumericExpression` yang mewakili nilai tersebut.

Kesalahan

Mari kita sedikit menyimpang untuk membahas penanganan kesalahan dalam interpreter kita. Tidak ada yang lebih menjengkelkan saat Anda memprogram selain pesan kesalahan yang buruk. Ketika Anda membuat kesalahan dalam kode Anda, Anda ingin tahu apa yang terjadi dan di mana. Sebagai pencipta bahasa pemrograman, Anda memiliki tanggung jawab untuk memberikan pesan kesalahan yang baik kepada pengguna Anda (pemrogram NanoBASIC).

NanoBASIC akan melaporkan dua jenis kesalahan umum: kesalahan parser dan kesalahan interpreter. Kesalahan parser dapat dianggap sebagai kesalahan sintaksis, seperti ketika token berada dalam urutan yang salah. Misalnya, perlu ada ekspresi numerik (mewakili nomor baris) setelah `GOTO`, tetapi tidak setelah pernyataan `IF`. Kesalahan interpreter adalah kesalahan semantik. Kesalahan terjadi ketika program mencoba melakukan sesuatu yang tidak masuk akal, seperti mencoba menggunakan variabel sebelum diinisialisasi. Kita akan mendefinisikan kelas kesalahan untuk kedua jenis kesalahan tersebut:

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)
```

Baik `ParserError` maupun `InterpreterError` adalah subkelas dari `NanoBASICError`, yang pada gilirannya merupakan subkelas dari Exception, sebuah kelas bawaan Python yang dapat Anda timpa untuk membuat pengecualian khusus yang akan dilemparkan dalam program Anda. Kelas-kelas tersebut melaporkan pesan yang terkait dengan kesalahan dan di mana kesalahan itu terjadi dalam program asli. Misalnya, anggaplah kita memiliki program berikut:

```
10 PRINT(A)
```

Hal ini akan menyebabkan kesalahan berikut dilaporkan:

```
NanoBASIC.errors.InterpreterError: Var A used before initialized. Occurred at
line 1 and column 10
```

Kesalahan ini terjadi karena variabel A tidak pernah diinisialisasi menggunakan pernyataan `LET`. Anda akan melihat banyak kesalahan `ParserError` dan `InterpreterError` di bagian selanjutnya.

Parser

Parser mengambil token dari tokenizer dan mencoba mengubahnya menjadi struktur yang bermakna untuk menginterpretasikan program. Parsing adalah bidang ilmu komputer yang banyak dipelajari, dan ada banyak algoritma parsing yang berbeda. Bahkan ada program yang akan menghasilkan parser untuk Anda. Tidak mengherankan, program tersebut dikenal

sebagai generator parser. Generator parser dapat mengambil tata bahasa dalam bentuk BNF dan menghasilkan parser.

Kita tentu bisa menggunakan generator parser di sini, tetapi itu tidak akan seefektif menulis parser sendiri. Dan meskipun ada banyak algoritma parsing, ternyata salah satu yang paling sederhana juga merupakan salah satu yang paling efektif, dapat disesuaikan, dan banyak digunakan. Ini dikenal sebagai penurunan rekursif, dan mendasari parser C/C++ di dua kompiler paling populer di dunia, GCC dan Clang. Ini juga merupakan teknik yang digunakan dalam versi asli Tiny BASIC oleh Dennis Allison.

Dalam penurunan rekursif, umumnya setiap non-terminal yang didefinisikan dalam tata bahasa menjadi sebuah fungsi. Fungsi tersebut bertanggung jawab untuk memeriksa bahwa urutan token yang dianalisisnya mengikuti aturan produksi yang ditentukan dalam tata bahasa. Parser memeriksa token dengan melihatnya secara berurutan. Jika token yang dianalisis diharapkan menjadi bagian dari aturan produksi lain, parser penurunan rekursif hanya memanggil fungsi yang mewakili aturan produksi lain tersebut. Fungsi penurunan rekursif mengembalikan node masing-masing ketika berhasil—berhasil berarti fungsi tersebut memang menemukan token yang diharapkan.

Penurunan rekursif adalah teknik parsing top-down, artinya parsing dimulai dari "awal" tata bahasa (`<line>` dalam kasus kita) dan "menurun" hingga mencapai titik paling spesifik yang diperlukan. Itulah bagian penurunannya, tetapi bagian rekursifnya membutuhkan sedikit visualisasi lebih lanjut. Bayangkan kita sedang mengurai pernyataan `IF`. Pernyataan `IF` adalah jenis pernyataan, dan setiap non-terminal, termasuk "pernyataan" dan "pernyataan `IF`," dapat memperoleh fungsi yang sesuai dalam pengurai penurunan rekursif kita.

Pernyataan `IF` memiliki klausa `THEN` yang juga merupakan pernyataan. Oleh karena itu, ketika kita mengurai pernyataan `IF`, kita mungkin kembali memanggil fungsi kita untuk mengurai pernyataan untuk klausa `THEN`—fungsi yang sama untuk mengurai pernyataan yang memanggil fungsi kita untuk mengurai pernyataan `IF`! Ini adalah semacam rekursi. Kita akhirnya memanggil fungsi yang memanggil fungsi yang sedang kita gunakan saat ini.

Baik penurunan maupun rekursi akan menjadi lebih jelas saat kita mempelajari kode untuk kelas 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]
```

Kelas Parser menerima kumpulan token dari tokenizer. Saat proses parsing berlangsung, `token_index` internal melacak token mana yang sedang kita gunakan. Kita juga mendefinisikan beberapa properti praktis untuk mengambil token saat ini atau sebelumnya.

Metode pembantu `consume()` memeriksa apakah token saat ini adalah token yang diharapkan, menambah `token_index`, dan mengembalikan token yang diperiksa. Jika token bukan token yang diharapkan, kita akan melempar `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)
```

Fungsi pembantu seperti `consume()`, terkadang disebut `eat()` atau `accept()`, umum digunakan dalam parser karena memeriksa apakah sebuah token adalah token yang diharapkan dan melanjutkan jika ya, adalah pola yang sangat umum. Jika kita tidak memiliki `consume()`, Anda akan melihat banyak kode duplikat yang tidak perlu.

Tujuan parser kita adalah untuk menghasilkan AST yang dapat dilalui oleh runtime untuk mengeksekusi program NanoBASIC. Akar dari AST akan berupa daftar pernyataan. Cara lain untuk memikirkannya adalah bahwa program NanoBASIC hanyalah daftar pernyataan, yang ditulis secara berurutan, dari atas ke bawah file kode sumber. Pada akhirnya, runtime kita akan mengeksekusi pernyataan-pernyataan ini satu per satu. Oleh karena itu, parser penurunan rekursif kita dimulai di `parse()`, yang akan "menurun" melalui metode parser lainnya dan akhirnya mengembalikan daftar pernyataan:

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

Setiap pernyataan harus ditulis pada barisnya sendiri, di samping pengenal baris, jadi langkah pertama dalam proses penurunan adalah mengurai sebuah baris:

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

Kita mengharapkan pengidentifikasi baris berada di awal baris. Oleh karena itu, parse_line() dimulai dengan mencoba mengonsumsi token NUMBER. Jika berhasil, kita melanjutkan parsing pernyataan itu sendiri. Penggunaan cast() di sini adalah untuk pengecekan tipe. Jika Anda ingat dari kode tokenizer (kembali dan lihat, jika bermanfaat), associated_value sebuah token dapat berupa bilangan bulat, string, atau None.

Kita tahu bahwa NUMBER hanya akan memiliki bilangan bulat sebagai associated_value-nya, jadi aman untuk melakukan casting ke int. Pemeriksa tipe seperti mypy atau Pyright dapat menggunakan casting ini. Perhatikan bagaimana metode parse_line() sesuai dengan non-terminal <line> dalam tata bahasa. Mulai dari titik ini, banyak metode kita akan menjadi analog langsung dari non-terminal dalam tata bahasa atau aturan produksinya masing-masing (kembali dan lihat tata bahasa sebagai panduan). Misalnya, metode kita selanjutnya, parse_statement(), sesuai dengan non-terminal <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)
```

Metode ini bertanggung jawab untuk mencari tahu pernyataan mana dari enam pernyataan dalam NanoBASIC yang muncul dalam beberapa token berikutnya. Untungnya, setiap pernyataan dalam NanoBASIC dapat diidentifikasi oleh token awal tunggalnya (PRINT, IF, LET, GOTO, GOSUB, atau RETURN), jadi kita hanya perlu mencocokkan token saat ini dengan enam kemungkinan tersebut.

Untuk tujuan pengorganisasian, penulis telah memecah setiap jenis pernyataan ke dalam metodenya sendiri, meskipun ini tidak secara langsung sesuai dengan non-terminal.

Sebaliknya, Anda dapat menganggap setiap aturan produksi dari <pernyataan> sebagai penerima metodenya sendiri. Kita mulai dengan pernyataan PRINT, yang merupakan salah satu pernyataan yang lebih sulit untuk diuraikan karena dapat memiliki beberapa tipe yang dipisahkan koma yang berbeda dalam <expr-list>-nya.

```
# PRINT "COMMA",SEPARATED,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: # keep finding things to print
    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("Only strings and numeric expressions"
                         "allowed in print list.", self.current)
    # Comma means there's more to print
    if not self.out_of_tokens and self.current.kind is 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)
```

Kita menyimpan item yang akan dicetak dalam daftar Python printables. Untuk mengumpulkannya, kita terus maju (menggunakan perulangan), token demi token, memeriksa string ❶ atau ekspresi numerik ❷. Selama kita menemukan salah satu dari ini, diikuti oleh koma ❹, kita terus melakukan perulangan. Jika kita menemukan sesuatu yang bukan string atau ekspresi numerik ❸, kita melempar ParserError. Saat kita melakukan perulangan, kita juga melacak akhir kolom item terakhir untuk tujuan debugging. Node PrintStatement terakhir yang dikembalikan perlu mengetahui di mana ia dimulai dan di mana ia berakhir; ia dimulai di tempat token PRINT dimulai dan berakhir di akhir kolom terakhir dari item terakhir dalam daftar ekspresi.

Selanjutnya, mari kita lihat parse_if(), yang mencakup contoh bagus dari aspek rekursif dari penurunan rekursif:

```
# 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)
```

Seperti yang telah kita bahas, klausa THEN dari pernyataan IF adalah pernyataan lain. Untuk mengurai klausa THEN, kita memanggil `parse_statement()`, metode yang sama yang di atas dalam rantai panggilan membawa kita ke `parse_if()` sejak awal. Namun, pertama-tama, kita mengurai ekspresi Boolean di awal pernyataan IF. Kita akan melihat bagaimana cara melakukannya sebentar lagi.

Dalam semua metode penguraian yang telah dibahas sejauh ini, perhatikan bagaimana kita memanggil metode penguraian lain dan menganggapnya berfungsi. Terserah metode lain untuk melakukan penanganan kesalahan mereka sendiri dan terus-menerus memindahkan `token_index`, biasanya dengan memanggil `consume()`. Pola itu berlanjut dalam metode untuk empat jenis pernyataan lainnya:

```
# 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)
```

Keempat metode parse ini cukup mirip satu sama lain. Dalam setiap kasus, kita mengharapkan token awal tertentu (misalnya `LET` atau `GOTO`), dan kemudian kita memparsing beberapa informasi yang kita butuhkan untuk membuat node untuk jenis pernyataan tersebut. Misalnya, kita membutuhkan variabel dan ekspresi numerik untuk pernyataan `LET`, dan kita hanya membutuhkan ekspresi numerik (baris mana yang akan dituju) untuk pernyataan `GOTO`. Pernyataan paling sederhana untuk diparse adalah `RETURN` karena tidak ada yang datang setelah `RETURN`. Seperti yang dijanjikan, berikut metode `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"Expected boolean operator but found "
          f"{self.current.kind}.", self.current)
```

Ekspresi Boolean harus berisi dua ekspresi numerik dan salah satu token operator yang diizinkan di antara keduanya. Kita menyebut ekspresi numerik sebelum token sebagai kiri dan ekspresi numerik setelah token sebagai kanan. Token operator disimpan dalam node `BooleanExpression` sehingga kita dapat melakukan perbandingan yang sesuai saat runtime.

Penguraian ekspresi numerik mengikuti hierarki non-terminal dalam tata bahasa, dari `<expression>` ke `<term>` ke `<factor>`, dengan metode untuk masing-masing. Sebuah `<factor>` dapat mencakup `<var>` atau `<number>`, tetapi ini ditangani langsung di `parse_factor()` karena informasi yang dibutuhkan sudah terkandung dalam token masing-masing:

```
def parse_numeric_expression(self) -> NumericExpression:
  left = self.parse_term()
  # Keep parsing +s and -s until there are no more
  while True:
```

```
            if self.out_of_tokens: # what if expression is end of file?
                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 # no more, must be end of expression
        return left

    def parse_term(self) -> NumericExpression:
        left = self.parse_factor()
        # Keep parsing *s and /s until there are no more
        while True:
            if self.out_of_tokens: # what if expression is end of file?
                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 # no more, must be end of expression
        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("Expected matching close parenthesis.",
                    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("Unexpected token in numeric expression.",
                self.current)
```

Perhatikan urutan prioritas di sini. Dalam aritmatika, kita mengharapkan pembagian memiliki prioritas lebih tinggi daripada pengurangan, dan tanda kurung memiliki prioritas lebih tinggi daripada apa pun. Seperti yang penulis singgung ketika kita pertama kali membahas tata bahasa NanoBASIC, ini dapat dimodelkan dalam penurunan rekursif dengan urutan di mana non-terminal diuraikan. Semakin jauh Anda turun, semakin tinggi prioritasnya. Dalam hal ini, apa pun yang ditangani dalam `parse_term()` akan memiliki prioritas lebih tinggi daripada apa pun dalam `parse_numeric_expression()`, dan apa pun dalam `parse_factor()` akan memiliki prioritas lebih tinggi daripada apa pun dalam `parse_numeric_expression()` atau `parse_term()`. Ini juga alasan mengapa - dan + muncul dalam aturan produksi yang sama sementara / dan * muncul "lebih dalam."

Setiap kali kita membutuhkan sisi kiri atau kanan dari sebuah ekspresi dalam `parse_numeric_expression()` atau `parse_term()`, kita turun. Misalnya, `parse_numeric_expression()` tidak pernah memanggil `parse_numeric_expression()`. Sebaliknya, ia memanggil `parse_term()`. Ini mungkin tampak berlawanan dengan intuisi, karena Anda mungkin bertanya-tanya bagaimana beberapa penambahan berturut-turut ditangani. Kuncinya adalah bahwa `parse_numeric_expression()` dan `parse_term()` keduanya menggunakan perulangan, seperti yang kita miliki di `parse_print()` untuk mengakomodasi sejumlah aritmatika yang sembarangan (seperti banyak operasi penjumlahan). Salah satu cara untuk memikirkannya adalah bahwa kita turun dan menangani apa pun yang memiliki prioritas lebih tinggi, dan kemudian kembali ke atas untuk melanjutkan perulangan di `parse_numeric_expression()` jika ada token penambahan atau pengurangan lagi.

Mari kita coba mengerjakan sebuah contoh. Misalkan kita sedang menguraikan ekspresi `2 + 3 * 4 + 5`. Inisialisasi left dalam `parse_numeric_expression()` memanggil `parse_term()`, yang memanggil `parse_factor()`, yang mengembalikan NumberLiteral untuk 2.

Kemudian, kita akhirnya mengembalikan 2 kembali ke `parse_numeric_expression()`, yang menyimpannya di left. Selanjutnya, token + ditemui dan `parse_term()` dipanggil. Ia menguraikan 3 * 4 menggunakan beberapa panggilan ke `parse_factor()`. Kita akhirnya kembali ke `parse_numeric_expression()` dengan node `BinaryOperation` untuk `3 * 4` yang disebut sebagai right. Kemudian, left dan right digabungkan menjadi BinaryOperation baru dan dikaitkan dengan left (ikatan baru untuknya). Akhirnya, token + terakhir ditemui, dan 5 diuraikan dengan cara yang hampir sama seperti 2 (menurun hingga ke `parse_factor()`) dan dikaitkan dengan right. Sekali lagi, kiri dan kanan digabungkan menjadi kiri untuk nilai kembalian akhir dari `parse_numeric_expression()`.

Cobalah mengerjakan beberapa contoh aritmatika Anda sendiri untuk lebih memahami bagaimana operator pada tingkat penurunan yang lebih dalam memiliki prioritas yang lebih tinggi. Anda mungkin ingin membuka kode parser saat Anda bekerja. Kombinasi loop dan penggunaan kembali variabel untuk node yang berbeda dapat sulit dipahami, tetapi setelah Anda mengerjakan beberapa contoh, itu mulai masuk akal. Anda juga dapat mencoba menambahkan beberapa panggilan `print()` ke berbagai metode untuk mengilustrasikan perkembangan parsing. Anda dapat menghilangkan panggilan untuk menjalankan AST melalui runtime di `executioner.py` jika Anda belum selesai memasukkan seluruh program.

Ada cara yang lebih efisien untuk memparsing ekspresi aritmatika. Salah satu metode populer yang ditemukan oleh Dijkstra disebut algoritma shunting yard. Algoritma yang lebih efisien seperti shunting yard terkadang dikombinasikan dengan parser penurunan rekursif untuk bagian ekspresi aritmatika dalam semacam model hibrida.

Lingkungan Eksekusi

Hasil akhir dari parser kita akan berupa kumpulan node Statement AST yang disimpan sebagai daftar yang dapat ditelusuri dan dieksekusi oleh lingkungan eksekusi. penulis menamai kelas yang menelusuri AST sebagai Interpreter, meskipun penulis menyadari ini bisa sedikit membingungkan karena seluruh bab ini membahas tentang membangun interpreter. Ya, tokenizer dan parser adalah bagian dari interpreter secara keseluruhan, tetapi kelas Interpreter inilah tempat bahasa tersebut benar-benar diinterpretasikan dalam arti bahwa token yang menjadi node diubah menjadi sesuatu yang bermakna—sebuah program yang dieksekusi dengan beberapa output. Terlepas dari apakah itu nama yang baik atau tidak, kelas `Interpreter` menyediakan lingkungan eksekusi dan pemahaman tentang cara memodifikasi lingkungan atau memberikan output berdasarkan node statement dan expression yang ditemuinya.

Kelas Interpreter dimulai mirip dengan 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]
```

Alih-alih daftar token seperti yang kita miliki di kelas Parser, Interpreter menerima daftar pernyataan. Ada juga properti `current` untuk kenyamanan, untuk mengakses pernyataan saat ini. Lingkungan runtime terdiri dari pernyataan, `statement_index`, `variable_table` untuk melacak nilai setiap variabel, dan `subroutine_stack` yang akan membantu kita sampai ke tempat yang tepat setelah pasangan `GOSUB` dan `RETURN`. Selanjutnya, kita membutuhkan cara untuk menghubungkan pengidentifikasi baris ke indeks pernyataan. Pertimbangkan program NanoBASIC berikut:

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

Ketika `GOTO 50` dieksekusi, interpreter perlu menemukan pernyataan yang terkait dengan pengidentifikasi baris 50 dan melanjutkan eksekusi dari sana. Dalam NanoBASIC, programmer dapat secara sembarangan memilih pengidentifikasi baris apa pun untuk baris mana pun, selama semua pengidentifikasi baris adalah bilangan bulat dalam urutan menaik, jadi bagaimana kita dapat menemukan 50? Kita perlu mencarinya dalam daftar pernyataan. Karena baris-baris tersebut harus berurutan, metode `find_line_index()` kita dapat melakukan pencarian biner:

```
# Returns the index of a *line_id* using a binary search,
# or None if not found; assumes the statements list is sorted
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
```

Selanjutnya, metode run() mengeksekusi pernyataan-pernyataan secara berurutan:

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

Perhatikan bahwa kita menggunakan perulangan while yang dikendalikan oleh statement_index, bukan perulangan for...in. Ini karena kita mungkin memang melompat-lompat dan melewatkan atau mengulang beberapa pernyataan karena GOTO dan GOSUB. Dengan kata lain, saat kita menginterpretasikan berbagai pernyataan di dalam perulangan, statement_index dapat dimodifikasi.

Metode interpret() adalah inti dari interpreter. Metode ini menginterpretasikan node Statement dan memodifikasi lingkungan runtime atau membuat beberapa output tergantung pada arti dari setiap pernyataan tertentu:

```
    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)   #
              setup for 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)
```

Menelusuri AST ternyata jauh lebih mudah daripada membangunnya, karena kita dapat memanfaatkan pencocokan pola struktural yang disediakan oleh pernyataan `match` Python. Dalam setiap kasus (kecuali `ReturnStatement`, yang tidak memiliki properti), kita menangkap beberapa properti dari subkelas `Statement` yang kita cocokkan. Misalnya, baris case `LetStatement(name=name, expr=expr):` mengatakan bahwa, dengan asumsi statement adalah `LetStatement`, `statement.name` akan disimpan dalam variabel name lokal dan `statement.expr` akan disimpan dalam variabel expr lokal.

Tiga kasus menjaga interpreter tetap berjalan dengan menaikkan `statement_index` setelah mereka melakukan tugasnya. Kasus `GoToStatement`, `GoSubStatement`, dan `ReturnStatement` tidak, karena mereka melompat-lompat di sekitar kode dengan memodifikasi `statement_index` secara langsung. Setiap kali `GoSubStatement` ditemui, kita perlu tahu ke mana harus kembali ketika `ReturnStatement` dieksekusi berikutnya—semacam bookmark, jika Anda mau. Inilah tujuan dari `subroutine_stack`. Dalam kasus `GoSubStatement`, kita menyimpan `statement_index + 1` di tumpukan untuk menghindari loop tak terbatas (kembali ke sumber GOSUB); kemudian, dalam kasus `ReturnStatement`, kita mengeluarkan elemen dari tumpukan.

Perhatikan bagaimana, jika `boolean_expr` dari node `IfStatement` bernilai True, `interpret()` dipanggil secara rekursif dan `statement_index` tidak bertambah. Ini karena pernyataan yang terkait dengan klausa THEN itu sendiri akan memodifikasi `statement_index`. Mengevaluasi ekspresi numerik sebagian besar hanya masalah mengeksekusi operator Python yang tepat agar sesuai dengan token operator aritmatika NanoBASIC, atau mengambil variabel dari `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)
```

Perhatikan semua panggilan rekursif di `evaluate_numeric()`. Saat Anda pertama kali belajar pemrograman dalam bahasa imperatif seperti Python, rekursi mungkin tampak seperti topik yang esoteris. Namun, saat Anda kemudian menjadi programmer tingkat menengah atau mahir, Anda mulai melihat kegunaannya. Proyek ini adalah ilustrasi yang baik. Kita telah melihat di kelas `Parser` dan `Interpreter` betapa bermanfaatnya rekursi untuk mengekspresikan diri kita secara algoritmik.

Percaya atau tidak, ada seluruh bahasa pemrograman (kebanyakan dalam paradigma fungsional) yang tidak memiliki loop, hanya rekursi. Itu mungkin terdengar ekstrem, dan hanya menggunakan rekursi tentu akan menjadi cara yang buruk untuk memprogram Python, karena akan membuat kode Anda jauh kurang mudah dibaca oleh programmer Python lain dan menimbulkan beberapa penalti kinerja.

Tetapi ini menggarisbawahi betapa kuatnya teknik rekursi. Apa pun yang dapat Anda lakukan dengan loop, Anda juga dapat melakukannya secara rekursif, tetapi keajaibannya adalah ketika rekursi benar-benar membantu Anda mengekspresikan diri dengan lebih baik, seperti yang terjadi dalam proyek ini. Secara khusus, rekursi dapat sangat membantu ketika bekerja dengan struktur data hierarkis seperti AST.

Mengevaluasi ekspresi Boolean hampir sama dengan mengevaluasi ekspresi numerik. Ini adalah konversi dari operator NanoBASIC ke operator 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)
```

Dan itu saja! Menjalankan program NanoBASIC jauh lebih mudah daripada menguraikannya.

Jika ini adalah kompiler sederhana dan bukan interpreter sederhana, alih-alih menelusuri AST dan menjalankan beberapa tindakan, kita akan menghasilkan kode mesin saat kita menemukan setiap node.

2.4 MENJALANKAN PROGRAM

Sekarang interpreter NanoBASIC kita sudah lengkap, kita dapat menjalankan beberapa program NanoBASIC. Anda dapat menemukan program Tiny BASIC secara online yang dapat dijalankan di NanoBASIC (atau dimodifikasi jika menggunakan INPUT atau fitur lain yang tidak dimiliki NanoBASIC). Anda juga dapat menulis program NanoBASIC Anda sendiri, dan itu sebenarnya cara yang bagus untuk menguji interpreter Anda. Seperti yang disebutkan, penulis juga telah menyediakan beberapa program NanoBASIC sederhana di folder Contoh proyek. Misalnya, kita melihat *fib.bas* di awal bab. Contoh lain, *gcd.bas,* menemukan pembagi persekutuan terbesar dari dua angka yang ditentukan dalam kode sumber. Berikut cara mencari pembagi persekutuan terbesar dari 350 dan 539:

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

Sekali lagi, seperti pada Bab 1, perintah untuk menjalankan program mengasumsikan Anda berada di direktori utama repositori. Jangan lupa untuk menjalankan program sebagai modul menggunakan opsi `-m`.

2.5 PENGUJIAN NANOBASIC

Seperti halnya Brainfuck, akan sangat membantu jika kita memiliki beberapa pengujian integrasi yang memastikan interpreter kita berjalan dengan benar. Pengujian NanoBASIC sangat mirip dengan pengujian Brainfuck. Kita mengambil alih output standar dan memastikan output yang diharapkan sama dengan output sebenarnya untuk banyak program di folder Contoh:

tests/test_nanobasic.py

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

# Tokenizes, parses, and interprets a NanoBASIC
# program; stores the output in a string and returns it
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()
```

Salah satu fitur tambahan dari pengujian NanoBASIC adalah bahwa banyak di antaranya mengisolasi satu jenis pernyataan. Misalnya, *print2.bas* hanya menggunakan pernyataan `PRINT` dan ekspresi numerik, jadi jika `GOTO` tidak berfungsi dengan benar, `test_print2()` masih dapat lolos (tetapi mudah-mudahan kesalahan `GOTO` akan ditangkap oleh pengujian lain). Metodologi isolatif ini memberikan hasil pengujian yang lebih terperinci, tetapi kita masih membutuhkan pengujian integrasi yang lebih komprehensif seperti *fib.bas* untuk memastikan pernyataan-pernyataan tersebut bekerja dengan benar secara bersamaan. Serangkaian pengujian unit yang lebih kuat juga akan mencakup pengujian yang memeriksa tokenizer, parser, dan interpreter secara independen.

Anda akan menemukan bahwa semua pengujian lolos. Anda juga dapat mencoba menambahkan program BASIC buatan Anda sendiri sebagai pengujian tambahan.

KODE BERTEMU KEHIDUPAN

Ayah saya, Danny Kopec, yang merupakan seorang profesor ilmu komputer, belajar pemrograman di Dartmouth College, tempat ia mengambil kursus BASIC dengan presiden Dartmouth, John Kemeny, salah satu pencipta bahasa tersebut. Ketika penulis pertama kali belajar pemrograman sekitar usia delapan tahun di pertengahan tahun 1990-an, ia membelikan penulis salinan True BASIC, BASIC "resmi" yang dirilis oleh sebuah perusahaan yang didirikan oleh Kemeny dan salah satu pencipta BASIC, Thomas Kurtz. BASIC agak ketinggalan zaman pada tahun 1995, tetapi penulis tidak menyadarinya. penulis menghabiskan banyak waktu membuat game dengan True BASIC, meskipun penulis rasa penulis tidak pernah benar-benar mempelajari subrutin. penulis kira penulis menulis kode yang berantakan.

Seperti ayah saya, penulis akhirnya kuliah di Dartmouth untuk studi sarjana (jurusan ekonomi), dan setelah bekerja sebentar di Wall Street yang penulis benci, penulis melamar program pascasarjana di bidang ilmu komputer. Ketertarikan pada pemrograman yang dimulai dengan BASIC masih ada dalam diri saya. Bagaimana mungkin seseorang tanpa gelar ilmu komputer bisa kuliah pascasarjana di bidang ilmu komputer? penulis mengambil lima mata kuliah ilmu komputer saat kuliah S1, sebagian besar penulis mendapat nilai bagus, dan menerbitkan beberapa proyek di waktu luang saya, dan itu cukup untuk masuk ke beberapa program. Akhirnya penulis kembali ke Dartmouth untuk gelar master. Bagi siapa pun yang mempertimbangkan jalur ini, izinkan penulis mengatakan bahwa tidak memiliki gelar sarjana penuh di bidang ilmu komputer membuat transisi ke program master cukup sulit. penulis tidak akan merekomendasikan untuk mengambil gelar pascasarjana di bidang yang sangat berbeda tanpa banyak belajar mandiri.

Untuk terus memperparah kesalahan saya, selama semester pertama program master saya, penulis memutuskan untuk mengambil kelas tentang subjek yang sulit yaitu kompilator. Akhirnya itu menjadi nilai terendah dalam karier pascasarjana saya. Kelas tersebut melibatkan proyek besar membangun kompiler C dalam bahasa C (dengan beberapa fitur yang hilang). penulis memiliki seorang mitra untuk proyek tersebut, dan kami membagi pekerjaan menjadi beberapa fase kompiler, mirip dengan fase yang telah kita bahas di bab ini (tokenizer, parser, generator kode, dan sebagainya). Kompiler hanya akan berfungsi jika semua fase berfungsi.

Pada saat minggu terakhir semester tiba, penulis telah menulis beberapa ribu baris kode untuk fase saya, sementara mitra penulis telah menulis kurang dari 100 baris untuk bagiannya, meskipun penulis terus-menerus mendorongnya. penulis sering memikirkan pengalaman itu hari ini ketika penulis memberikan proyek kelompok kepada siswa: terkadang ketika mereka menyalahkan mitra mereka, mereka mengatakan yang sebenarnya. Tidak diragukan lagi kode penulis juga tidak bagus, tetapi setidaknya penulis yang menulisnya.

Kami bekerja dengan tergesa-gesa dengan beberapa malam begadang untuk membuat sesuatu berfungsi, menggabungkan sedikit kode mitra penulis dengan kode penulis dan menulis lebih banyak lagi untuk mengisi semua kekosongan. Pada akhirnya, kompiler kami melakukan beberapa hal dasar dengan benar, tetapi gagal dalam sebagian besar tes otomatis profesor. Bagaimana seseorang yang hampir gagal dalam mata kuliah kompilator akhirnya menulis bab tentang interpreter ini? Delapan tahun kemudian, dan beberapa tahun setelah memulai karier di bidang pendidikan ilmu komputer, penulis teringat pengalaman penulis dengan BASIC. penulis juga pernah menggunakan bahasa pemrograman anak-anak bernama Logo saat masih kecil, dan kedua bahasa tersebut merupakan cara yang sangat baik untuk belajar pemrograman saat masih anak-anak.

Sebagai tantangan bagi diri penulis sendiri, saya memutuskan untuk membangun bahasa pemrograman anak-anak penulis sendiri yang merupakan perpaduan antara BASIC dan Logo. Ini bukan ide baru—banyak orang telah melakukan proyek serupa—tetapi penulis ingin membuat sesuatu yang lebih baik dan membuktikan kepada diri penulis sendiri bahwa kelas kompilator tidak harus menjadi akhir dari pengembangan bahasa pemrograman bagi saya.

Hasilnya adalah produk bernama SeaTurtle yang terjual ratusan eksemplar.

Ratusan, bukan ribuan. Ini tidak akan membuat penulis kaya, tetapi ini membuktikan kepada penulis bahwa penulis dapat menulis bahasa pemrograman yang benar-benar ingin dibeli orang. Bahasa pemrograman sungguhan. Oke, bahasa pemrograman "anak-anak" sungguhan.

Pengalaman dengan SeaTurtle itu membuat saya menciptakan NanoBASIC sebagai proyek Swift untuk kelas Bahasa Pemrograman Baru penulis (disebutkan dalam kotak "Kode Bertemu Kehidupan" di Bab 1). Dan pengalaman itu membawa saya ke bab ini. NanoBASIC sangat sederhana, tetapi nyata, dan begitu Anda membangun sesuatu yang nyata, Anda tidak jauh dari membangun sesuatu yang menarik. Pada akhirnya, perspektif yang penulis peroleh dari semua pengalaman ini, termasuk kelas kompilator yang sulit, menjadikan penulis orang yang tepat untuk menulis bab ini. Sekarang setelah Anda membacanya, semoga Anda tidak perlu berjuang dengan cara yang sama seperti penulis di kelas itu.

Aplikasi di Dunia Nyata

Seperti yang telah dibahas sebelumnya dalam “Sejarah BASIC”, BASIC adalah bahasa standar revolusi komputasi pribadi. Jutaan programmer memulai karir mereka dengan menulis BASIC, dan Tiny BASIC adalah dialek BASIC yang nyata dan banyak digunakan. Beberapa pelopor Tiny BASIC mempelopori gerakan Perangkat Lunak Bebas. Berkat lisensi terbuka mereka, Tiny BASIC diporting ke berbagai platform di mana kesederhanaannya justru menjadi keuntungan. Ia berjalan di mesin dengan memori yang sangat sedikit sehingga mereka tidak dapat menjalankan bahasa yang jauh lebih canggih daripada Tiny BASIC.

Mungkin tidak tampak banyak, tetapi bagi pengguna komputer pribadi awal tanpa alternatif lain (karena biaya atau ketersediaan) dan memori terbatas, Tiny BASIC merupakan peningkatan besar dibandingkan harus menulis kode mesin. Karena dilisensikan secara bebas dan diporting ke begitu banyak platform, ia juga menawarkan semacam portabilitas untuk program yang ditulis di dalamnya. Jika sebuah mesin dapat menjalankan Tiny BASIC, maka ia dapat menjalankan program Anda.

Tiny BASIC masih digunakan hingga saat ini. Seperti Brainfuck, ia berfungsi sebagai alat pendidikan, tetapi juga digunakan untuk lebih dari itu. Saat tulisan ini dibuat, sebuah perusahaan Jerman mengirimkan mikrokontroler yang menjalankan versi Tiny BASIC.

Interpreter yang akan Anda temukan menjalankan bahasa pemrograman populer modern jauh lebih canggih daripada yang kita bangun di bab ini, namun mereka memiliki blok bangunan penting yang sama—tokenizer, parser, representasi perantara seperti AST, dan lingkungan runtime. Kecanggihan tambahan umumnya ada untuk mendukung fitur tambahan atau kinerja, tetapi teknik yang relatif sederhana dari bab ini sudah cukup untuk membangun prototipe yang berfungsi dari bahasa baru atau bahkan bahasa khusus domain (DSL) yang siap produksi untuk pekerjaan Anda. Secara umum, DSL tidak memerlukan kinerja tinggi.

Banyak bahasa pemrograman dunia nyata yang sukses dimulai dengan implementasi yang cukup sederhana dan berkembang seiring waktu. Misalnya, Ruby awalnya adalah interpreter penjelajah AST seperti NanoBASIC. Ruby kemudian dikompilasi menjadi bytecode yang dieksekusi pada mesin virtual (lebih lanjut tentang mesin virtual di Bab 5), dan versi Ruby yang lebih baru telah menggabungkan kompiler just-in-time (JIT). Baik implementasi bahasa tersebut menelusuri AST, menggunakan bytecode, atau memiliki kompiler JIT, ia membutuhkan prinsip-prinsip yang telah kita bahas dalam bab ini.

LATIHAN SOAL

1. Memungkinkan karakter tanda kutip ganda yang di-escape untuk muncul dalam string NanoBASIC. Ini akan membutuhkan sedikit pembelajaran tentang ekspresi reguler.
2. Ubah NanoBASIC menjadi Tiny BASIC dengan mengimplementasikan pernyataan INPUT, yang memungkinkan pengguna untuk mengetik nilai numerik yang disimpan dalam variabel. Ini cukup untuk mengimplementasikan Tiny BASIC "dasar" (dan untuk dapat menjalankan program Tiny BASIC yang Anda temukan secara online dengan interpreter Anda), tetapi banyak versi dunia nyata juga memiliki ekstensi, seperti cara menghasilkan angka acak yang disebut `RND().`
3. Tiny BASIC berjalan dalam mode interaktif yang mendukung perintah (dilihat sebagai pernyataan dalam tata bahasa asli) `CLEAR, LIST, RUN,` dan `END.` Baris yang dimulai dengan nomor baris disimpan. Penyimpanan tersebut dapat dihapus, dicantumkan, atau dijalankan. Program juga dapat dihentikan. Buat mode interaktif (pada dasarnya `REPL`) untuk NanoBASIC dengan keempat perintah yang sama ini. Saya mengatakan Latihan 2 akan mengubah NanoBASIC menjadi Tiny BASIC, tetapi dengan mode interaktif ini, Anda akan benar-benar mengimplementasikan Tiny BASIC.
4. Tambahkan dukungan untuk interpolasi string ke NanoBASIC. Ketika sebuah string berisi tanda dolar sebelum pengidentifikasi, periksa apakah pengidentifikasi tersebut didefinisikan dalam tabel variabel dan keluarkan nilainya sebagai penggantinya. Misalnya, "Nilai X adalah $X" akan mencetak "Nilai X adalah 24" jika variabel X memiliki nilai 24. Ini akan memerlukan modifikasi tokenizer, parser, dan runtime.
5. Tulis program NanoBASIC yang melakukan sesuatu yang menarik dan menguji setiap pernyataan dalam interpreter. Anggap ini sebagai uji integrasi akhir.

BAB 3
PEMROSESAN CITRA RETRO

Apa yang Anda lakukan ketika Anda perlu menampilkan gambar pada layar dengan warna yang lebih sedikit daripada yang ada di gambar itu sendiri? Memecahkan masalah itu adalah ranah algoritma dithering, yang secara strategis menggunakan palet warna terbatas untuk menciptakan ilusi lebih banyak warna. Dalam bab ini, kita akan menulis program yang dapat mengambil foto modern apa pun dan menampilkannya pada Macintosh monokrom klasik. Program ini akan mengkonversi foto menjadi versi hitam-putih 1-bit yang di-dither dan mengekspornya ke format yang dapat dibaca oleh Macintosh awal: MacPaint. Sepanjang jalan, kita akan mempelajari algoritma dithering, algoritma kompresi, dan sedikit tentang format file.

3.1 APA ITU DITHERING?

Algoritma dithering sengaja memasukkan noise ke dalam gambar dengan cara tertentu yang membuat gambar tampak memiliki kedalaman warna lebih dari yang sebenarnya. Trik pada mata manusia ini memiliki aplikasi praktis dan artistik. Jika Anda pernah melihat grafik gerak penuh pada konsol game atau komputer awal tahun 1990-an, maka Anda mungkin memiliki gambaran seperti apa dithering itu. Teknik ini juga lazim dalam GIF animasi, karena GIF hanya dapat mendukung 256 warna. (Ada cara curang untuk mendapatkan lebih dari 256 warna dalam GIF, tetapi sebagian besar program ekspor tidak mendukungnya.)

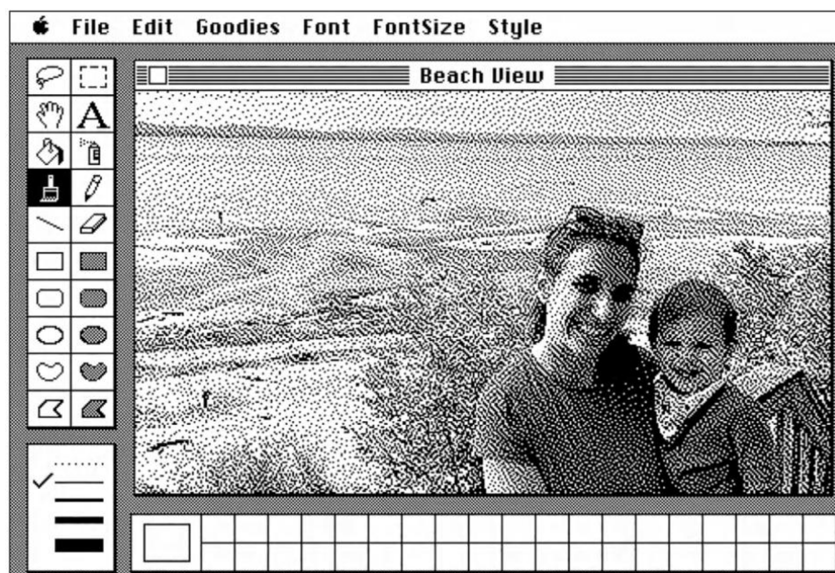
Jika Anda melihat Gambar 3.1, Anda akan melihat gambar yang sama dalam format JPEG dan GIF. JPEG memiliki 45.807 warna, sedangkan GIF hanya memiliki 256. Berkat dithering, perbedaan tersebut tidak mudah terlihat seperti yang Anda duga. (Jika Anda membaca buku ini dalam bentuk cetak, lihat direktori gambar di repositori GitHub buku ini untuk versi gambar berwarna.)

Gambar 3.1: JPEG 45.807 warna direduksi menjadi GIF 256 warna dengan dithering

Penggunaan umum lain dari dithering dan teknik sejenisnya adalah untuk membuat gambar hitam-putih tampak seperti skala abu-abu. Surat kabar telah melakukan sesuatu yang mirip dengan dithering (teknik yang disebut halftone) sejak lama untuk membuat printer hitam-putih mereka mereproduksi foto. Pada perangkat komputasi, dithering memungkinkan gambar yang memiliki kedalaman, bahkan pada layar 1-bit yang hanya dapat menampilkan dua warna (biasanya hitam dan putih). Ini adalah teknik yang masih relevan hingga saat ini. Misalnya, Panic Playdate, konsol game yang dirilis pada tahun 2022, memiliki layar hitam-putih 1-bit. Sebagian besar perangkat Amazon Kindle mendukung 16 tingkat skala abu-abu, sehingga banyak sampul buku dan foto yang ditampilkan di Kindle harus didekati melalui dithering (walaupun bukan dithering 1-bit).

Apple Macintosh asli tahun 1984 memiliki layar hitam-putih 1-bit, seperti halnya beberapa model selanjutnya. Bahkan, Apple terus menjual Macintosh 1-bit hingga Classic II dihentikan produksinya pada tahun 1993, dan perusahaan baru berhenti memberikan dukungan pengguna untuk Classic II pada tahun 2001.

Selama 17 tahun masa dukungan tersebut, pengembang pihak ketiga menciptakan banyak grafis keren untuk Macintosh monokrom, dan mereka melakukannya menggunakan algoritma dithering. Kita akan menargetkan Macintosh monokrom klasik tersebut dalam proyek kita, bersama dengan editor grafis kesayangan yang mereka jalankan, MacPaint. Gambar 3.2 menunjukkan hasil akhirnya: gambar dari Gambar 3.1 ditampilkan pada MacPaint yang berjalan pada emulator Mac Plus. (Mac Plus 1986 adalah sedikit evolusi dari Macintosh asli tahun 1984 tetapi memiliki batasan layar yang sama.) Gambar tersebut dibuat menggunakan program yang dibangun dalam bab ini.



Gambar 3.2: Pemandangan pantai dari Gambar 3-1, dikonversi oleh program kami untuk ditampilkan di MacPaint

Anda mungkin memperhatikan bahwa pemandangan pantai lebih "diperbesar" pada Gambar 3.2 daripada pada Gambar 3.1. Untuk Gambar 3.1, penulis pertama-tama memperkecil

resolusi pemandangan agar kedua versi dapat berdampingan dalam gambar yang sama (saya menggunakan perangkat lunak untuk menghitung warna versi resolusi rendah). penulis menjalankan gambar asli beresolusi penuh melalui program kami untuk mengonversinya ke MacPaint untuk Gambar 3.2. MacPaint terbatas pada dokumen yang lebarnya 576 piksel dan tingginya 720 piksel, jadi seperti yang akan kita lihat, program kami akan selalu terlebih dahulu mengubah ukuran gambar agar sesuai dengan batasan tersebut. Namun, jendela tampilan MacPaint pada Mac Plus bahkan lebih kecil karena layar Mac Plus hanya selebar 512 piksel dan beberapa piksel tersebut ditempati oleh bilah alat. Oleh karena itu, kita hanya melihat sekitar 400 dari 576 piksel pada Gambar 3.2. Tinggi penuh gambar juga terhalang.

Memulai

Alur kerja yang akan diikuti proyek kita cukup sederhana:

1. Membaca gambar dari disk.
2. Mengubah ukurannya dan mengonversinya ke skala abu-abu.
3. Melakukan dithering menjadi hitam putih.
4. Menulisnya ke disk dalam format MacPaint.

Bagian unik dan menarik dari proyek ini adalah langkah 3 dan 4, jadi kita akan menggunakan pustaka untuk menyelesaikan langkah 1 dan 2. Mungkin pustaka pengolah gambar paling populer di dunia Python adalah Pillow. Menginstalnya seharusnya semudah pip install pillow. Pillow dapat membaca gambar dalam format populer apa pun hanya dengan satu baris kode, dan hanya beberapa baris lagi untuk mengubah ukuran gambar dan mengonversinya ke skala abu-abu. Kita akan menyelesaikan pekerjaan persiapan sederhana ini langsung di file *.__main__.py* utama kita, bersamaan dengan menangani argumen baris perintah seperti yang telah kita lakukan di dua proyek sebelumnya. Mari kita mulai dengan kode untuk mengubah ukuran dan mengkonversi ke skala abu-abu, yang terdapat dalam sebuah fungsi yang diberi nama `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)
      # Size to within the bounds of the maximum for 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)
      # Convert to grayscale
```

```
    return image.convert("L")
```

Seperti yang disebutkan sebelumnya, gambar MacPaint dibatasi hingga resolusi 576 piksel lebar dan 720 piksel tinggi. Kami mendefinisikan nilai-nilai tersebut sebagai konstanta di macpaint.py.

Dalam prepare(), jika gambar terlalu besar, kami menskalakannya secara proporsional. Untuk itu, kami menentukan rasio lebar gambar terhadap tingginya dan membandingkannya dengan rasio dimensi maksimum MacPaint. Dengan menskalakan satu dimensi ke ukuran maksimum yang diizinkan di MacPaint dan dimensi lainnya sesuai dengan rasio yang tepat, kita mendapatkan gambar akhir yang sebesar mungkin di MacPaint tanpa ada bagian yang terpotong. Untuk mengetahui rumus pengubahan ukuran, penulis melakukan beberapa aljabar perkalian silang yang sangat sederhana di atas kertas. penulis sarankan Anda melakukan hal yang sama jika Anda penasaran tentang cara kerjanya.

Seperti membaca file gambar, melakukan pengubahan ukuran di Pillow sangat mudah dengan metode image.thumbnail(). Ini menawarkan beberapa algoritma bawaan untuk benar-benar menghitung warna apa yang akan dimiliki setiap piksel yang diubah ukurannya; LANCZOS mungkin merupakan kualitas tertinggi (dengan sedikit mengorbankan performa). Terakhir, image.convert("L") mengkonversi gambar ke skala abu-abu. "L" untuk mode skala abu-abu adalah singkatan dari luminansi, yang dalam grafika komputer terkadang juga dikenal sebagai luma. Sekarang mari kita tangani argumen baris perintah:

```
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)
```

Pada titik ini, di proyek ketiga kami, kami telah cukup sering melihat ArgumentParser. Untuk Retro Dither, kami memiliki beberapa opsi baris perintah tambahan dibandingkan dengan proyek-proyek sebelumnya. Salah satunya, output_file, digunakan pengguna untuk menentukan nama file dan jalur keluaran untuk hasilnya. Parameter opsional -g atau --gif digunakan pengguna untuk menentukan apakah mereka menginginkan keluaran GIF selain keluaran MacPaint. Jika pengguna meminta keluaran GIF, kami menggunakan Pillow untuk menulis versi GIF dari gambar yang telah di-dither.

Pillow adalah pustaka yang sangat lengkap dan ampuh. Kami tidak akan menggunakan banyak fiturnya di bab ini, tetapi jika Anda perlu melakukan manipulasi gambar di Python, sangat layak untuk meluangkan waktu mempelajarinya. Kami juga akan melihatnya lagi di bab berikutnya.

3.2 ALGORITMA DITHERING

Ada banyak algoritma dithering yang berbeda, kelas yang paling populer dikenal sebagai algoritma difusi kesalahan. Algoritma jenis ini mengambil sebagian perbedaan antara posisi akhir piksel dan posisi awalnya (kesalahan) dan menyebarkannya (mendifusikan) di antara piksel-piksel terdekat. Algoritma dithering difusi-kesalahan yang paling populer dikenal sebagai dithering Floyd-Steinberg (diciptakan oleh Robert Floyd dan Louis Steinberg pada tahun 1976). Bahkan, Pillow memiliki dukungan bawaan untuk dithering Floyd-Steinberg.

Hanya menggunakan Pillow untuk melakukan dithering tidak akan menyenangkan, dan kita tidak akan belajar apa pun dalam prosesnya. Sebagai gantinya, kita akan mengimplementasikan algoritma yang tidak dimiliki Pillow: algoritma dithering Atkinson. Algoritma ini diciptakan oleh Bill Atkinson pada tahun 1984 khusus untuk digunakan dengan perangkat lunak pada Macintosh asli, seperti MacPaint, yang merupakan karya Atkinson.

Oleh karena itu, tidak mengherankan jika dithering Atkinson merupakan teknik populer pada Mac monokrom. Ini akan membuat hasil kita terlihat lebih autentik. Seperti Floyd-Steinberg, dithering Atkinson adalah algoritma difusi kesalahan. Seperti yang akan kita lihat, hanya dibutuhkan beberapa perubahan untuk beralih dari satu algoritma difusi kesalahan ke algoritma lainnya.

Sebelum kita membahas detail spesifik dithering Atkinson, mari kita bahas algoritma dithering difusi kesalahan secara lebih umum. Sebagian besar algoritma ini melihat piksel gambar satu per satu dari kiri atas hingga kanan bawah. Pergerakannya dari kiri ke kanan di setiap baris, dan kemudian turun satu baris setelah setiap baris selesai. Untuk setiap piksel yang diproses, langkah-langkah berikut terjadi:

1. Temukan warna keluaran mana yang paling dekat dengannya. (Dalam dithering, warna keluaran ditentukan sebelumnya, misalnya, hitam dan putih.)
2. Temukan perbedaan antara warna keluaran dan warna asli piksel.
3. Tambahkan sebagian perbedaan ke beberapa piksel di sebelah kanan dan di bawah piksel saat ini.

Mari kita ambil langkah-langkah umum tersebut dan terapkan pada skenario spesifik kita, di mana piksel dikonversi dari skala abu-abu ke hitam dan putih melalui dithering Atkinson:

1. Temukan apakah hitam atau putih lebih dekat ke abu-abu dalam piksel. Ini akan didasarkan pada semacam ambang batas. Misalnya, jika warna abu-abu disimpan sebagai bilangan bulat tak bertanda 8-bit, mungkin ada 256 gradasi abu-abu yang diberi nomor 0 hingga 255, dan ambang batas kita bisa 127. Setiap piksel yang lebih besar dari 127 dapat ditandai sebagai putih (255 dalam skema ini), dan setiap piksel yang kurang dari atau sama dengan 127 dapat ditandai sebagai hitam (0).
2. Kurangi selisih antara warna piksel baru dan warna aslinya. Bayangkan warna abu-abu aslinya adalah 204. Selisihnya adalah 255 – 204 = 51. Ini adalah kesalahan kita.

3. Tambahkan seperdelapan dari kesalahan ke enam piksel tertentu yang dekat dengan piksel asli. Ini adalah langkah difusi. Dalam hal ini, 51 // 8 = 6 (kita perlu melakukan pembagian bilangan bulat). Piksel yang disesuaikan adalah: satu ke kanan, dua ke kanan, satu secara diagonal ke bawah dan kiri, satu secara diagonal ke bawah dan kanan, satu tepat di bawah, dan satu dua di bawah.

Hanya langkah ke-3 yang berbeda antara dithering Floyd-Steinberg dan dithering Atkinson. Anda akan melihat bahwa karena kita mendistribusikan seperdelapan dari kesalahan di antara enam piksel, kita hanya mendistribusikan total enam perdelapan atau tiga perempat dari total kesalahan. Dalam dithering Floyd-Steinberg, semua kesalahan didistribusikan di antara empat piksel yang berdekatan. Perbedaan dalam cara kesalahan didistribusikan ini memberikan dithering Atkinson tampilan yang tampaknya menekankan perubahan kontras, sedangkan dithering Floyd-Steinberg dapat memiliki tampilan yang lebih halus. Gambar 3.3 menunjukkan gambar asli (panel pertama), dithering Atkinson (panel kedua), dan dithering Floyd-Steinberg (panel ketiga).

Gambar 3.3: Nenek penulis dalam potret yang menunjukkan kontras yang baik antara berbagai warna yang menggambarkan bagaimana tepi muncul dalam dithering

Tabel 3-1 dan 3-2 berisi matriks yang mewakili difusi kesalahan dalam dithering Atkinson dan dithering Floyd-Steinberg, masing-masing. X menunjukkan lokasi piksel asli, dan header kolom dan baris menunjukkan pergerakan dalam kolom atau baris menjauh dari piksel asli.

Tabel 3-1: Difusi Kesalahan dalam Dithering Atkinson

Δ	-1	0	+1	+2
0		X	1/8	1/8
+1	1/8	1/8	1/8	
+2		1/8		

Tabel 3-2: Penyebaran Kesalahan pada Dithering Floyd-Steinberg

Δ	-1	0	+1
0		X	7/16
+1	3/16	5/16	1/16

Meskipun kita akan menerapkan dithering Atkinson, kita akan tetap memisahkan matriks sedemikian rupa sehingga mudah bagi Anda untuk memasukkan matriks yang berbeda. Misalnya, Anda dapat dengan mudah mengubah kode menjadi dithering Floyd-Steinberg atau varian difusi kesalahan lainnya, atau Anda dapat mencoba metode Anda sendiri. Bahkan, melakukan hal itu adalah salah satu latihan di akhir bab ini.

Kode dithering kita dimulai dengan mendefinisikan beberapa konstanta:

RetroDither/ dither.py

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

THRESHOLD = 127

class PatternPart(NamedTuple):
   dc: int # change in column
   dr: int # change in row
   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)]
```

Kami menetapkan `THRESHOLD` ke `127` (sekitar titik tengah antara `0` dan `255`) seperti yang dijelaskan dalam algoritma kami. Dalam `ATKINSON`, kami meratakan matriks dithering Atkinson menjadi enam tuple bernama PatternPart, yang masing-masing menentukan di mana salah satu piksel yang diubah berada relatif terhadap piksel asli dan berapa fraksi kesalahan yang harus ditambahkan ke dalamnya.

Selanjutnya, kita akan mulai mendefinisikan fungsi `dither()`, di mana dithering berlangsung:

```
# Assumes we are working with a grayscale image (Mode "L" in Pillow)
# Returns an array of dithered pixels (255 for white, 0 for black)
def dither(image: Image.Image) -> array:
   # Distribute error among nearby pixels
   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)
```

Fungsi `dither()` dimulai dengan fungsi pembantu `diffuse()` yang mengambil kesalahan dan mendistribusikannya di antara piksel-piksel terdekat dalam bagian-bagian yang ditentukan oleh sebuah pola, yang hanyalah daftar tuple `PatternPart`. Kita mengulangi setiap PatternPart dalam pola tersebut, menemukan piksel yang terkait dengan bagian tersebut, dan menambahkan sebagian kecil kesalahannya ke piksel tersebut. Seperti yang telah dibahas, pola tersebut merupakan inti dari apa yang membuat satu algoritma difusi kesalahan berbeda dari algoritma lainnya. Perhatikan bahwa semua perhitungan di sini adalah perhitungan bilangan bulat. Ini karena nilai piksel disimpan sebagai bilangan bulat.

BEBERAPA KETIDAKEFISIENAN

Dokumentasi Pillow mencatat bahwa metode `getpixel()` dan `putpixel()` cukup lambat. Pillow menyertakan akses langsung ke data piksel dalam array sebagai alternatif. Selain itu, iterasi melalui semua tuple PatternPart kurang efisien daripada menyimpan array lokasi mentah. Bahkan, karena setiap error_part tepat seperdelapan dari total kesalahan dalam dithering Atkinson, kita dapat menghemat banyak perhitungan hanya dengan membagi error dengan 8 sekali. Namun, kode ini tidak dimaksudkan untuk menjadi yang paling efisien, tetapi lebih sebagai kode yang mudah dibaca untuk sebuah bab dalam buku yang mengajarkan algoritma tersebut. Kita juga ingin dapat memasukkan algoritma dithering difusi yang berbeda di mana error_part mungkin tidak selalu sama.
Terlepas dari ketidakefisienan ini, karena kita perlu menskalakan setiap gambar terlebih dahulu agar sesuai dengan batasan MacPaint, kinerja program secara keseluruhan hampir instan. Jika kita melakukan dithering pada gambar yang lebih besar, konsesi yang dibuat di sini mungkin menjadi masalah.

Pada bagian selanjutnya dari fungsi `dither()`, kita akan menelusuri setiap piksel dalam gambar, mengubahnya menjadi hitam atau putih tergantung pada `THRESHOLD`, menghitung perbedaan dari warna abu-abu asli, dan menyebarkan kesalahan di antara piksel-piksel terdekat menggunakan fungsi bantu `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
            # Every new pixel is either solid white or solid black
            # since this is all that the original Macintosh supported
            new_pixel = 255 if old_pixel > THRESHOLD else 0
            result[y * image.width + x] = new_pixel
            difference = int(old_pixel - new_pixel)
            # Disperse error among nearby upcoming pixels
            diffuse(x, y, difference, ATKINSON)

    return result
```

Pola `ATKINSON` saat ini dikodekan secara permanen, tetapi ada cara mudah untuk mengubahnya ke pola yang berbeda. Perhatikan bahwa variabel hasil sebenarnya adalah array piksel, bukan `Pillow Image` lain. Ini karena kita perlu memproses lebih lanjut data piksel mentah dari gambar yang di-dither untuk menyimpannya dalam format MacPaint. Dalam tipe array Python dari pustaka standar, array yang didefinisikan menggunakan kode tipe 'B' menyimpan byte tanpa tanda. Karena kita akan memanipulasi banyak byte mentah, kita akan melihat tipe array ini berulang kali baik di bab ini maupun di bab-bab selanjutnya tentang emulator.

Yang membingungkan, Python menyediakan setidaknya tiga tipe berbeda untuk bekerja dengan byte mentah: `bytes`, `bytearray`, dan `array("B")`. Anda pada akhirnya dapat menulis kode Anda menggunakan salah satunya. Tipe array sangat ditujukan untuk representasi yang ringkas dan bekerja dengan file.

3.3 FORMAT FILE MACPAINT

Sebelumnya memang sudah ada program melukis, tetapi untuk semua yang muncul setelahnya, MacPaint menetapkan standar. Program ini ditulis oleh Bill Atkinson dan dirilis pada tahun 1984 oleh Apple bersamaan dengan Macintosh pertama. Meskipun didahului oleh Xerox Star dan Apple Lisa, Macintosh adalah komputer pribadi pertama yang tersedia secara luas dengan mouse dan antarmuka pengguna grafis (GUI). MacPaint adalah salah satu perangkat lunak unggulan yang menunjukkan betapa kuatnya mouse dan GUI. Seorang pengulas di New York Times mengatakan, “Ini lebih baik daripada perangkat lunak sejenis lainnya yang ditawarkan pada komputer pribadi dengan faktor 10.” Banyak alat dan teknik manipulasi grafis yang pertama kali muncul di MacPaint masih ada hingga saat ini dalam perangkat lunak grafis modern.

Saya mendorong Anda untuk mencoba MacPaint untuk memahami cara kerjanya. Ada demo langsungnya di Internet Archive, tetapi Anda mungkin mendapatkan kepuasan maksimal dari bab ini dengan meluangkan waktu untuk mengunduh emulator sehingga Anda dapat memuat gambar sebenarnya yang akan dihasilkan program kita langsung di MacPaint.

Meskipun revolusioner, MacPaint dibatasi oleh perangkat keras platform perintisnya. Seperti Macintosh asli, MacPaint hanya mendukung hitam dan putih. Seperti yang disebutkan sebelumnya, dokumennya juga terbatas pada ukuran tetap 576 piksel lebar kali 720 piksel tinggi. Macintosh asli bahkan tidak memiliki hard drive; ruang disk terbatas karena semuanya

harus dijalankan dari disket. Untuk mengakomodasi kendala ruang ini, MacPaint menggunakan skema kompresi sederhana yang dikenal sebagai pengkodean panjang run. Kita akan kembali ke beberapa keunikan ini sebentar lagi.

Langkah selanjutnya dalam proyek kita adalah menulis kode yang menerjemahkan piksel dithered dari sebuah gambar ke dalam format MacPaint. Pengalaman penulis menunjukkan bahwa pemrograman terhadap format file biner hanyalah masalah mengikuti dokumen spesifikasi dengan sangat hati-hati. Sayangnya, format MacPaint agak kurang dikenal saat ini, sehingga menemukan dokumen spesifikasi membutuhkan sedikit pencarian. Apple sendiri mendokumentasikan format tersebut dalam Catatan Teknis PT24. Namun, deskripsi yang paling mudah diakses dan komprehensif yang penulis temukan ada di situs bernama `FileFormat.info.`

Secara permukaan, file MacPaint cukup sederhana. File tersebut terdiri dari header 512 byte diikuti oleh data piksel yang dikompresi menggunakan pengkodean run-length. Sebelum dienkode run-length, setiap piksel disimpan sebagai bit, yaitu 1 untuk hitam atau 0 untuk putih. Kedengarannya cukup mudah. Namun, ada keanehan: tidak seperti hampir semua sistem operasi lainnya, sistem operasi Mac klasik menyimpan file dalam dua "cabang." Kita akan membahasnya lebih detail nanti, tetapi singkatnya, file MacPaint yang ditransfer atau dibuat pada sistem operasi selain Mac OS klasik harus dienkode dalam format khusus yang disebut MacBinary agar metadata-nya tetap ada selama transfer.

Oleh karena itu, kita perlu mengubah file output kita menjadi file MacBinary juga, dengan menambahkan header tambahan khusus. Kita akan menangani format file yang rumit ini langkah demi langkah. Pertama, kita akan menangani data piksel. Kemudian, kita akan mengimplementasikan pengkodean run-length. Dan terakhir, kita akan membuat header MacPaint dan MacBinary. Saat kita bekerja, kita perlu menggunakan berbagai operasi bitwise termasuk pergeseran, `OR,` dan `AND.` Jika manipulasi bit tingkat rendah semacam ini baru bagi Anda, atau jika Anda sedikit kurang mahir, lihat lampiran buku untuk gambaran umum operasi bitwise di Python.

3.4 MENERJEMAHKAN BYTE KE BIT

Dalam mode "L", Gambar Bantal kita dienkode menggunakan 1 byte per piksel. Namun, dalam bitmap MacPaint, piksel dikodekan dengan 1 bit per piksel, artinya setiap byte mewakili delapan piksel. Ini merupakan penghematan besar untuk hitam dan putih, dan sangat masuk akal karena kita hanya membutuhkan dua nilai (1 dan 0) untuk mewakili dua warna. Setelah kita mendefinisikan beberapa konstanta, fungsi pertama kita di macpaint.py adalah konverter yang mengambil array byte yang kita dapatkan dari `dither()` dan mengubahnya menjadi "array bit":

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
	
	# Convert an array of bytes where each byte is 0 or 255
	# to an array of bits where each byte that is 0 becomes a 1
	# and each byte that is 255 becomes a 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

Perulangan di sini mengiterasi 8 byte sekaligus dari array asli, memeriksa apakah masing-masing adalah piksel putih (255) atau hitam (0). Format bitmap MacPaint membalikkan ini, menjadikan piksel putih 0 dan piksel hitam 1. Baris `next_bit = 1 - (original[byte_index + bit_index] & 1)` melakukan pembalikan tersebut. Perhatikan bahwa kita sebenarnya tidak memeriksa nilai terhadap 255, melainkan lebih memilih untuk hanya memeriksa bit pertama (& 1) karena membuat kode lebih ringkas dan lebih berkinerja.

Kita tahu bahwa kita hanya menyimpan 255 dan 0 dalam array sebelum `bytes_to_bits()` dipanggil, jadi tidak ada alasan untuk khawatir kita secara tidak sengaja menangkap nilai perantara; jika ada angka 1 di mana pun dalam byte, itu pasti 255. Kedelapan bit tersebut dikodekan dalam `next_byte` dengan menempatkan setiap bit ke tempat yang sesuai dengan operasi OR: `next_byte = next_byte|(next_bit << (7 - indeks_bit)). Kemudian kita menambahkan next_byte ke bits_array.

Kita tidak memanggil bytes_to_bits() pada semua data piksel sekaligus, karena bitmap MacPaint perlu diisi dengan piksel putih (0) pada setiap baris bitmap di mana data piksel tidak mencapai panjang penuh. Kita menangani pengisian tersebut dengan fungsi prepare():

```
# Convert the array of bytes into bits using the helper function.
# Pad any missing spots with white bits due to the original
# image having a smaller size than 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
```

Kita menelusuri data piksel mentah yang berasal dari `dither()` satu baris demi satu baris dan mengkonversi baris tersebut menjadi bit menggunakan `bytes_to_bits()`. Jika baris tersebut tidak mencakup lebar penuh dokumen MacPaint, kita menambahkan piksel putih ke baris tersebut. Kita melakukan hal yang sama untuk setiap baris penuh di bawah data piksel.

3.5 MENERAPKAN PENGKODEAN PANJANG BERURUTAN (RUN-LENGTH ENCODING)

Menyimpan piksel sebagai bit individual daripada byte menghemat banyak ruang, tetapi itu tidak cukup. Ketika MacPaint diluncurkan pada tahun 1984, Macintosh asli memiliki disket yang hanya mendukung 400KB data. Tidak ada hard drive, dan konfigurasi standar hanya memiliki satu drive disket. Bayangkan betapa besarnya file MacPaint jika tidak dikompresi. 576 piksel dalam satu baris membutuhkan 576 bit, yaitu 72 byte. Terdapat 720 baris, jadi 720 dikalikan dengan 72 byte adalah 51.840 byte. Dengan menambahkan header 512 byte, file MacPaint akan berukuran 52.352 byte tanpa kompresi. Itu berarti disket bahkan tidak dapat menyimpan delapan file MacPaint!

Untuk mengatasi masalah ruang disk ini, format file MacPaint menggabungkan skema kompresi sederhana yang disebut pengkodean run-length. Dalam skema ini, alih-alih mengulang hal yang sama, Anda menentukan berapa kali hal itu harus diulang. Misalnya, anggaplah kita ingin menyimpan string $AAAAAABCCCCCABBBB$.

Jika setiap karakter menggunakan 1 byte, maka akan menjadi 17 byte. Bukankah akan lebih efisien untuk mengatakan tujuh A daripada mengulang A tujuh kali? Atau mengatakan lima C?

Misalkan kita menggunakan hingga 1 byte sebelum karakter untuk menyimpan jumlah pengulangannya. String tersebut kemudian dapat dikodekan sebagai $6AB5CA4B$, yang berukuran 8 byte.

Namun, ada masalah dengan skema ini. Ingat, dalam memori komputer, ini akan berupa byte mentah. Bagaimana kita tahu bahwa B adalah karakter dan bukan byte yang menunjukkan sejumlah pengulangan karakter berikutnya? Dengan kata lain, B kemungkinan akan disimpan dalam memori menggunakan kode ASCII/Unicode-nya, 66. Program kita kemungkinan akan menginterpretasikannya sebagai indikasi 66 karakter berikutnya, bukan huruf B tunggal.

Sebagai gantinya, kita dapat memiliki skema di mana setiap karakter didahului oleh angka, bahkan karakter tunggal. Ini akan mengubah string yang dikodekan menjadi $6A1B5C1A4B$. Itu 10 byte, penghematan 7 byte dibandingkan aslinya, yang signifikan. Skema pengkodean ini adalah bentuk pengkodean run-length, tetapi akan cepat rusak: untuk banyak

string, sebenarnya kurang efisien daripada hanya menyimpan karakter mentah. Misalnya, string ABC akan menjadi $1A1B1C$. Itu dua kali lipat ukurannya.

Anda dapat memiliki skema pengkodean di mana setiap angka menunjukkan pengulangan atau sejumlah karakter literal tertentu. Kemudian, ABC menjadi $3ABC$. Itu masih lebih panjang, tetapi skema baru ini mungkin merupakan kompromi yang baik untuk kasus yang lebih kompleks. Misalnya, string $AAAAABCBCAAAAAA$ akan menjadi $5A4BCBC6A$.

Versi ini mendekati skema pengkodean yang digunakan format file MacPaint, tetapi masih ada masalah: Bagaimana Anda tahu bahwa angka 5 berarti lima A, tetapi angka 4 berarti urutan literal empat karakter $(BCBC)$ daripada empat B? Anda mungkin berkata, "Yah, Anda bisa membaca dua karakter ke depan dan melihat bahwa C bukanlah angka, jadi angka 4 tidak mungkin berarti empat B," tetapi bahkan itu pun tidak berhasil, karena (sekali lagi) dalam memori komputer, C disimpan sebagai angka. Kita membutuhkan peningkatan lain.

MacPaint menghilangkan ambiguitas antara angka yang menunjukkan urutan literal dan angka yang menunjukkan urutan pengulangan dengan merepresentasikan keduanya menggunakan bilangan bulat 1-byte bertanda. Angka n antara 0 dan 127 menunjukkan n + 1 byte literal berikutnya. Angka n antara –1 dan –127 menunjukkan 1 – n pengulangan byte berikutnya. Angka –128 tidak digunakan. Skema kompresi ini dikenal sebagai PackBits, dan digunakan tidak hanya di MacPaint tetapi juga di beberapa format file populer lainnya.

Fungsi PackBits sebenarnya telah terintegrasi ke dalam versi klasik Mac OS. Menurut catatan teknis Apple sendiri tentang hal ini, "dokumen MacPaint tipikal dikompresi menjadi sekitar 10K" dengan PackBits. Tabel 3-3 merangkum skema pengkodean PackBits.

Tabel 3-3: Skema Pengkodean PackBits (Bertanda)

Nilai n	Arti
0 ke 127	Selanjutnya terdapat n + 1 byte literal.
129 ke 255	Byte berikutnya diulang 1 - n kali.
128	Skip

Kita akan bekerja dengan bilangan bulat tak bertanda, jadi lebih mudah untuk menulis ulang tabel daripada melakukan konversi pada setiap byte dari bilangan bertanda ke tak bertanda (atau memikirkan komplemen dua). Tabel 3-4 adalah skema pengkodean dengan byte yang dikonversi ke bilangan bulat tak bertanda.

Tabel 3-4: Skema Pengkodean PackBits (Tak Bertanda)

Nilai n	Arti
0 ke 127	Selanjutnya terdapat n + 1 byte literal.
129 ke 255	Byte berikutnya diulang sebanyak 257 -n kali.
128	Skip

Apakah Anda sudah memikirkan keterbatasan skema ini? Bagaimana jika ada lebih dari 128 byte dalam satu baris? Untungnya, kita tidak akan menemui masalah ini saat mengkodekan

file MacPaint, karena file tersebut dikodekan satu baris demi satu baris. Satu baris di MacPaint hanya dapat berisi 576 piksel, yang disimpan sebagai 72 byte. Karena 72 kurang dari 128, kita tidak perlu khawatir tentang batasannya.

Untuk memeriksa pemahaman Anda, cobalah mengerjakan contoh yang disediakan Apple dalam Catatan Teknis $TN1023$. penulis telah mengonversinya dari heksadesimal ke desimal untuk kenyamanan Anda di Tabel 3-5. Satu baris adalah data yang belum dikompresi, dan baris lainnya adalah data yang sudah dikompresi. Cobalah untuk merumuskan ulang data yang sudah dikompresi dari data yang belum dikompresi menggunakan Tabel 3-4 sebagai referensi. Kemudian, periksa pekerjaan Anda terhadap data yang sudah dikompresi di Tabel 3-5.

Tabel 3-5: Contoh PackBits

Type	Byte
Packed	170, 170, 170, 128, 0, 42, 170, 170, 170, 170, 128, 0, 42, 34, 170, 170, 170, 170, 170, 170, 170, 170, 170, 170
Unpacked	254, 170, 2, 128, 0, 42, 253, 170, 3, 128, 0, 42, 34, 247, 170

Mari kita implementasikan encoder PackBits. Fungsi `run_length_encode()` menerima array byte dan mengembalikan array byte yang dienkode panjang run menggunakan skema PackBits. Fungsi ini dimulai dengan fungsi pembantu internal, `take_same()`, yang dapat menemukan deretan nilai yang berulang dan mengembalikan panjangnya:

```
# MacPaint expects RLE to happen on a per-line basis (MAX_WIDTH).
# In other words there are line boundaries.
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
```

Untuk menemukan deret, `take_same()` hanya berulang kali memeriksa apakah byte setelah byte saat ini sama dengannya. Fungsi ini juga berhati-hati agar tidak melampaui akhir array sumber. Kita menambah count setiap kali ditemukan kecocokan, tetapi karena byte pertama yang diperiksa (byte di awal) bukanlah kecocokan dari byte sebelumnya, count akan selalu satu kurang dari jumlah item dalam deret. Oleh karena itu, `count + 1` dikembalikan jika ditemukan kecocokan, atau 0 jika tidak. Ini membuat tidak mungkin untuk memiliki deret berulang hanya dengan satu karakter yang sama: itu hanya akan menjadi karakter tunggal, yang akan menjadi bagian dari literal karena di PackBits tidak ada cara untuk "mengulang sekali". Dengan kata lain, domain `take_same()` adalah 0 dan semua bilangan bulat lebih besar dari atau sama dengan 2.

Fungsi run_length_encode() berlanjut dengan sedikit pengaturan:

```
rle_data = array('B')
# Divide data into MAX_WIDTH size boundaries by line
for line_start in range(0, len(original_data), MAX_WIDTH // 8):
  data = original_data[line_start:(line_start + (MAX_WIDTH // 8))]
```

Output akan disimpan dalam rle_data. Kita mengulangi array original_data satu baris demi satu baris. Format file MacPaint menentukan bahwa setiap baris dienkode run-length secara individual, bukan semua piksel dienkode run-length sekaligus. Variabel data mewakili satu baris data piksel yang siap untuk dienkode run-length. Langkah selanjutnya adalah mencari pengulangan dan run literal:

```
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
```

Kita mengulangi setiap baris data 1 byte pada satu waktu, dengan indeks melacak byte saat ini yang sedang diperiksa. Dua hitungan dikumpulkan: same (diinisialisasi secara langsung menggunakan operator yang disebut walrus :=) adalah jumlah item dalam satu baris yang sama, dan not_same adalah panjang dari sebuah deret literal. Berikut cara perhitungannya dalam loop while:

1. Kita mencoba menemukan deret yang berulang menggunakan take_same().
2. Jika upaya gagal (same adalah 0), maka ini pasti deret literal, sehingga not_same
1. ditambahkan.
2. Langkah 1 dan 2 diulang sampai deret yang berulang ditemukan (same tidak sama dengan 0) atau byte yang sedang dilihat (indeks + not_same) berada di luar akhir baris.

Ada tiga kemungkinan setelah perulangan ini:

1. Deret berulang langsung ditemukan (same kemudian tidak sama dengan 0) dan not_same tidak pernah bertambah, artinya sama dengan 0.
1. Deret literal awalnya ditemukan dan not_same bertambah hingga deret berulang ditemukan, mengisi same.
2. Deret literal awalnya ditemukan yang berlanjut hingga akhir baris, dan perulangan keluar karena index + not_same < len(data) tidak berlaku.

Karena kemungkinan kedua, ada skenario di mana same dan not_same sama-sama lebih besar dari 0. Ingatlah hal itu saat kita memeriksa sisa kode untuk fungsi ini:

```
  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
```

Ini adalah bagian yang menulis data yang dikodekan PackBits ke dalam array. Pola-pola ini diambil langsung dari Tabel 3-4. Karena kemungkinan menemukan `not_same` dan `same_run` dalam satu iterasi loop, ada dua pernyataan `if` di sini, bukan klausa `else`. Lebih lanjut, karena struktur loop `while` bagian dalam, `same_run tidak` sama akan selalu ditemukan sebelum run yang sama. Inilah sebabnya mengapa pernyataan `if` untuk tidak sama muncul pertama. Jika ada satu run untuk masing-masing, maka indeks akan ditambah dengan jumlah yang tepat oleh tidak sama agar berada di tempat yang tepat untuk pengkodean run yang sama.

IMPLEMENTASI ALTERNATIF

Apakah Anda menganggap fungsi `run_length_encode()` elegan atau terlalu rumit? penulis telah beberapa kali mencoba menulis ulang versi asli penulis dalam bentuk yang lebih ringkas dan mudah dibaca, dan hasilnya adalah seperti yang Anda lihat di sini. Penulis menemukan bahwa memusatkan kode pada `take_same()` dan hanya menghitung ketika gagal lebih mudah dibaca daripada mencoba secara bersamaan untuk menetapkan run yang sama dan tidak sama. Namun, ini kurang efisien daripada versi asli saya, yang menggunakan lebih banyak kondisi; sedikit lebih panjang; dan tidak memiliki fungsi internal. Jika Anda tidak menyukai versi ini, penulis meninggalkan versi asli penulis sebagai komentar di bagian bawah file sumber di GitHub.

3.6 PENGUJIAN PENGKODEAN PANJANG BERURUTAN

Saat penulis mencoba menulis ulang `run_length_encoding()` dengan beberapa cara berbeda agar lebih mudah dibaca, penulis menyadari bahwa penulis membutuhkan cara cepat untuk memastikan implementasi baru penulis benar, jadi penulis menulis beberapa pengujian unit. Seperti semua pengujian untuk buku ini, file untuk pengujian ini muncul di direktori tests di root repositori kode sumber.

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

```
class RetroDitherTestCase(unittest.TestCase):
  #Example from
  #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)

  #Example where packed data is longer than unpacked data
  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()
```

Tes-tes ini memastikan pengkodean run-length berfungsi dengan benar dan menyimpulkan implementasi format file MacPaint kami. Hanya ada satu langkah lagi untuk membuat gambar dithered kami siap digunakan pada Mac retro.

Konversi ke MacBinary

Pada sebagian besar sistem operasi, sebuah file hanyalah gumpalan data tunggal. Sistem file atau sistem operasi mungkin menyimpan beberapa metadata tentang setiap file, tetapi file itu sendiri berdiri sendiri. Hal itu tidak terjadi pada Mac OS klasik, di mana banyak file memiliki dua cabang. Cabang data akan menyimpan data utama file, sementara cabang sumber daya dapat menyimpan data tambahan seperti bitmap, suara, atau bahkan kode yang dapat dieksekusi. Cabang sumber daya dapat menyimpan berbagai jenis data di satu tempat, sehingga seperti basis data sumber daya. Ini adalah salah satu fitur sistem operasi yang memungkinkan banyak aplikasi menjadi sepenuhnya mandiri—sebuah file yang dapat dieksekusi tunggal yang dapat diseret dan dijatuhkan sebagai ikon tunggal, tanpa file tambahan.

Sayangnya, karena resource fork tidak ada di sistem operasi lain, file Mac klasik sering kali bermasalah saat ditransfer ke atau dari sistem operasi tersebut. Perlu kehati-hatian. Ini adalah masalah sejak awal, sehingga format file "bundling" dengan cepat dikembangkan. Salah satu yang paling populer dan terstandarisasi dengan baik dikenal sebagai MacBinary. Dalam file MacBinary, header khusus diikuti oleh data fork dan resource fork secara bersamaan.

Kita perlu menggabungkan file MacPaint kita sebagai file MacBinary agar dapat berfungsi dengan benar di Mac OS klasik dan terbuka secara default di MacPaint. Ironisnya, file MacPaint sebenarnya tidak memiliki resource fork, hanya memiliki data fork. Tetapi file MacBinary juga menggabungkan metadata yang disimpan dalam sistem file (MFS/HFS/HFS+) dari Mac OS klasik.

Bagian pentingnya adalah kode tipe dan kode pembuat. Mac OS klasik tidak menggunakan ekstensi file untuk mengaitkan file dengan program yang seharusnya membukanya. Sebaliknya, ia menggunakan kode tipe dan kode pembuat, yang sebagian besar transparan bagi pengguna. Ini memungkinkan pengguna untuk memberi nama file mereka sesuka hati dan tetap dapat "diklik dua kali". Setelah file MacBinary yang dibuat program kita di-unbundled oleh MacBinary (atau program lain seperti Stuffit) pada Mac OS klasik, file MacPaint yang dihasilkan akan membuka MacPaint saat diklik dua kali.

Untungnya, format file MacBinary cukup sederhana. Untuk mematuhi spesifikasi MacBinary, program kita perlu:

1. Menambahkan header MacBinary 128 byte sebelum sisa file (yang hanya merupakan data fork karena kita tidak memiliki resource fork).
2. Mengisi header MacBinary dengan nilai yang tepat di beberapa tempat yang ditunjukkan.
3. Memastikan file diakhiri dengan kelipatan 128 byte dengan menambahkan padding di bagian akhirnya jika perlu.

MacBinary memiliki spesifikasi resmi yang telah disetujui oleh komite yang terdiri dari pihak-pihak yang berkepentingan. Namun, kita hanya perlu mengisi beberapa kolom agar file MacBinary kita dapat dikenali dengan benar. (Bagian header lainnya harus berupa 0.) Tabel 3-6 mencantumkan nilai-nilai ini, panjangnya, dan offset masing-masing dalam header 128 byte.

Tabel 3-6: Kolom Header MacBinary yang Wajib Diisi

Offset	Panjang	Tipe	Nilai
1	1	Integer	Panjang nama file (maksimal 63 karakter)
2	1 ke 63	MacRoman	Nama File
65	4	MacRoman	Jenis file (seharusnya "PNTG")
69	4	MacRoman	Pembuat file (seharusnya "MPNT")
83	4	Integer	Panjang garpu data
91	4	Integer	Waktu pembuatan dalam detik sejak 1/1/1904
95	4	Integer	Waktu modifikasi dalam detik sejak 1/1/1904

Perhatikan bahwa nilai disimpan dalam format big-endian, karena Mac OS klasik berjalan pada mikroprosesor big-endian. (Jika itu tidak berarti apa-apa bagi Anda, lihat kotak "Big-Endian vs. Little-Endian".) MacRoman adalah pengkodean karakter yang digunakan pada Mac OS klasik.8

Fungsi `macbinary_header()` adalah kodifikasi dari Tabel 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) # filename length
   macbinary[2:(2 + len(filename))] = array("B", filename.encode("mac_roman")) # filename
   macbinary[65:69] = array("B", "PNTG".encode("mac_roman")) # file type
   macbinary[69:73] = array("B", "MPNT".encode("mac_roman")) # file creator
   macbinary[83:87] = array("B", data_size.to_bytes(4, byteorder='big')) # size of data fork
   timestamp = int((datetime.now() - datetime(1904, 1, 1)).total_seconds()) # Mac timestamp
   macbinary[91:95] = array("B", timestamp.to_bytes(4, byteorder='big')) # creation stamp
   macbinary[95:99] = array("B", timestamp.to_bytes(4, byteorder='big')) # modification stamp
   return macbinary
```

Nama file yang lebih dari 63 karakter akan dipotong bagian akhirnya.

BIG-ENDIAN VS. LIFT-ENDIAN

Bagaimana urutan byte yang mewakili suatu data, seperti angka, harus disimpan? Ini adalah pertanyaan yang banyak diperdebatkan, dengan dunia ilmu komputer terbagi menjadi dua kubu: big-endian dan little-endian.

Coba pikirkan sejenak tentang bagaimana kita merepresentasikan angka dalam kehidupan sehari-hari.
Jika Anda berasal dari budaya yang menggunakan sistem angka Arab, seperti dunia berbahasa Inggris, Anda mungkin terbiasa menulis angka dari kiri ke kanan, dimulai dengan angka yang mewakili bagian terbesar dari angka tersebut dan menurun dari sana. Misalnya, angka 450 dimulai dengan angka 4 yang mewakili ratusan, kemudian angka 5 yang mewakili puluhan, dan kemudian angka 0 yang mewakili satuan. Angka 4 mewakili bagian terbesar dari angka tersebut (400) dan diletakkan pertama. Keputusan untuk meletakkan angka 4 pertama dibuat sejak lama dan sebagian besar bersifat arbitrer. Secara teori, kita bisa memiliki sistem angka yang menuliskan 450 dari terkecil hingga terbesar sebagai 054, tetapi jelas kita tidak memilikinya.

Angka 450 membutuhkan 2 byte untuk direpresentasikan dalam biner: 00000001 dan 11000010. Jika Anda mengetahui biner, Anda tahu bahwa setiap 1 mewakili satu pangkat 2 yang, jika ditambahkan dengan pangkat 2 "lainnya", akan menghasilkan angka akhir. Byte pertama, 00000001, menempatkan angka 1 tunggalnya di posisi ke-28 dan mewakili 256. Byte kedua, 11000010, mewakili angka 194 karena angka 1 untuk 21 (2), 26 (64), dan 27 (128) diaktifkan, dan 128 + 64 + 2 = 194. Kita memiliki byte untuk 256 dan 194, dan 256 + 194 = 450.

Karena kita menulis 450 dari yang terbesar ke yang terkecil, Anda mungkin berasumsi bahwa komputer Anda akan menyimpan byte yang mewakili bagian terbesar dari 450 terlebih dahulu, menghasilkan 0000000111000010 ketika byte-byte tersebut digabungkan. Ini dikenal sebagai urutan big-endian, tetapi bukan seperti itu cara kerja sebagian besar komputer saat ini. Komputer modern pada umumnya dibangun dengan mikroprosesor yang menggunakan salah satu dari dua arsitektur: x86-64 (Intel, AMD) atau ARM64 (Apple, Qualcomm, dan sebagainya). Karena alasan teknis dan historis, arsitektur tersebut menyimpan angka dalam urutan little-endian, di mana byte yang mewakili ujung terkecil dari angka tersebut berada di urutan pertama. Dalam sistem little-endian, angka 2-byte 450 disimpan sebagai 1100001000000001. Penting untuk mengetahui sistem mana yang digunakan, karena menginterpretasikan 1100001000000001 dalam urutan little-endian seolah-olah dalam urutan big-endian akan menghasilkan nilai yang sama sekali berbeda.

Meskipun urutan little-endian mendominasi arsitektur komputer saat ini, sistem tertentu, seperti yang berjalan pada arsitektur mikroprosesor 68K (awalnya oleh Motorola)—termasuk Macintosh asli—menyimpan angka-angkanya dalam urutan big-endian. Ya, pada Macintosh asli, 450 disimpan "dengan benar" sebagai 0000000111000010, tetapi pada komputer di depan Anda,

kemungkinan besar disimpan sebagai 1100001000000001. Lebih rumit lagi, sebagian besar data yang ditransmisikan melalui internet dikirim dalam urutan big-endian. Saat Anda menjelajahi web di mikroprosesor x86-64 atau ARM64 Anda, ada konversi endian yang terjadi di latar belakang.

Menyatukan Semuanya

Untuk menulis file MacPaint kita yang dibundel sebagai file MacBinary, kita perlu mengambil array piksel kita dari `dither()` dan:

1. Memanggil `prepare()` untuk mengonversinya dari byte ke bit dan menambahkan 0 di belakangnya.
2. Memanggil `run_length_encode()` untuk melakukan pengkodean run-length pada array bit.
3. Memanggil `macbinary_header()` untuk menggabungkan hasilnya dengan header MacBinary.
4. Menambahkan header MacPaint 512 byte juga.
5. Menambahkan 0 di belakang hasil akhir hingga kelipatan 128 byte untuk mengikuti spesifikasi MacBinary yang mensyaratkan hal ini.

Ini sebagian besar hanya masalah memanggil fungsi yang sudah kita miliki:

```
# Writes array *data* to *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 # header requires this
    output = macbinary_header(out_file,data_size)+array('B', [0] * HEADER_LENGTH)+rle
    output[MACBINARY_LENGTH + 3] = 2 # Data Fork Header Signature ❶
    # MacBinary format requires that there be padding of 0s up to a
    # multiple of 128 bytes for the data fork
    padding = 128 - (data_size % 128)
    if padding > 0:
        output += array('B', [0] * padding)
    with open(out_file + ".bin", "wb") as fp:
        output.tofile(fp)
```

Satu-satunya bagian kode ini yang belum kita bahas adalah header MacPaint berukuran 512 byte. MacPaint sebagian besar menggunakan header ini untuk menyimpan data pola yang ditentukan pengguna. Program yang diekspor ke MacPaint umumnya tidak memiliki pola yang ditentukan pengguna karena pola tersebut merupakan artefak yang dibuat oleh pengguna di MacPaint. Oleh karena itu, kita dapat membiarkan sebagian besar header MacPaint sebagai 0. Satu-satunya hal yang harus kita lakukan adalah menambahkan sedikit tanda tangan ke header MacPaint pada byte ke-3, yang selalu diatur ke 2 ❶

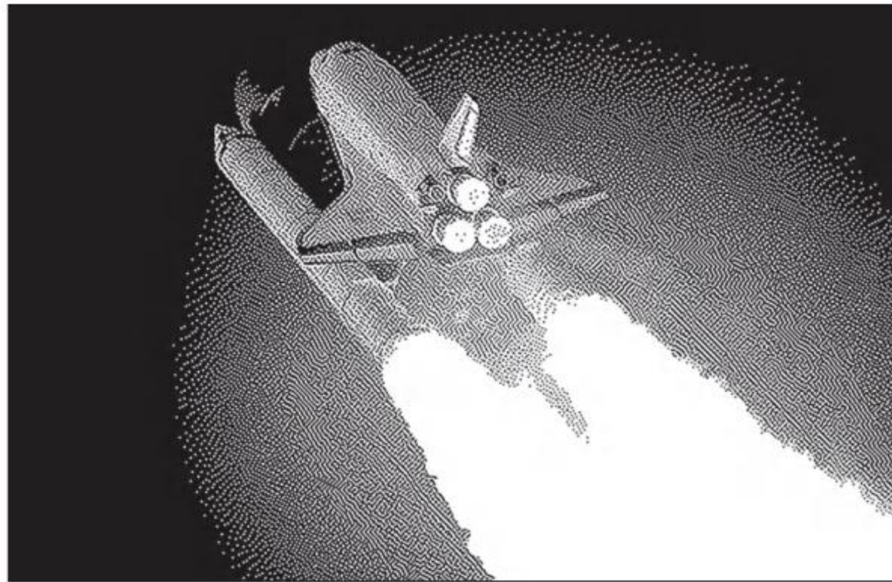
Hasilnya

Untuk menjalankan program, Anda perlu menentukan file input dan file output. Misalnya:

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

Program ini menambahkan ekstensi .bin untuk file output secara otomatis.

Gambar 3-4 menunjukkan gambar yang dibuat menggunakan program kami yang menurut penulis memiliki kualitas artistik yang bagus.



Gambar 3-4: Pesawat ulang-alik lepas landas

Tapi bagaimana Anda bisa melihat hasil karya Anda sendiri? Tentu saja, di Mac klasik! MacPaint berjalan di mesin dengan System 1 hingga Mac OS 9. Anda seharusnya dapat menemukan Mac klasik sungguhan yang menjalankan salah satu OS ini untuk menampilkan foto Anda. Anda juga memerlukan program untuk membuka bundel file MacBinary. Ada program yang diberi nama MacBinary yang dapat Anda unduh. Stuffit Expander, aplikasi dekompresi yang sangat populer untuk Mac OS klasik, juga dapat membuka file MacBinary.

Keduanya didistribusikan sebagai perangkat lunak gratis. Sebagian besar Mac yang Anda temukan di pasaran dari tahun 1990-an akan memiliki Stuffit Expander yang sudah terpasang.

Saya menyadari bahwa tidak semua orang akan cukup berkomitmen pada proyek ini untuk pergi dan menemukan komputer retro sungguhan. Tidak masalah—Anda masih dapat menikmati hasil kerja keras Anda melalui emulasi. Apple mulai mendistribusikan versi awal Mac OS dan MacPaint (dengan merilis kode sumber MacPaint lengkap) secara gratis bertahun-tahun yang lalu. Dimungkinkan untuk mendapatkan semua perangkat lunak yang Anda butuhkan, secara legal dan tanpa biaya, untuk menciptakan pengalaman Macintosh retro sejati di emulator.

Ada beberapa emulator yang tersedia, tetapi mungkin yang paling retro adalah Mini vMac, yang penulis gunakan untuk membuat beberapa tangkapan layar di bab ini. Menyiapkan Mini vMac atau salah satu dari beberapa emulator Mac klasik populer lainnya (Basilisk II, SheepShaver, dan sejenisnya) berada di luar cakupan buku ini, tetapi penulis pikir, sebagai seorang programmer, Anda akan merasa cukup mudah untuk melakukannya. Setiap emulator memiliki mekanisme yang berbeda untuk mentransfer file dari mesin Anda ke lingkungan yang diemulasikan. Terlepas dari spesifikasinya, mungkin akan lebih mudah daripada mendapatkan Mac lama dan menghubungkan Mac tersebut ke internet atau menemukan drive disket untuk mesin modern Anda. Meskipun demikian, Mac lama itu menyenangkan! Atau mungkin kesenangan itu hanyalah nostalgia penulis karena tumbuh bersama mereka.

KODE BERTEMU KEHIDUPAN

Saya telah menggunakan MacPaint sejak ayah saya membawa pulang Macintosh LC pada tahun 1990. Saya baru berusia tiga tahun, tetapi tampaknya saya menjadi cukup mahir menggunakannya sehingga ia mengajak saya untuk memberikan demonstrasi di kelas yang dia ajarkan di Universitas Maine. Mungkin sebagian karena faktor kebaruan, atau mungkin idenya adalah, "Ini sangat mudah, anak berusia tiga tahun pun bisa melakukannya!" Apa pun alasannya, ia selalu percaya pada saya.

Saya masih menyimpan beberapa gambar masa kecil saya di disk, yang membuat saya tertarik pada format file ini baru-baru ini. Anehnya, saya tidak dapat menemukan program apa pun di Mac modern saya yang dapat membuka format MacPaint klasik. Saya harus beralih ke LibreOffice atau mentransfer file ke Mac lama untuk melihatnya.

Kemudian, saya menemukan beberapa file MacPaint misterius yang semakin membangkitkan minat saya pada disket yang pernah menjadi milik saudara laki-laki saya 30 tahun yang lalu. File-file itu berisi karya seni yang aneh, cukup canggih untuk akhir tahun 1980-an atau awal 1990-an, mencampur objek dunia nyata yang didigitalkan (istilah tahun 1980-an untuk dipindai dan di-dither) dengan gambar digital murni. Akhirnya saya menyimpulkan bahwa kemungkinan besar itu adalah karya seorang seniman Amerika terkenal bernama Robert W. Fichter atau karya seseorang yang mengaguminya. File-file itu memiliki beberapa motif yang sama dan kata-kata yang persis sama dengan beberapa karya Fichter yang telah diterbitkan.

Apakah saya memiliki peninggalan seni digital berharga yang hilang ditelan waktu? Bagaimana saudara laki-laki saya mendapatkan file MacPaint ini? Saya

meneleponnya. Dia sudah tidak memikirkannya selama beberapa dekade, tetapi dia mengatakan dia mendapatkannya dari seorang mahasiswa di Universitas Maine saat dia masih di sekolah menengah. Dia tidak tahu asal-usulnya yang tepat, tetapi mahasiswa itu mengatakan kepadanya bahwa file-file itu sangat penting dan memiliki makna tersembunyi. Masuk akal bahwa disk itu berasal dari universitas. Mungkin Fichter pernah memberikan kuliah di sana, atau mungkin karya-karyanya disalin melalui sneakernet (ini terjadi di era sebelum jaringan internet tersebar luas). Saya mencoba menghubungi Fichter sendiri, tetapi tidak berhasil. Sayangnya, ia meninggal pada tahun 2023. Sampai saat penulisan ini, saya masih tidak tahu apakah file-file tersebut adalah karya seni digital asli Robert W. Fichter. Jika Anda seorang ahli tentang karyanya atau mengenalnya, silakan hubungi saya!

Pada waktu yang hampir bersamaan, saya berpikir untuk mengerjakan proyek dithering untuk buku ini setelah menemukan artikel oleh John Earnest tentang dithering Atkinson di Hacker News. Saya memutuskan bahwa algoritma dithering itu sendiri terlalu sederhana untuk sebuah bab buku, tetapi kemudian saya mendapat ide: format MacPaint yang baru-baru ini sangat saya minati juga merupakan produk Bill Atkinson, dan format tersebut mencakup algoritma menarik lainnya, yaitu pengkodean run-length. Mengapa saya tidak menggabungkan keduanya menjadi satu proyek?

Saya tidak berhenti di situ. Setelah menulis kode untuk bab ini, saya memutuskan untuk membuat MacPaint lebih mudah diakses oleh pengguna Mac modern. Saya memindahkan kode tersebut, ditambah beberapa algoritma dithering tambahan, ke Swift dan membuat antarmuka pengguna berbasis AppKit yang bagus di sekitarnya. Saya menjual perangkat lunak tersebut sebagai Retro Dither di Mac App Store. Sebagai penulis teknis, biasanya hal-hal yang Anda lakukan sebagai proyek profesional atau hobi berubah menjadi materi untuk sebuah buku; tidak biasa jika hal-hal berjalan sebaliknya.

Aplikasi Di Dunia Nyata

Atkinson dithering dan MacPaint hanyalah beberapa teknologi yang digunakan Bill Atkinson untuk menghidupkan grafis pada Macintosh asli. Dia juga pencipta QuickDraw, yang menyediakan primitif grafis pada semua komputer Lisa dan Macintosh klasik. Kontribusinya yang luar biasa juga jauh melampaui grafis. Dia membuat beberapa penyempurnaan pada elemen yang sekarang kita anggap sebagai widget standar dalam GUI. Atkinson juga pencipta HyperCard, salah satu platform hiperteks pertama yang didistribusikan secara luas. Anda dapat menganggapnya sebagai versi web non-jaringan dari akhir tahun 1980-an. Itu sangat berpengaruh.

Anda mungkin penasaran untuk apa Bill Atkinson awalnya mengembangkan Atkinson dithering. Jadi penulis mendapatkan salinan buku langka berjudul Inside MacPaint karya Jeffrey S. Young dan diterbitkan oleh Microsoft Press pada tahun 1985. Penulis tidak menemukan jawaban pasti, tetapi penulis menemukan jawaban yang mungkin. Ternyata Atkinson pernah mengerjakan digitizer untuk Macintosh awal. Pada dasarnya, ini semacam pemindai yang memungkinkan Anda untuk bekerja dengan gambar dunia nyata di MacPaint. Meskipun buku tersebut tidak secara eksplisit menyebutkan penggunaan dithering Atkinson, penulis tidak dapat membayangkan bahwa itu adalah kebetulan. Kemungkinan besar aplikasi dunia nyata pertama dari dithering Atkinson adalah digitizer.

Dithering adalah teknik yang banyak digunakan pada konsol game dan komputer tahun 1980-an dan 1990-an, yang cukup kuat untuk mendukung gambar digital tetapi seringkali memiliki palet warna yang terbatas. Dan perangkat dengan palet terbatas seperti Amazon Kindle dan Panic Playdate terus menjadi benteng dithering. Seperti yang disebutkan sebelumnya, dithering juga merupakan cara GIF animasi tampak menampilkan lebih banyak warna daripada yang didukung secara alami. Tanpa peretasan, GIF terbatas pada 256 warna.

Pengkodean run-length tentu saja tidak terbatas pada MacPaint. Ini adalah teknik kompresi yang banyak digunakan. Segala jenis format data yang memiliki banyak karakter berulang merupakan kandidat untuk pengkodean run-length. Di luar MacPaint, teknik ini digunakan sebagai teknik kompresi utama dalam beberapa format gambar bitmap lainnya pada tahun 1980-an. Teknik ini juga terkadang dikombinasikan dengan teknik kompresi lain untuk merumuskan meta-algoritma yang lebih canggih. Misalnya, salah satu komponen `DEFLATE`, algoritma yang digunakan dalam file ZIP, memanfaatkan pengkodean run-length.

Saya akan menutup dengan kutipan dari Bill Atkinson, dari wawancara yang dia lakukan di Inside MacPaint. Dia ditanya oleh Jeffrey Young, “Apa yang Anda anggap penting untuk menciptakan program yang hebat?”

Kunci utama dalam mendesain program yang baik adalah memutuskan hal-hal apa yang harus dihilangkan. Penulis memiliki beberapa fitur canggih yang saya hilangkan untuk membuat MacPaint lebih bersih, lebih sederhana, lebih mudah didekati, dan tidak terlalu menakutkan. Penulis mungkin membuang lebih banyak kode daripada yang saya pertahankan. Tujuan penulis adalah desain yang ramping, efisien, dan bersih. Proses pemrograman tipikal adalah 95 persen debugging dan hanya 5 persen kreasi nyata. Sebagian besar bug hanyalah kesalahan ketik sederhana, atau hal-hal yang sebenarnya dapat dibantu oleh program kompilasi. Penulis lebih suka memfokuskan perhatian pada algoritma secara keseluruhan, karena di situlah penulis mendapatkan kemenangan besar.

Saya membandingkan pemrograman dengan pembuatan model menggunakan tanah liat. Saat Anda membuat pot di atas roda putar, Anda ingin menjaganya tetap lunak dan fleksibel selama mungkin sebelum membakarnya. Karena setelah dibakar, pot akan jauh lebih sulit dibentuk.

LATIHAN SOAL

1. Tambahkan opsi baris perintah yang mengubah program kita untuk menggunakan dithering Floyd-Steinberg.
2. Cobalah membuat pola dithering difusi kesalahan Anda sendiri.
3. Tulis program yang dapat melakukan sebaliknya, yaitu mengkonversi file MacPaint menjadi GIF atau PNG.
4. Jika Anda telah menyelesaikan Latihan 3, tulis pengujian integrasi yang memeriksa apakah file MacPaint yang dikonversi ke GIF atau PNG mempertahankan data piksel yang sama dengan aslinya.

BAB 4
ALGORITMA MELUKIS STOKASTIK

Semua bab lain dalam buku ini mengikuti spesifikasi seperti tata bahasa bahasa pemrograman, format file, atau arsitektur mesin. Bab ini berbeda. Dalam bab ini, kita akan menciptakan karya seni. Dan itu akan bersifat subjektif. Dapatkah algoritma stokastik sederhana menggunakan bentuk yang dihasilkan secara acak untuk menciptakan gambar yang menyerupai karya seni manusia? Saya pikir jawabannya adalah ya, tetapi Anda harus menilainya sendiri setelah melihat beberapa output program.

4.1 PENDAHULUAN

Program dalam bab ini bekerja dengan mencoba menggambar ulang sebuah foto dari awal. Kita mulai dengan kanvas kosong. Kita mencoba menggambar satu bentuk berwarna dengan ukuran dan penempatan acak. Jika bentuk tersebut membuat kanvas terlihat lebih mirip dengan foto, maka kita mempertahankannya. Jika tidak, kita mencoba bentuk yang berbeda. Kita mengulangi proses ini untuk sejumlah iterasi yang ditentukan. Itulah keseluruhan algoritmanya.

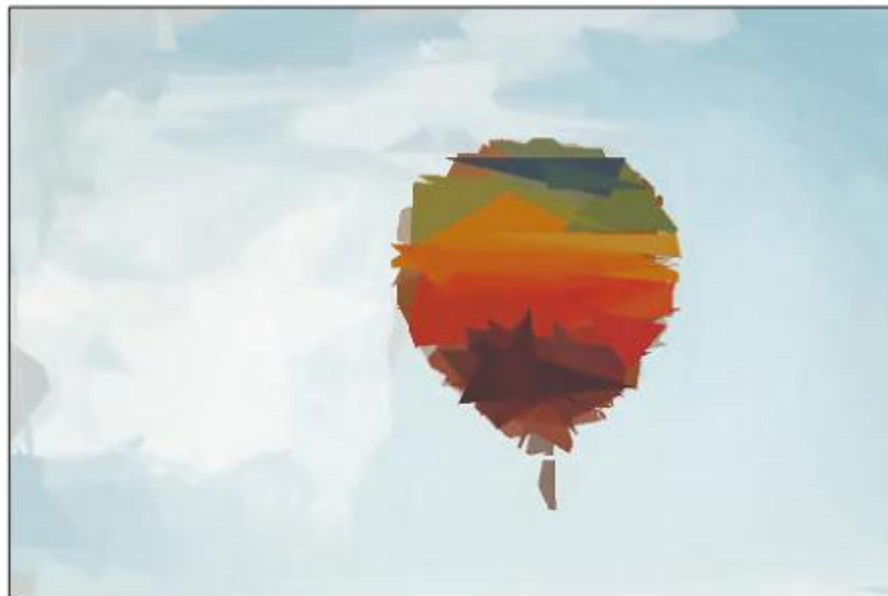
Gambar 4-1: Balon udara panas dengan 540 elips

Jika kedengarannya sederhana, itu karena memang sederhana. Tentu saja, ada banyak detail yang lebih kompleks untuk diisi, tetapi itu tidak mengubah inti utama program ini. Bagaimana "kemiripan" diukur? Haruskah suatu bentuk dimodifikasi untuk mencoba meningkatkan kesesuaiannya? Bagaimana warna bentuk dipilih? Jenis bentuk apa yang harus digunakan?

Pertanyaan terakhir itu akan mengarah pada banyak tampilan abstrak yang berbeda untuk "lukisan" kita. Misalnya, Gambar 4-1 menggunakan elips untuk mendekati foto balon udara panas. (Untuk pembaca cetak, lihat direktori gambar dari repositori pendamping untuk versi berwarna dari gambar bab ini.)

Gambar tersebut menunjukkan foto asli dan "kesan" yang dibuat oleh program kami menggunakan 540 elips selama 100.000 iterasi, yang membutuhkan waktu 104 detik untuk diselesaikan di laptop saya. penulis pikir hasilnya cukup bagus. Hampir impresionistik.

Menurut saya, elips memberikan tampilan seperti kaca patri. Akan lebih baik jika bentuk yang digunakan agak menyerupai kontur subjek dalam foto asli, seperti elips untuk balon udara panas. Gambar 4-2 menunjukkan foto yang sama yang dilukis menggunakan 307 segitiga selama 100.000 iterasi, membutuhkan waktu 109 detik di laptop saya. Menurut pendapat subjektif saya, hasilnya tidak sebagus yang pertama.



Gambar 4-2: Balon udara panas dengan 307 segitiga

Meskipun bagian luar balon udara panas yang melengkung tidak terisi dengan baik oleh segitiga, kupu-kupu jauh lebih baik. Gambar 4-3 adalah kesan dari foto kupu-kupu, menggunakan 573 segitiga yang dihasilkan dengan 1 juta iterasi selama 4.512 detik.

Gambar 4-3: Seekor kupu-kupu dengan 573 segitiga

Gambar kupu-kupu ini membutuhkan waktu sedikit lebih lama untuk dihasilkan daripada gambar rata-rata yang membutuhkan jutaan iterasi karena penulis menggunakan metode "warna paling umum" alih-alih metode "warna rata-rata"—lebih lanjut tentang itu nanti. Hasilnya adalah warna di setiap bentuk lebih tajam dan tidak terlalu menyatu.

Hasilnya seringkali dapat ditingkatkan dengan menggunakan lebih banyak bentuk yang ditempatkan dalam lebih banyak iterasi, dan karenanya membutuhkan lebih banyak waktu. Namun, karena ini didasarkan pada proses stokastik (ditentukan secara acak), hasilnya akan sangat bervariasi—bahkan dengan pengaturan yang sama untuk gambar yang sama. Contoh yang penulis sajikan di sini dipilih secara selektif.

Foto dengan banyak detail membutuhkan waktu paling lama untuk dilukis dengan tingkat akurasi yang wajar. Dengan alat seperti ini, Anda harus mencapai keseimbangan antara abstraksi dan pengenalan. Jika sebuah gambar terlalu abstrak, gambar tersebut tidak akan mudah dikenali.

Namun, jika sebuah gambar memiliki detail yang sangat banyak sehingga hampir menjadi replika sempurna dari aslinya, maka gambar tersebut akan kehilangan daya tariknya sebagai "karya seni."

Keseimbangan ini sangat sulit dicapai dengan gambar orang. Misalnya, Gambar 4-4 adalah gambar dua teman penulis di pantai di Santa Monica, California. Gambar ini terdiri dari 4.212 elips yang dihasilkan dengan 10 juta iterasi selama 8.262 detik. Teman-teman penulis hanya terlihat biasa saja sebagai kesan abstrak, tetapi pantainya terlihat luar biasa!

Gambar 4-4: Pemandangan Santa Monica dengan 4.212 elips

Saya menemukan bahwa menggunakan bentuk garis sangat efektif untuk membuat lukisan orang. Karena garis sangat tipis, dibutuhkan lebih banyak garis untuk menggambar sebuah gambar. Gambar 4-5 menunjukkan gambar domain publik kecil John F. Kennedy yang menyampaikan pidato terkenalnya di Berlin, bersama dengan versi lukisan yang menggunakan 16.633 garis yang dihasilkan dengan 10 juta iterasi selama 6.957 detik.

Gambar 4-5: Pidato JFK di Berlin dengan 16.633 garis

Aspek menarik dari contoh JFK adalah bahwa versi abstrak yang dihasilkan program sebenarnya memiliki resolusi lebih tinggi daripada aslinya. Hal ini dimungkinkan karena program tersebut bekerja di dunia grafis vektor, di mana matematika menentukan output, bukan piksel. Algoritma tersebut tidak dengan tekun menyalin setiap piksel—

algoritma tersebut memberikan "kesan" dari aslinya dengan bentuk vektor.

Garis menghasilkan beberapa hasil abstrak yang paling menakjubkan. Gambar 4-6 adalah cakrawala Kota New York dengan Jembatan Manhattan di depannya melalui 12.303 garis yang dihasilkan dengan 1 juta iterasi selama 909 detik.

Gambar 4-6: Pemandangan kota New York dengan 12.303 garis

Program yang dikembangkan dalam bab ini akan membutuhkan waktu lama untuk dijalankan jika Anda ingin menggunakan banyak bentuk. Misalnya, gambar JFK membutuhkan waktu hampir dua jam untuk dijalankan di laptop Apple M1 saya. Waktu yang dibutuhkan akan bervariasi tergantung pada mikroprosesor mesin Anda. Secara umum, Anda dapat membiarkan program berjalan di latar belakang saat Anda melakukan pekerjaan lain.

4.2 OPSI BARIS PERINTAH

Ini adalah program yang sangat mudah dikonfigurasi dengan banyak fitur dan parameter yang dapat diubah. Selain jalur file input dan output, ArgumentParser kita perlu menangani semua opsi baris perintah dalam Tabel 4-1.

Tabel 4-1: Opsi Baris Perintah untuk Impressionist

Option	Extended	Kemungkinan	Default	Keterangan
`-t`	`--trials`	Integer	10000	Jumlah uji coba yang harus dijalankan
`-m`	`--method`	'random', 'average', 'common’	‘average’	Metode untuk menentukan warna bentuk
`-s`	`--shape`	'ellipse', 'triangle', 'quadrilateral', 'line'	‘ellipse’	Tipe bentuk yang digunakan
`-l`	`--length`	Integer	256	Panjang (tinggi) gambar akhir dalam piksel.
`-v`	`--vector`	Boolean	False	Buat keluaran vektor?

-a	--animate	Integer	0	Jika angka yang diberikan lebih besar dari 0, maka akan dibuat GIF animasi dengan jumlah milidetik per bingkai yang diberikan, yang menunjukkan gambar tersebut dibangun satu bentuk pada satu waktu.

File utama kita hanyalah kodifikasi dari Tabel 4-1. Semua opsi diteruskan ke konstruktor kelas `Impressionist`, yang akan kita bahas sebentar lagi.

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

if __name__ == "__main__":
  # Parse the file argument
  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)
```

Opsi -v menginstruksikan program untuk mengeluarkan hasilnya dalam format vektor. Dalam implementasi kita, kita akan menargetkan file SVG. Sebelum kita masuk ke algoritma utama aplikasi, mari kita sedikit menyimpang untuk melihat bagaimana kita dapat mendukung fitur ini.

4.3 FORMAT SVG

SVG adalah singkatan dari Scalable Vector Graphics. Ini adalah format berbasis XML untuk menentukan gambar vektor. Semua browser web utama modern dan program menggambar vektor mendukungnya. Alih-alih menggunakan pustaka pihak ketiga untuk menulis ke SVG, kita akan menulis kelas pendek kita sendiri untuk melakukannya. Spesifikasi SVG besar, tetapi kita hanya membutuhkan sebagian kecilnya untuk mendukung bentuk yang akan dikeluarkan program kita, sehingga tugasnya akan relatif mudah.

XML adalah format berbasis teks, dan karena kita mengeluarkannya, bukan menguraikannya, kita bahkan tidak perlu program kita untuk benar-benar memahami struktur XML. Kita hanya perlu menggabungkan string dari string lain yang mewakili elemen XML penyusunnya. Meskipun pendekatan ini membatasi kemampuan pengujian dan modularitas penulis SVG kami, jumlah standar SVG yang kami implementasikan sangat kecil sehingga hampir mudah untuk diperiksa kebenarannya secara manual.

Namun demikian, pendekatan ini tidak cocok untuk produksi.

Sebelum kita masuk ke kode, berikut contoh file SVG sederhana yang dapat dihasilkan program kami. File ini hanya memiliki satu segitiga yang dibangun menggunakan elemen poligon, ditambah persegi panjang latar belakang (saya sedikit memperbaiki formatnya dengan beberapa indentasi agar lebih mudah dibaca):

```
<?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>
```

Jika Anda menyimpan kode ini dalam file teks dengan ekstensi `.svg`, Anda dapat membukanya menggunakan peramban web atau editor gambar vektor untuk melihat segitiga yang dihasilkan. Saat kita membahas pembuatan kelas SVG kita, ingatlah contoh ini untuk memvisualisasikan bagaimana berbagai elemen bergabung untuk membentuk file SVG yang lengkap.

File SVG dimulai dengan deklarasi bahwa itu adalah file XML, dan kemudian elemen pertama adalah elemen svg yang menjelaskan versi spesifikasi SVG dan lebar serta tinggi gambar. Selain itu, setiap gambar yang dihasilkan program kita didukung oleh persegi panjang besar yang berisi warna rata-rata gambar. Ini membantu algoritma untuk memadukan warna dengan lebih baik. Oleh karena itu, file SVG kita juga dimulai dengan elemen rect besar:

```
# 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}" />'
```

Seperti yang ditunjukkan oleh properti background_color dalam konstruktor, warna direpresentasikan menggunakan tuple yang terdiri dari tiga bilangan bulat. Ini adalah kode warna RGB. Setiap elemen adalah bilangan bulat antara 0 dan 255 yang mewakili jumlah warna primer masing-masing (merah, hijau, atau biru) dalam output.

Misalnya, warna merah "murni" adalah (255, 0, 0), dan warna ungu adalah sesuatu seperti (128, 0, 128), yang merupakan campuran merah dan biru. Menggambar tiga jenis bentuk yang didukung program kami (elips, garis, dan poligon) hanyalah masalah memasukkan elemen SVG elips, garis, atau poligon masing-masing ke dalam file teks output:

```
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'
```

Terakhir, untuk menghasilkan file SVG, kita menutup elemen svg yang dimulai di dalam konstruktor dan menulis string yang telah digabungkan ke disk:

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

Jika Anda melihat spesifikasi resmi SVG, Anda mungkin merasa kewalahan, tetapi tidak perlu takut. Seperti yang diharapkan bagian ini tunjukkan, tidak perlu banyak usaha untuk mendapatkan nilai dari standar besar seperti SVG. Hanya dalam 20 baris kode, kita telah menulis pembuat SVG yang sangat terbatas, tetapi bermanfaat.

4.4 ALGORITMA STOCHASTIC

Algoritma yang menghasilkan kesan abstrak foto yang (terkadang) indah ini sangat sederhana. Singkatnya, algoritma ini mencoba menggambar bentuk dengan ukuran dan posisi acak, satu bentuk pada satu waktu. Jika bentuk yang ditambahkan membuat gambar abstrak

terlihat lebih mirip dengan foto aslinya, maka bentuk tersebut dipertahankan. Peningkatan ini berpotensi disempurnakan lebih lanjut dengan mengubah ukuran bentuk, yang merupakan masalah memindahkan setiap titiknya. Jika bentuk yang ditambahkan membuat gambar terlihat kurang mirip dengan foto aslinya, maka bentuk tersebut dibuang dan bentuk baru dicoba.

Berikut penjelasan algoritma yang lebih detail dalam beberapa langkah:

1. Buat kanvas kosong dengan ukuran yang sama dengan foto asli dan warna latar belakang yang sama dengan warna rata-rata foto asli.
2. Coba gambar bentuk pada kanvas di lokasi acak dan dengan ukuran acak. Warnai bentuk tersebut menggunakan warna rata-rata dari wilayah yang sesuai pada foto asli, warna yang paling umum di wilayah tersebut, atau secara acak.
3. Bandingkan warna piksel kanvas (dengan bentuk yang ditambahkan) dengan foto asli. Jika bentuk yang ditambahkan telah membuat piksel seluruh kanvas lebih mirip dengan piksel foto asli, maka pertahankan bentuk yang ditambahkan.
4. Coba modifikasi bentuk pada setiap titik (memperluas atau mempersempit) satu piksel pada satu waktu. Terus gerakkan titik-titik tersebut ke arah yang semakin mengurangi perbedaan antara piksel seluruh kanvas dan piksel foto asli. Berhenti ketika pergerakan tersebut tidak lagi meningkatkan perbedaan.
5. Ulangi langkah 2, 3, dan 4 beberapa kali.
6. Keluarkan gambar akhir yang dibuat di kanvas setelah sejumlah percobaan.

Ada banyak parameter yang dapat dikonfigurasi dari algoritma ini. Bentuk seperti apa yang harus digunakan? Berapa banyak percobaan yang harus dijalankan? Bagaimana warna untuk setiap bentuk harus dipilih? Dan ada beberapa submasalah yang harus dipecahkan. Bagaimana perbedaan antara dua gambar dihitung? Bagaimana Anda menemukan piksel di wilayah yang mencakup suatu bentuk?

Implementasi Utama

Tidak termasuk komentar, implementasi utama algoritma melukis kita kurang dari 150 baris Python. Sebagian besar keringkasan itu berkat pustaka Pillow yang hebat, yang telah dibahas di Bab 3. Pillow menangani pembacaan dan penulisan berbagai format gambar bitmap. Ia juga memiliki fasilitas untuk menggambar primitif sederhana seperti bentuk yang kita butuhkan. Terakhir, Pillow memiliki fungsi untuk menghitung perbedaan antara gambar dan menghitung warna rata-rata di suatu wilayah gambar. Ini akan menjadi fungsi pembantu yang penting untuk program kita, memungkinkan kita untuk berkonsentrasi pada algoritma inti sambil menyerahkan pekerjaan yang rumit kepada Pillow. Itulah yang memungkinkan oleh pustaka yang hebat.

Pengaturan

Kita mulai dengan beberapa impor dasar, definisi beberapa tipe yang dibutuhkan, sebuah konstanta, dan sebuah fungsi pembantu:

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]
```

Enum `ColorMethod` mengontrol bagaimana kita akan menghitung warna di suatu wilayah—yaitu, warna apa yang akan digunakan untuk mengisi suatu bentuk. Enum `ShapeType` mengatur bentuk yang akan kita gambar. Versi program saat ini hanya menggambar satu jenis bentuk di setiap lukisan, tetapi akan mudah untuk memodifikasi kode untuk memungkinkan lebih dari satu jenis bentuk. penulis telah menyerahkan itu untuk latihan.

Tipe `CoordList` berlaku untuk koordinat yang mendefinisikan satu bentuk.

Saat menjalankan algoritma, untuk kinerja, kita perlu bekerja dengan jumlah piksel total yang terbatas. Cara termudah untuk mencapai ini adalah dengan menskalakan gambar input jika lebih tinggi dari `MAX_HEIGHT`. Dengan kata lain, `MAX_HEIGHT` adalah tinggi maksimum gambar yang diskalakan. Perhatikan bahwa, secara teknis, kita juga harus mendefinisikan lebar maksimum, tetapi dalam praktiknya, sangat jarang rasio aspek gambar sedemikian rupa sehingga hanya membatasi satu dimensi saja tidak cukup (tidak banyak gambar yang sangat lebar tetapi memiliki tinggi yang sangat sedikit). Untuk menyederhanakan, kita hanya mendefinisikan satu dimensi maksimum.

Metode `get_most_common_color()` menentukan warna yang paling sering muncul dalam sebuah gambar. Metode ini menggunakan metode Pillow, `getcolors()`, yang mengembalikan semua warna dalam sebuah gambar beserta jumlahnya. Kemudian, ia menggunakan fungsi `max()` bawaan Python untuk mengekstrak yang paling sering muncul.

Konstruktor kelas Impressionist bertanggung jawab untuk mengatur parameter unik dari suatu eksekusi algoritma tertentu, membuka file gambar input, menskalakannya, membuat latar belakang awal gambar output, memanggil metode untuk menjalankan iterasi algoritma yang sebenarnya, dan memanggil metode untuk menghasilkan file akhir. Itu mungkin terdengar banyak, tetapi inti dari algoritma ada di metode lain. Konstruktor hanyalah titik awal yang memanggil metode dari pustaka Pillow dan metode lain yang akan kita bahas sebentar lagi untuk melakukan pekerjaan sebenarnya. Berikut adalah awal dari konstruktor:

```
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 = []
    # Open image file and store in instance variable, execute algorithm
    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)
```

Konstruktor dimulai dengan mengatur beberapa parameter dan menskalakan gambar masukan. Lukisan yang dihasilkan harus memiliki rasio aspek yang sama dengan gambar asli, sehingga rasio aspek dipertahankan. Metode `thumbnail()` Pillow adalah cara yang mudah untuk melakukan penskalaan. Berikut bagian selanjutnya dari konstruktor:

```
  # Start the generated image with a background that is the
  # average of all the original's pixels in color
  average_color = tuple((round(n) for n in ImageStat.Stat(self.original).mean))
  self.glass = Image.new("RGB", new_size, average_color)
```

Modul Pillow `ImageStat` dapat digunakan untuk menemukan warna rata-rata dalam sebuah gambar. Modul ini melihat nilai RGB dari setiap piksel dalam gambar dan merata-ratakan komponen merah, hijau, dan biru secara terpisah. Kita mengambil warna rata-rata yang dihasilkan dan menetapkannya sebagai latar belakang gambar kerja algoritma kita (`self.glass`). Dengan kata lain, warna rata-rata dari gambar asli akan menjadi warna awal setiap piksel dalam gambar kerja.

Variabel untuk gambar kerja diberi nama glass karena awalnya penulis menamai program ini Stained Glass. Setelah mengubah judulnya, penulis masih merasa bahwa nama glass untuk variabel tersebut menjelaskan bahwa ini adalah permukaan yang memberikan kesan yang difilter dari aslinya.

Konstruktor berlanjut:

```
# Keep track of how far along we are, our best result so far, and
# how much time elapses as the processing takes place
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)
```

Inti dari algoritma ini terletak pada metode `trial()`, yang mencoba menggambar sebuah bentuk untuk melihat apakah bentuk tersebut meningkatkan skor kemiripan antara gambar kerja dan gambar asli. Di sini, `trial()` dipanggil sebanyak beberapa kali. Saat percobaan dijalankan, kita melacak seberapa dekat kita dengan penyelesaian dan berapa lama waktu yang dibutuhkan program. Akhirnya, gambar kerja yang telah selesai dikeluarkan dengan bantuan `create_output()`.

Metode Utilitas

Sebelum kita sampai ke `trial()`, kita membutuhkan beberapa metode pembantu. Bagian penting dari algoritma melukis adalah memverifikasi bahwa setiap bentuk tambahan membawa gambar kerja lebih dekat ke gambar asli. Metode `difference()` menghitung skor kemiripan untuk dua gambar, mengukur seberapa mirip keduanya satu sama lain:

```
  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
```

Modul `ImageChops` dari Pillow memiliki metode `difference()` bawaan. Metode ini menemukan perbedaan pada tingkat piksel demi piksel antara dua gambar. Dengan kata lain, bagaimana dua piksel di lokasi yang sama dalam dua gambar berbeda satu sama lain? Perbedaannya hanyalah nilai absolut dari pengurangan setiap saluran warna di setiap piksel. Misalnya, perbedaan antara piksel RGB yang berwarna (`10, 100, 50`) dan piksel lain yang berwarna (`10, 40, 20`) adalah (`0, 60, 30`).

Namun, ini tidak cukup untuk algoritma kita. Kita membutuhkan satu angka, sebuah skor, yang menyatakan seberapa mirip dua gambar tersebut. Setelah menemukan perbedaan piksel demi piksel, kita dapat mengompresnya menjadi satu angka dengan merata-ratakan semua perbedaan tersebut.

Kita melakukan ini menggunakan modul ImageStat yang sama yang melakukan perhitungan rata-rata untuk kita guna menemukan warna rata-rata di konstruktor. Terakhir, meskipun tidak sepenuhnya diperlukan (rata-rata piksel dapat digunakan sebagai skor), kita membagi dengan perbedaan maksimum yang mungkin untuk mendapatkan skor sebagai rasio. Setiap kali kita menghasilkan bentuk baru, bentuk tersebut ditempatkan di lokasi acak di layar. Kita menghitung koordinat acak ini menggunakan `random_coordinates()`:

```
  def random_coordinates(self) -> CoordList:
    num_coordinates = 4 # ellipse or line
    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: # x coordinates
            coordinates.append(random.randint(0, self.original.width))
        else: # y coordinates
            coordinates.append(random.randint(0, self.original.height))
    return coordinates
```

Berbagai jenis bentuk membutuhkan jumlah koordinat yang berbeda. Misalnya, segitiga memiliki enam koordinat karena memiliki tiga titik, dan setiap titik memiliki satu koordinat x dan satu koordinat y. Koordinat harus valid—yaitu, harus berada di suatu tempat di permukaan gambar. Metode ini memastikan hal tersebut dengan memastikan bahwa koordinat acak tidak boleh kurang dari 0 atau lebih dari lebar atau tinggi gambar.

Kita juga membutuhkan cara untuk melihat "wilayah" dari foto asli yang sesuai dengan bentuk dalam gambar kerja, sehingga kita dapat menganalisis warna wilayah tersebut. Akan sangat mahal secara komputasi untuk menemukan piksel yang tepat di bawah bentuk sembarang. Sebagai gantinya, kita akan menggunakan metode statis `bounding_box()` untuk mengidentifikasi wilayah persegi panjang yang mencakup bentuk tersebut:

```
@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
```

Kotak pembatas adalah persegi panjang yang sejajar dengan sumbu (artinya tepinya sejajar dengan tepi gambar) di sekitar bentuk tertentu, yang ditentukan berdasarkan koordinat x dan y minimum dan maksimum bentuk tersebut. Kita akan meneruskan persegi panjang tersebut ke metode `crop()` bawaan Pillow untuk memotong gambar asli hingga hanya wilayah yang diinginkan. Kita akan meninggalkan teknik alternatif untuk mengekstrak wilayah yang lebih sempit dari gambar asli untuk latihan.

Percobaan

Inti dari algoritma ini adalah metode `trial()`. Setiap percobaan adalah upaya untuk menempatkan satu bentuk dalam gambar kerja. Jika bentuk baru tersebut mendekatkan gambar kerja ke gambar asli, maka bentuk tersebut dipertahankan. Jika skor perbedaan dapat ditingkatkan lebih lanjut dengan menggeser koordinatnya, maka koordinat bentuk tersebut digeser. Metode ini dimulai dengan menemukan tempat untuk bentuk baru menggunakan `random_coordinates()` dan menemukan wilayah pendukung dari koordinat tersebut:

```
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
```

Terdapat perulangan while yang kurang elegan di sini untuk memperhitungkan skenario yang tidak mungkin terjadi di mana koordinat acak semuanya sejajar di sepanjang salah satu sumbu. Dalam kasus tersebut, kita perlu menghasilkan ulang koordinat. Terdapat latihan di akhir bab untuk menghilangkan perulangan ini. Bagian selanjutnya dari metode ini memilih warna untuk bentuknya:

```
  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: # must be random
    color = tuple(random.choices(range(256), k=3))
  original = self.glass
```

Bergantung pada `ColorMethod`, kita memilih warna rata-rata di wilayah latar belakang (sekali lagi menggunakan `ImageStat`), memilih warna yang paling umum di wilayah latar belakang, atau cukup memilih warna acak. Kemudian, kita menyimpan keadaan gambar kerja saat ini (`self.glass`) dalam variabel lokal, original, untuk digunakan kembali jika dilakukan pergeseran koordinat (kita mencoba menggambar ulang bentuk sedikit lebih besar atau sedikit lebih kecil ke berbagai arah, jadi kita membutuhkan kanvas asli tempat bentuk itu digambar). Sekarang kita siap untuk mencoba menggambar bentuk:

```
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: # must be triangle or quadrilateral or line
    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
```

Fungsi internal, `experiment()`, mengembalikan True jika upaya untuk menggambar bentuk baru berhasil dalam hal menurunkan perbedaan antara gambar kerja dan gambar asli. Modul `ImageDraw` di Pillow menangani penggambaran sebenarnya. Perbedaan dihitung

menggunakan metode `difference()` yang telah didefinisikan sebelumnya dan dibandingkan dengan perbedaan terbaik yang ditemukan sejauh ini. Jika bentuk tersebut telah meningkatkan gambar, gambar kerja diganti dengan gambar yang menyertakan bentuk baru.

Bagian terakhir dari `trial()` mencoba membuat peningkatan bertahap pada setiap bentuk dengan menggeser koordinatnya. Jika penggeseran meningkatkan skor perbedaan dibandingkan dengan versi gambar kerja dengan koordinat asli bentuk tersebut, maka penggeseran dipertahankan dan penggeseran lain dicoba ke arah yang sama:

```
if experiment():
   # Try expanding every direction, keep going in better directions
   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))
```

Kode ini adalah semacam algoritma pendakian bukit (hill climbing), di mana kita terus bergerak ke arah yang sama untuk menyelesaikan suatu masalah (dalam hal ini, mengoptimalkan perbedaan) selama solusi terus membaik. Kita berhenti ketika solusi tersebut berhenti membaik. Ini mungkin menyebabkan maksimum lokal, tetapi ini adalah cara yang sederhana dan efektif untuk meningkatkan solusi yang sudah ada.

Dalam hal ini, kita memiliki solusi yang sudah ada karena kita hanya menyimpan bentuk-bentuk yang awalnya memperbaiki perbedaan (ditunjukkan oleh experiment() yang mengembalikan True). Lihat kotak "*Pendakian Bukit*" untuk informasi lebih lanjut tentang cara kerja algoritma jenis ini.

Algoritma secara keseluruhan akan bekerja tanpa proses penyesuaian (nudging), tetapi penyesuaian tersebut meningkatkan kesesuaian setiap bentuk. Hal ini, pada gilirannya, meningkatkan tampilan keseluruhan lukisan akhir dan mengurangi jumlah bentuk yang diperlukan untuk mendapatkan hasil yang wajar.

Setelah bentuk akhir ditetapkan setelah penyesuaian apa pun, kita menambahkan koordinat dan warnanya ke daftar bentuk. Mempertahankan daftar ini, terpisah dari menggambar bentuk-bentuk dalam gambar kerja, diperlukan untuk menghasilkan output akhir.

PENDAKIAN BUKIT

Pendakian bukit adalah teknik optimasi sederhana yang bertujuan untuk menemukan nilai maksimum atau minimum suatu fungsi dengan terus bergerak ke arah yang sama sementara pencarian tampaknya "meningkat". Dalam penjelasan klasik teknik ini, Anda diminta untuk membayangkan bahwa Anda mengenakan penutup mata dan berdiri di dasar bukit yang ingin Anda daki. Anda dapat merasakan dengan kaki Anda kemiringan tanah di sekitar Anda. Dengan setiap langkah, arah mana pun yang tampaknya menghasilkan kemiringan ke atas yang paling curam akan terasa seperti jalan yang harus Anda tempuh jika Anda ingin mendaki bukit secepat mungkin. Anda dapat terus memilih untuk naik ke arah ini selama Anda dapat merasakan dengan kaki Anda bahwa Anda sedang mendaki. Akhirnya Anda akan mencapai titik di mana Anda tidak lagi mendaki, terlepas dari arah langkah Anda selanjutnya, dan kemudian Anda dapat berhenti.

Apakah Anda akan mencapai puncak bukit? Tentu saja mungkin, terutama jika bukit tersebut memiliki satu puncak. Tetapi mungkin juga bukit tersebut memiliki beberapa puncak dan Anda baru saja mencapai salah satu puncak yang lebih kecil. Itu disebut terjebak dalam maksimum lokal. Hill climbing akan selalu menemukan maksimum lokal, tetapi mungkin tidak menemukan maksimum global.

Hill climbing adalah teknik populer dalam kecerdasan buatan karena sangat sederhana. Ini adalah titik awal yang baik untuk banyak masalah. Dalam program kami, kami terus menggeser koordinat ke arah yang sama sampai perbedaan dengan gambar awal tidak lagi membaik. Ini adalah jenis hill climbing: kita terus bergerak ke arah yang sama sampai keadaan tidak membaik. Tentu saja, mungkin saja penempatan bentuk tersebut adalah kesalahan sejak awal dibandingkan dengan beberapa penempatan alternatif lainnya, dan penggeseran kita hanya membawa kita ke dalam lubang kelinci menuju maksimum lokal. Dengan algoritma yang begitu sederhana, tidak ada cara untuk mengetahuinya dengan pasti.

4.5 PENGGUNAAN ALGORITMA MELUKIS STOKASTIK

Gambar kerja diskalakan ke `MAX_HEIGHT`, tetapi gambar keluaran akhir harus memiliki tinggi yang ditentukan pengguna (sekali lagi, untuk penyederhanaan, kami membiarkan pengguna hanya mengatur tinggi dan bukan lebar). Kita tidak bisa begitu saja "meregangkan" bitmap tanpa pikselasi.

Sebagai gantinya, kita menggambar ulang gambar kerja menggunakan data dalam daftar bentuk, dengan setiap bentuk diskalakan dengan tepat. Metode untuk menghasilkan gambar juga menggabungkan opsi untuk menghasilkan file vektor (menggunakan kelas SVG dari sebelumnya) dan menghasilkan GIF animasi melalui Pillow. Ini secara signifikan menambah panjangnya. Berikut adalah awal dari 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)
```

Kita mulai dengan membuat gambar baru dengan ukuran yang sesuai berdasarkan parameter tinggi yang ditentukan pengguna. Kita mengisi gambar keluaran awal dengan warna rata-rata dari gambar asli seperti yang dilakukan pada gambar kerja. Metode ini berlanjut:

```
        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]
```

Gambar keluaran akan dihasilkan dengan mereproduksi setiap bentuk dalam daftar bentuk secara iteratif pada skala yang tepat ❶. Untuk juga membuat keluaran SVG atau GIF animasi, saat gambar keluaran dihasilkan, setiap langkah akan diulangi pada objek svg atau disalin sebagai gambar ke daftar animation_frames yang membentuk "film" GIF animasi:

```
        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: # must be triangle or quadrilateral or line
            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)
```

Sisa metode ini hanyalah menggambar bentuk pada gambar keluaran dan menulis file ke disk.

Hasilnya

Ada banyak hal yang terkandung dalam 150 baris kode tersebut. Program ini menampilkan percobaan stokastik, beberapa wawasan tentang cara membuat tebakan yang tepat tentang warna setiap bentuk, sedikit pendakian bukit, dan penggunaan pustaka yang bagus. Hasil keren dalam ilmu komputer lebih banyak tentang algoritma dan teknik daripada jumlah baris kode. Tetapi algoritma ini juga sangat sederhana—namun sangat efektif. Tidak, outputnya tidak seimpresif jaringan saraf terbaru, tetapi sungguh menakjubkan seberapa jauh teknik sederhana dapat membawa sebuah program.

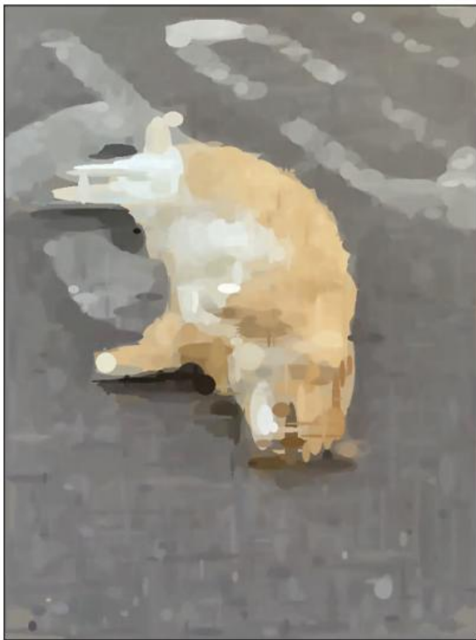
Gambar 4-7: Touro Park dengan 19.578 jalur

Kelemahan utama dari algoritma ini adalah lambat dan acak. Anda dapat mencoba gambar yang sama beberapa kali dengan parameter yang sama dan mendapatkan hasil yang berbeda. Dan Anda mungkin menunggu lama untuk mendapatkan hasil yang bervariasi, terkadang buruk. Gambar 4-8 adalah tampilan lebih dekat dari Menara Newport.

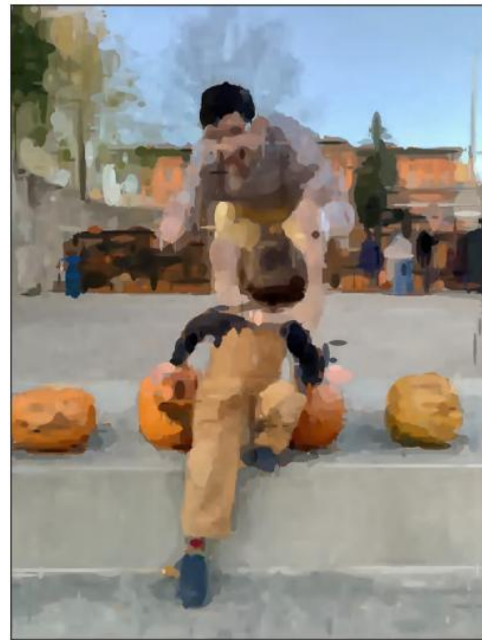
Gambar 4-8: Menara Newport dengan 11.409 jalur

Namun, penulis memiliki beberapa hasil yang mengesankan, meskipun diakui dipilih secara selektif, untuk dibagikan kepada Anda. Yang pertama adalah beberapa pemandangan dari Touro Park di Newport, Rhode Island. Penulis suka bagaimana bentuk garisnya memberikan nuansa seperti lukisan cat minyak pada masing-masing gambar. Gambar 4-7 adalah pemandangan luas taman dengan Menara Newport yang terkenal di sebelah kanan.

Gambar 4-9 menunjukkan seekor kucing yang penulis temukan berguling-guling di trotoar. Bentuk elips memberikan tampilan abstrak yang bagus pada kucing tersebut. Mungkinkah ini karya seorang pelukis impresionis?



Gambar 4-9: Seekor kucing berguling di trotoar dengan elips



Gambar 4-10: Adegan Halloween dengan elips

Terakhir, penulis menyajikan adegan dari Halloween pada Gambar 4-10. Saya suka bagaimana labu dan orang-orang di latar belakang digambarkan menggunakan elips.

KODE BERTEMU KEHIDUPAN

Pada pertengahan tahun 2010-an, saya pertama kali berpikir untuk membuat program seperti yang ada di bab ini menggunakan algoritma genetika. Saya melakukan sedikit riset dan menemukan bahwa beberapa orang telah mendahului saya. Namun, dalam melakukan riset tersebut, saya juga menemukan proyek Primitive karya Michael Fogleman. Ia telah menciptakan program yang menghasilkan seni abstrak, seperti program-program lama yang menggunakan algoritma genetika, tetapi menggunakan teknik yang lebih sederhana yang disebut simulated annealing.

Saya ingin meluangkan waktu sejenak untuk berterima kasih kepada Michael atas pengaruhnya yang besar terhadap karier pemrograman saya. Michael adalah programmer yang sangat berbakat, tetapi selain berbakat, ia juga menulis kode yang sangat mudah dibaca. Dan kebetulan ia menciptakan proyek di banyak bidang yang menarik minat saya.

Meskipun proyek ini tidak memiliki kode yang sama dengan Primitive karya Michael, fakta bahwa ia dapat mengubah foto menjadi seni abstrak menggunakan algoritma yang begitu sederhana mendorong saya untuk percaya bahwa saya dapat melakukan hal yang sama menggunakan teknik saya sendiri yang bahkan lebih sederhana. Saya mencoba mengimplementasikan algoritma saya sebagai aplikasi iOS. Saya sebagian besar berhasil, tetapi sayangnya, iPhone era 2017 tidak cukup cepat untuk menjalankan program saya dalam waktu yang wajar.

Saya mencoba mengoptimalkannya, tetapi masalahnya adalah algoritma saya, bukan implementasinya. Pada saat yang sama, aplikasi berbasis pembelajaran mesin yang menarik untuk transformasi foto artistik mulai muncul untuk iOS, dan saya menyadari teknik saya yang lebih lambat dan sederhana tidak dapat bersaing. Namun, itu tetap menjadi demo yang keren, dan ketika saya membuat proyek untuk buku ini, saya mengingatnya. Saya pikir itu adalah ilustrasi yang bagus tentang kekuatan algoritma acak dan pendakian bukit.

Setelah saya memindahkan kode Swift saya ke Python untuk buku ini, saya memutuskan untuk mengujinya pada teman-teman saya dengan memposting di Facebook foto putra saya yang berusia satu tahun, Daniel, di ayunan.

Bibi saya, yang memiliki mata artistik yang cukup terlatih, mengira saya telah mulai melukis. Saat itulah saya tahu programnya cukup bagus.

Aplikasi di Dunia Nyata

Selain terlihat keren, output dari program ini tidak memiliki banyak aplikasi praktis. Namun, teknik yang digunakan untuk membangunnya tentu saja memiliki aplikasi praktis. Teknik-teknik tersebut termasuk dalam istilah umum yang dikenal sebagai optimasi stokastik. Misalkan Anda memiliki masalah optimasi yang ingin Anda selesaikan, tetapi Anda tidak

mengetahui algoritma deterministik (algoritma yang memberikan hasil yang sama setiap kali dengan mengikuti langkah-langkah yang sama setiap kali) untuk menyelesaikannya. Dalam hal ini, teknik yang melibatkan percobaan acak (`stokastik`) mungkin diperlukan.

Mungkin tidak begitu jelas, tetapi tantangan dalam bab ini adalah contoh masalah optimasi. Program kita mencoba mengoptimalkan gambar yang sedekat mungkin dengan foto aslinya. Fungsi objektif (hal yang memeriksa apakah kita berada di arah yang benar) adalah metode `difference().` Semakin rendah perbedaannya, semakin optimal solusi potensial untuk masalah yang diwakili oleh gambar tertentu.

Salah satu bidang praktis terkenal di mana algoritma optimasi stokastik berguna adalah masalah pedagang keliling klasik. Masalah ini mengharuskan seorang pelancong untuk mengunjungi setiap lokasi yang ditentukan pada peta tepat sekali dan kembali ke titik awal mereka menggunakan rute terpendek yang mungkin. Ini adalah apa yang dilakukan truk pengiriman (seperti FedEx atau UPS) setiap hari, jadi ini memiliki aplikasi yang sangat praktis. Sayangnya, tidak ada algoritma deterministik yang diketahui untuk menyelesaikan masalah pedagang keliling secara optimal untuk sejumlah besar lokasi dalam waktu yang wajar. Sebaliknya, teknik optimasi stokastik seperti algoritma genetika memberikan cara yang berguna untuk menyelesaikan masalah ini, tetapi solusinya mungkin suboptimal. Algoritma genetika mungkin tidak selalu menghasilkan solusi sempurna untuk masalah pedagang keliling, tetapi hampir selalu menghasilkan solusi yang cukup baik.

Program kami juga menggunakan pendakian bukit (hill climbing). Meskipun ini adalah salah satu prosedur pencarian lokal yang paling sederhana (cukup terus berjalan ke arah yang sama jika arah tersebut berhasil), ini adalah teknik yang sangat umum, dan kinerjanya sama baiknya dengan teknik yang lebih canggih dalam banyak skenario. Pendakian bukit juga menjadi dasar bagi algoritma lain yang lebih canggih. Misalnya, algoritma simpleks untuk menyelesaikan masalah pemrograman linier menggunakan metode hill climbing.

LATIHAN SOAL

1. Modifikasi program untuk menggambar lebih dari satu jenis bentuk dalam lukisan yang sama. Misalnya, program dapat membuat gambar dengan elips dan segitiga dalam output akhir.
2. Karena piksel yang digunakan untuk menghitung warna untuk bentuk baru didasarkan pada kotak pembatas dan bukan area tepat di bawah bentuk tersebut, hasilnya tidak akurat. Modifikasi `trial()` untuk bereksperimen tidak hanya dengan pergeseran koordinat tetapi juga pergeseran warna. Ini dapat menghasilkan warna yang lebih sesuai.
3. Modifikasi `trial()` untuk menggunakan piksel tepat di bawah suatu bentuk untuk menentukan warna bentuk tersebut. Ini menantang. Salah satu caranya adalah dengan menggunakan beberapa perhitungan geometris untuk menentukan piksel yang tepat untuk setiap jenis bentuk. Ini kemungkinan akan jauh kurang efisien secara komputasi daripada hanya memotong kotak pembatas seperti yang dilakukan program asli. Pertimbangkan untuk menggunakan fasilitas mask di Pillow sebagai gantinya.

4. Perulangan while True di awal `trial()` terasa seperti kode yang kurang baik. Tulis ulang bagian awal `trial()` tanpa perulangan tersebut.

BAB 5
MEMBANGUN MESIN VIRTUAL CHIP-8

Pada bab ini, kita akan mengembangkan versi mesin virtual yang dikenal sebagai CHIP-8, sebuah platform dari masa awal komputasi pribadi yang terutama digunakan untuk bermain game. Meskipun program kita akan dapat memainkan game CHIP-8, bukan game itu sendiri yang menarik minat kita—melainkan apa yang dapat kita pelajari tentang pemrograman tingkat rendah dan bagaimana komputer bekerja pada tingkat register dan instruksi dengan membangun mesin virtual CHIP-8. Wawasan ini menjadikan pembangunan mesin virtual CHIP-8 sebagai langkah pertama yang populer ke dunia pemrograman emulator.

5.1 MESIN VIRTUAL

Bayangkan mesin virtual (VM) sebagai komputer yang sepenuhnya didefinisikan dalam perangkat lunak. Program yang dirancang untuk berjalan di VM dapat berjalan di platform apa pun yang memiliki implementasi VM tersebut. Dengan cara ini, VM memungkinkan perangkat lunak yang benar-benar portabel.

VM (Virtual Machine) sangat terkait dengan emulator. Emulator adalah perangkat lunak yang berpura-pura menjadi perangkat keras. Ini memungkinkan program yang ditulis untuk perangkat keras tersebut untuk berjalan di mesin lain yang tidak memiliki perangkat keras tersebut. Emulator harus mengikuti spesifikasi perangkat keras asli dengan cermat sehingga dapat menciptakan kembali semua fungsi yang diharapkan oleh program yang berjalan di emulator tanpa menyadarinya. Saya katakan tanpa menyadarinya karena perangkat lunak yang berjalan di emulator tidak tahu bahwa ia tidak berjalan di perangkat keras yang sebenarnya; emulator harus bekerja persis seperti perangkat keras asli agar program dapat berfungsi dengan benar.

VM juga merupakan perangkat lunak yang mengikuti spesifikasi lingkungan tempat perangkat lunak berjalan. Perbedaannya adalah, sementara emulator mengikuti spesifikasi perangkat keras, VM mengikuti spesifikasi yang mungkin sepenuhnya didefinisikan sebagai abstraksi dalam istilah perangkat lunak.

Meskipun yang satu adalah spesifikasi perangkat keras dan yang lainnya adalah spesifikasi perangkat lunak, mengimplementasikan emulator sederhana cukup mirip dengan mengimplementasikan VM sederhana. Faktanya, keduanya sangat mirip sehingga meskipun proyek yang diselesaikan dalam bab ini secara teknis adalah proyek VM, proyek ini sangat umum disarankan sebagai proyek emulasi pertama. Jika Anda pendatang baru di komunitas pengembangan emulator dan bertanya dari mana harus memulai, CHIP-8 hampir selalu menjadi jawabannya.

Mungkin VM yang paling terkenal adalah Java Virtual Machine (JVM). Ketika Java pertama kali muncul pada pertengahan tahun 1990-an, filosofi "tulis sekali, jalankan di mana saja" dipuji-puji. JVM dikembangkan untuk semua sistem operasi utama (Windows, Linux, Mac OS, dan sebagainya), dan program Java yang sama dapat dikompilasi ke dalam format bytecode

asli JVM dan dijalankan di komputer mana pun dengan JVM tanpa perubahan, terlepas dari platform yang mendasarinya. Itu masih berlaku hingga saat ini, tetapi ceruk asli Java "tulis sekali, jalankan di mana saja" sebagian besar telah digantikan oleh aplikasi web.

VM CHIP-8 berasal dari era yang jauh lebih awal. Pada tahun 1970-an, Joseph Weisbecker adalah seorang insinyur perintis yang mengembangkan salah satu mikroprosesor 8-bit pertama, RCA 1802. Ia dan RCA membangun komputer pribadi awal menggunakan penemuannya. Ia ingin memiliki cara untuk memprogram game untuk mesin tersebut dalam bahasa tingkat tinggi daripada kode mesin, jadi ia mengembangkan CHIP-8 (dan bahasa opcode yang menyertainya). Putrinya, Joyce Weisbecker, kemudian menggunakan CHIP-8 untuk menjadi pengembang video game wanita pertama yang karyanya diterbitkan. Pada tahun 1980-an, CHIP-8 diporting ke banyak platform lain, termasuk banyak kalkulator grafik. Oleh karena itu, ia menjadi VM yang benar-benar portabel, analog dengan bentuk awal bagaimana kita memikirkan VM saat ini.

5.2 MESIN VIRTUAL CHIP-8

Mesin Virtual CHIP-8 awalnya dirancang untuk komputer pribadi dengan keterbatasan sumber daya yang luar biasa pada akhir tahun 1970-an, seperti COSMAC VIP. Dirilis pada tahun 1977, COSMAC VIP memiliki mikroprosesor 8-bit RCA 1802 yang berjalan kurang dari 2 megahertz (MHz), RAM 2KB (dapat diperluas hingga 4KB), dan ROM 512-byte. Ia juga memiliki chip khusus untuk menampilkan grafik 1-bit dengan resolusi hingga 64×128, membaca dan menulis kaset, dan memainkan suara bip.

Sungguh menakjubkan menurut standar saat ini bahwa sesuatu yang berharga dapat diprogram pada mesin seperti COSMAC VIP, namun mesin ini dirancang untuk permainan video. Bahkan, permainan tersebut dijalankan melalui lapisan abstraksi lain, yaitu Mesin Virtual CHIP-8. Konsol video game paling populer pada era itu, Atari 2600, juga dirilis pada tahun 1977 dan memiliki spesifikasi yang hampir sama. Keterbatasan ini hanyalah hal yang wajar.

Saat memprogram VM atau emulator, kinerja alat yang Anda gunakan adalah perhatian utama. VM atau emulator menambahkan lapisan abstraksi lain antara program dan perangkat keras, dan setiap lapisan abstraksi umumnya disertai dengan biaya kinerja. Untuk mencapai kecepatan yang diinginkan dari sistem asli, overhead harus dijaga seminimal mungkin, dan beberapa bahasa pemrograman (atau lebih tepatnya, beberapa implementasi runtime utama dari bahasa pemrograman) menjadi penghalang. Inilah sebabnya mengapa umum untuk melihat VM dan emulator diprogram dalam bahasa tingkat rendah seperti C, C++, dan Rust.

Meskipun demikian, mengingat betapa terbatasnya perangkat keras target asli CHIP-8, tidak sulit untuk membuat VM CHIP-8 yang berkinerja baik saat ini pada sistem modern apa pun. Bahkan runtime bahasa pemrograman yang relatif lambat seperti CPython sudah cukup. Anda tentu tidak ingin memprogram emulator konsol game canggih menggunakan Python, atau JVM. Tetapi CHIP-8? Python lebih dari cukup untuk itu.

Untuk memahami CHIP-8, mari kita mulai dengan membahas register dan tata letak memorinya. Kemudian, penulis akan memberikan gambaran umum tentang instruksi yang dapat dieksekusi oleh VM, sebelum masuk ke detail implementasi yang lebih rumit.

Register dan Memori

Pada mikroprosesor fisik, register adalah memori tercepat yang tersedia. Register berada langsung di dalam mikroprosesor dan tidak memerlukan latensi untuk mengakses chip lain. Menempatkan data di register seringkali merupakan satu-satunya cara untuk memanipulasinya, karena sebagian besar instruksi manipulasi data (misalnya, aritmatika) yang didukung oleh mikroprosesor beroperasi pada data di dalam register. Instruksi muat/simpan terpisah mentransfer data antara register dan RAM eksternal.

Dalam hal register, terdapat pertimbangan klasik antara waktu dan ruang: register adalah lokasi penyimpanan tercepat untuk menyimpan data, tetapi ukurannya sangat terbatas. Misalnya, mikroprosesor 8-bit tipikal pada akhir tahun 1970-an mungkin hanya memiliki beberapa register 8-bit (ya, masing-masing hanya dapat menyimpan satu byte), tetapi dapat mengakses puluhan kilobyte RAM eksternal.

Sebagian besar VM, seperti CHIP-8, juga memiliki register, tetapi register tersebut tidak selalu dipetakan langsung ke register perangkat keras fisik pada mikroprosesor. Karena itu, register tersebut belum tentu lebih cepat daripada RAM. Hal itu mungkin tampak aneh, tetapi register menyediakan substrat tempat instruksi dapat beroperasi.

Tidak ada juga yang menghalangi implementasi VM tertentu untuk memetakan register virtual ke register perangkat keras nyata untuk peningkatan kinerja—selama jumlah register virtual tidak melebihi jumlah register fisik.

Dalam pembahasan berikut, nama yang sama digunakan untuk merujuk pada register CHIP-8 seperti yang akan digunakan dalam kode Python untuk implementasinya. VM CHIP-8 memiliki 16 register 8-bit serbaguna, yang disebut sebagai `v[0]` hingga `v[15]`. Register ini dapat digunakan untuk semua jenis data, dan semua instruksi aritmatika dan logika utama beroperasi pada register ini. Dari register serbaguna ini, `v[15]` (atau `v[0xF]` dalam heksadesimal) istimewa karena digunakan untuk menyimpan flag. Register indeks, `i`, digunakan untuk manipulasi di beberapa lokasi memori sekaligus dan untuk menunjukkan di mana data yang perlu digambar ke layar berada di memori. Penghitung program, pc, adalah register khusus yang melacak alamat memori dari instruksi berikutnya yang akan dieksekusi.

`Vs`, `i`, dan `pc` merupakan register utama, tetapi didukung oleh beberapa register semu untuk pengaturan waktu. Dua byte ini, `delay_timer` dan `sound_timer`, digunakan untuk menerapkan jeda dalam permainan atau menunjukkan berapa lama suara bip harus diputar. Ada instruksi khusus untuk memodifikasi timer ini. Semua register tercantum dalam Tabel 5-1. Register-register tersebut awalnya dijelaskan dalam Manual Instruksi RCA COSMAC VIP CDP18S711.

Tabel 5-1: Register dan Pseudo-Register CHIP-8

Register	Nama	Keterangan
`v[0] to v[14]`	Register tujuan umum	Masing-masing dapat menyimpan semua jenis data 8-bit.
`v[15]`	Daftar bendera	Menyimpan sebuah flag (1 atau 0) setelah operasi tertentu, seperti flag carry setelah penjumlahan.
`pc`	Penghitung program	Melacak alamat 16-bit dalam memori dari instruksi yang sedang dieksekusi.
`i`	Daftar indeks memori	Menyimpan alamat 16-bit yang digunakan untuk menyelesaikan instruksi yang mencakup beberapa tempat yang berdekatan di memori.
`delay_timer`	Pengatur waktu tunda	Menyimpan nilai 8-bit yang dikurangi 60 kali per detik hingga mencapai 0.
`sound_timer`	Pengatur waktu suara	Menyimpan nilai 8-bit yang dikurangi 60 kali per detik hingga mencapai 0; selama nilainya di atas 0, bunyi bip akan diputar oleh speaker komputer.

Sebuah VM CHIP-8 tipikal memiliki RAM serbaguna sebesar 4KB. Ini sejalan dengan COSMAC VIP ketika dimuat dengan memori ekspansi. Namun, ada kendalanya: pada VIP, 512 byte pertama memori harus berisi kode untuk VM CHIP-8 itu sendiri (ya, seluruh VM hanya muat dalam 512 byte kode mesin—bayangkan itu saat kita menulis versi kita). Itu hanya menyisakan 3,5KB RAM yang dapat digunakan. Agar kompatibel dengan versi sebelumnya saat ini, VM kita juga harus mencadangkan 512 byte pertama RAM.

Instruksi

VM CHIP-8 sebagian besar digunakan untuk memprogram game, jadi ia mencakup instruksi khusus untuk tindakan seperti memindahkan sprite dan memainkan bunyi bip. Instruksi-instruksi tersebut berada di samping semua instruksi umum dan fungsional yang akan Anda temukan di setiap set instruksi mikroprosesor atau bahasa pemrograman tingkat rendah—instruksi untuk memanipulasi memori, melakukan aritmatika, mengawasi aliran kontrol, menangani timer, dan mengelola tampilan. Secara total, ada 35 instruksi yang akan kita implementasikan. Semua instruksi ditentukan dalam heksadesimal— lihat kotak "Heksadesimal" untuk informasi lebih lanjut tentang sistem penomoran tersebut.

SISTEM ANGKA HEKSADESIMAL

Heksadesimal, atau basis-16, adalah sistem bilangan yang biasanya digunakan untuk bekerja dengan byte tingkat rendah pada sistem komputasi (alamat RAM, instruksi CPU, dan sejenisnya). Sistem ini dapat merujuk pada nilai dalam byte secara lebih ringkas dan konsisten daripada biner atau desimal standar (basis-10, sistem bilangan yang biasa kita gunakan). Misalnya, Anda dapat merepresentasikan bilangan 8-bit apa pun menggunakan dua digit

heksadesimal, dan yang lebih bermanfaat, setiap dari dua digit tersebut sesuai dengan tepat setengah dari byte ketika ditulis dalam biner (setengah dari byte dikenal sebagai nibble). Jika Anda seorang programmer pada tahun 1970-an atau 1980-an, Anda akan sering bekerja dengan heksadesimal, tetapi saat ini pengembang Python rata-rata jarang menggunakannya di luar pemrograman tingkat rendah.

Dalam heksadesimal, selain 10 simbol $0-9$, disediakan enam simbol tambahan, $A-F$, yang sesuai dengan nilai desimal $10-15$. Dalam Python, literal heksadesimal dimulai dengan awalan $0\mathrm{x}$. Misalnya, $0\mathrm{xFF}$ sama dengan angka desimal 255, atau angka biner $0b11111111$. Satu F dalam versi heksadesimal mengacu pada setengah pertama dari angka satu dalam versi biner (1111), dan F lainnya mengacu pada kumpulan angka satu kedua (1111). Ini adalah nilai maksimum 1 byte. Untuk mengilustrasikan konversi lebih jelas, angka heksadesimal $0\mathrm{xF0}$ dapat ditulis dalam biner sebagai $0b11110000$, dengan F untuk 1111 dan 0 untuk 0000.

Untuk mengkonversi dari heksadesimal ke desimal, kalikan setiap digit heksadesimal dari kanan ke kiri dengan pangkat 16, dimulai dengan 160. Misalnya, $0\mathrm{xFF}$ dapat ditulis ulang sebagai $(15 \times 160) + (15 \times 161)$. Angka kanan (F) menjadi $15 \times 1 = 15$, angka kiri menjadi $15 \times 16 = 240$, dan $240 + 15 = 255$. Berikut contoh lain: $0\mathrm{xA5B}$ adalah $(11 \times 160) + (5 \times 161) + (10 \times 162)$. Ini setara dengan 2.651 dalam desimal.

Instruksi-instruksi ini ada di sini sebagai referensi cepat dan untuk memberi Anda gambaran tentang "situasi umum". Kita akan membahas detail cara kerja setiap instruksi dalam kode, tetapi kenyataannya sebagian besar kode cukup mudah dipahami berdasarkan deskripsi instruksi. Sebagian besar instruksi dapat diimplementasikan hanya dalam beberapa baris Python.

Saya menghabiskan banyak waktu memikirkan cara mengelompokkan instruksi untuk diskusi ini. Pada akhirnya, penulis memutuskan untuk mengurutkannya secara numerik sehingga muncul dalam urutan yang sama di sini seperti dalam kode. Setiap instruksi dalam CHIP-8 adalah 16 bit, atau dengan kata lain, 2 byte atau 4 nibble, sehingga diterjemahkan menjadi empat digit heksadesimal. Setiap digit heksadesimal huruf besar 0–F dalam sebuah instruksi adalah literal. Setiap huruf kecil menunjukkan nilai yang akan digunakan sebagai bagian dari implementasi instruksi. Garis bawah (_) menunjukkan bahwa nibble bersifat arbitrer. Instruksi-instruksi ini awalnya dijelaskan dalam Manual Instruksi RCA COSMAC VIP CDP18S711.5

Beberapa instruksi yang tercantum di sini tidak ada dalam spesifikasi CHIP-8 asli (misalnya, 8x_6 dan 8x_E). Fungsionalitasnya terkadang berbeda di berbagai implementasi CHIP-8.

Pembersihan Layar dan Lompatan Dasar

Rangkaian instruksi pertama digunakan untuk membersihkan seluruh layar sekaligus dan untuk berpindah dari satu bagian program ke bagian program lainnya.

- **00E0** Bersihkan layar.
- **00EE** Kembali dari subrutin.
- **0nnn** Panggil program di nnn, atur ulang timer dan register, dan bersihkan layar.
- **1nnn** Lompat ke alamat nnn tanpa mengatur ulang.
- **2nnn** Panggil subrutin di nnn.

Lompatan Bersyarat

Rangkaian instruksi berikutnya adalah untuk lompatan ke bagian program lain jika kondisi tertentu benar.

- **3xnn** Lewati instruksi berikutnya jika v[x] sama dengan nn.
- **4xnn** Lewati instruksi berikutnya jika v[x] tidak sama dengan nn.
- **5xy_** Lewati instruksi berikutnya jika v[x] sama dengan v[y].

Penyesuaian Register Serbaguna, Aritmatika, dan Manipulasi Bit

Berikutnya adalah instruksi standar yang akan Anda temukan di CPU atau VM mana pun untuk tindakan seperti melakukan perhitungan matematika, mengatur register, dan menggeser bit.

- **6xnn** Atur v[x] menjadi nn.
- **7xnn** Tambahkan nn ke v[x].
- **8xy0** Atur v[x] menjadi v[y].
- **8xy1** Atur v[x] menjadi v[x] | v[y] (bitwise OR).
- **8xy2** Atur v[x] menjadi v[x] & v[y] (bitwise AND).
- **8xy3** Atur v[x] menjadi v[x] ^ v[y] (bitwise XOR).
- **8xy4** Tambahkan v[y] ke v[x] dan atur flag carry.
- **8xy5** Kurangi v[y] dari v[x] dan atur flag borrow.
- **8x_6** Geser v[x] ke kanan satu bit dan atur flag ke bit paling tidak signifikan.
- **8xy7** Kurangi v[x] dari v[y] dan simpan hasilnya di v[x]; atur flag borrow.
- **8x_E** Geser v[x] ke kiri satu bit dan atur flag ke bit paling signifikan.

Instruksi Lain-lain

Instruksi-instruksi ini tidak memiliki area subjek yang seragam, tetapi opcode-nya berdekatan secara numerik.

- **9xy0** Lewati instruksi berikutnya jika v[x] tidak sama dengan v[y].
- **Annn** Atur i ke nnn.
- **Bnnn** Lompat ke nnn + v[0].
- **Cxnn** Atur v[x] ke bilangan bulat acak (0–255) & nn (bitwise AND).
- **Dxyn** Gambar sprite setinggi n di (v[x], v[y]); atur flag saat terjadi tabrakan.

Instruksi Tombol dan Timer

Kumpulan instruksi berikutnya adalah untuk memanipulasi timer VM dan memeriksa status berbagai tombol atau menunggu tombol tertentu ditekan.

Ex9E Lewati instruksi berikutnya jika tombol v[x] diatur (ditekan).
ExA1 Lewati instruksi berikutnya jika tombol v[x] tidak diatur (tidak ditekan).
Fx07 Atur v[x] ke timer tunda.
Fx0A Tunggu hingga penekanan tombol berikutnya, lalu simpan tombol di v[x].
Fx15 Atur timer tunda ke v[x].
Fx18 Atur timer suara ke v[x].

Instruksi Register i

Semua instruksi dalam kumpulan terakhir ini terkait dengan register indeks memori (i).

Fx1E Tambahkan v[x] ke i.
Fx29 Atur i ke lokasi karakter v[x] dalam set font.
Fx33 Simpan nilai desimal berkode biner (BCD) dalam v[x] di lokasi memori i, i + 1, dan i + 2.
Fx55 Buang register v[0] hingga v[x] ke dalam memori, dimulai dari i.
Fx65 Simpan memori dari i hingga i + x ke dalam register v[0] hingga v[x].

Pertimbangkan sejenak betapa sederhananya instruksi-instruksi ini. Anda sebenarnya tidak memerlukan mekanisme canggih apa pun untuk memiliki "komputer" yang berfungsi. Bandingkan 35 instruksi CHIP-8 yang dijelaskan di sini dengan 8 instruksi dalam implementasi Brainfuck kita dari Bab 1. Keduanya adalah mesin Turing yang dibatasi memori, dan keduanya tidak jauh berbeda satu sama lain seperti yang mungkin terlihat dari sintaks instruksi yang tampak dangkal.

DESIMAL BERKODE BINER

Desimal berkode biner (BCD) adalah cara menyimpan angka desimal dalam bentuk biner. Saat ini tidak banyak digunakan, tetapi umum digunakan pada komputer-komputer awal. Misalnya, beberapa mikroprosesor dari tahun 1970-an menyertakan instruksi eksplisit untuk aritmatika BCD, yang menawarkan presisi lebih tinggi dalam menangani pembulatan desimal dan sampai batas tertentu membuat kode mesin lebih mudah dibaca. Bagi programmer modern pada umumnya, mempelajari BCD tidak banyak gunanya kecuali sebagai rasa ingin tahu.

Implementasi

Sekarang kita sudah mengetahui arsitektur CHIP-8, kita siap untuk mengimplementasikan VM kita. File `main.py` akan berisi loop eksekusi utama yang menangani

input pengguna, memperbarui tampilan, mengelola timer, dan yang terpenting, memberi tahu VM untuk melangkah ke instruksi berikutnya. File ini juga merupakan tempat argumen baris perintah yang menentukan file ROM diuraikan. Sementara itu, `vm.py` adalah VM yang sebenarnya.

> **ROM**
>
> Pernahkah Anda bertanya-tanya mengapa file yang menyimpan game yang digunakan dalam emulator disebut ROM? ROM adalah singkatan dari read-only memory (memori hanya baca). Sebagian besar sistem video game awal menggunakan kartrid plastik yang merupakan wadah khusus untuk chip ROM yang langsung dipasang ke konsol. Ketika game dikonversi menjadi file untuk emulator, seseorang harus memasang chip ROM ke perangkat khusus yang terhubung ke komputer mereka dan "mengambil" data dari chip ROM untuk menyimpannya dalam sebuah file. File tersebut akan berisi salinan persis data pada chip ROM, mungkin dengan beberapa informasi header tambahan tergantung pada ekosistem emulasi.
> Meskipun chip ROM asli tidak dapat dimodifikasi datanya, "file ROM" ini sama seperti file lainnya dan dapat dimodifikasi untuk mengubah gim. Oleh karena itu, muncul subkultur peretasan ROM, di mana pengembang mengubah grafis atau gameplay gim yang dimaksudkan untuk dijalankan di emulator.

Kita akan menggunakan dua pustaka eksternal dalam implementasi kita. Pygame, sebuah pustaka Python yang dirancang untuk pengembangan game, menyediakan cara mudah untuk menampilkan jendela di layar, mengisi jendela tersebut dengan piksel dari tampilan VM kita, dan menangani input keyboard. NumPy, sebuah pustaka komputasi numerik, dapat membantu membuat array dua dimensi yang digunakan sebagai buffer pendukung untuk piksel jendela Pygame. Array ini akan berfungsi sebagai "RAM grafis" VM kita. Pygame secara native bekerja dengan array NumPy, dan array NumPy lebih berkinerja daripada apa pun di pustaka standar Python untuk merepresentasikan buffer ini. Pastikan Anda telah menginstal Pygame dan NumPy sebelum menjalankan program.

Seperti mereplikasi format file di Bab 3, mengimplementasikan VM atau emulator membutuhkan sejumlah besar manipulasi bit tingkat rendah. Lihat lampiran untuk membaca tentang operator bitwise Python.

Loop Eksekusi

Loop eksekusi bertanggung jawab untuk memajukan VM satu instruksi, menggambar ulang layar, menangani setiap kejadian (penekanan tombol yang akan diteruskan ke VM), memainkan suara bip, dan memperbarui dua timer CHIP-8. Pygame membuat menggambar, memainkan suara, dan membaca input keyboard hampir mudah; ini adalah pustaka yang

sangat mudah digunakan. Mari kita mulai dengan beberapa kode inisialisasi dan lanjutkan hingga ke awal loop eksekusi:

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):
# Startup Pygame, create the window, and load the sound 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) # load the virtual machine with the program data
timer_accumulator = 0.0 # used to limit the timer to 60 Hz
# Main virtual machine loop
while True:
   frame_start = timer()
   vm.step()
   if vm.needs_redraw:
      pygame.surfarray.blit_array(screen, vm.display_buffer)
      pygame.display.flip()
```

Pada awal run loop, waktu dicatat dengan `frame_start = timer()` untuk mengukur durasi setiap iterasi loop. Ini karena timer CHIP-8 perlu dikurangi 60 kali per detik (jika nilainya di atas nol). VM kemudian diperintahkan untuk mengeksekusi instruksi (dan karenanya untuk berpindah ke instruksi berikutnya) melalui `vm.step()`. Jika ditunjukkan oleh `vm.needs_redraw`, tampilan kemudian digambar ulang melalui dua panggilan sederhana ke Pygame. Satu menyalin buffer tampilan VM ke layar, dan yang lainnya menampilkannya.

Perhatikan bahwa kode tersebut menggunakan istilah frame sedikit berbeda dari biasanya. Dalam sebagian besar program, satu frame adalah satu penyegaran penuh dari keseluruhan output grafis program, tetapi dalam konteks ini, run loop kita tidak akan selalu menggambar ulang grafik setiap iterasi, karena `vm.needs_redraw` mungkin tidak selalu bernilai True.

Yang pasti akan terjadi setiap "frame" adalah satu instruksi akan dieksekusi sebagai hasil dari panggilan ke `vm.step()`. Oleh karena itu, penulis berpikir untuk menggunakan kata "instruksi" daripada "frame" di bagian kode ini, misalnya, `"instruksi_start"` daripada `"frame_start"`. Namun, lebih dari sekadar eksekusi instruksi yang terjadi di dalam run loop—ada juga output grafis, penanganan keyboard, dan output suara—jadi "instruksi" terdengar terlalu terbatas. Tetapi sekali lagi, "frame" juga tidak sepenuhnya akurat. Memang benar apa yang mereka katakan: salah satu masalah tersulit dalam ilmu komputer adalah penamaan.

Run loop diakhiri dengan menangani peristiwa keyboard, memutar suara ketika Boolean `vm.play_sound` VM menunjukkan, dan menangani pengaturan waktu:

```
# Handle keyboard events
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()

# Sound
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()

# Handle timing
frame_end = timer()
frame_time = frame_end - frame_start # time it took in seconds
timer_accumulator += frame_time
# Every 1/60 of a second decrement the timers
if timer_accumulator > TIMER_DELAY:
  ❶ vm.decrement_timers()
    timer_accumulator = 0
# Limit the speed of the entire machine to 500 "frames" per second
if frame_time < FRAME_TIME_EXPECTED:
        difference = FRAME_TIME_EXPECTED - frame_time
  ❷ pygame.time.delay(int(difference * 1000))
    timer_accumulator += difference
```

Meskipun kita tidak menggunakan frame untuk mengukur frame per second (FPS) tradisional, seperti yang mungkin Anda kenal dari game, pengaturan waktu setiap iterasi tetap penting. Kita perlu melacak pengaturan waktu untuk memastikan penghitung waktu mundur VM berjalan setiap 1/60 detik seperti yang dipersyaratkan oleh spesifikasi CHIP-8 ❶, dan untuk membatasi kecepatan keseluruhan VM ❷.

Jika VM berjalan terlalu cepat, game tidak akan dapat dimainkan karena dirancang untuk komputer lambat tahun 1970-an. Anda dapat menyesuaikan kecepatan VM, dan karenanya perangkat lunak apa pun yang berjalan di atasnya, dengan mengubah konstanta FRAME_TIME_EXPECTED di `vm.py`. Dalam pengujian, penulis menemukan bahwa 500 "frame" per detik, atau dengan kata lain, setiap "frame" kira-kira 1/500 detik, merupakan kecepatan yang solid untuk sebagian besar game.

5.3 ARGUMEN BARIS PERINTAH

Seperti pada program sebelumnya, kita menggunakan `ArgumentParser` untuk menangani argumen baris perintah:

```
if __name__ == "__main__":
    # Parse the file argument
    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)
```

Dalam hal ini, kita hanya memiliki satu argumen baris perintah—nama file yang berisi data program untuk VM CHIP-8. Byte mentah file dibaca dan diteruskan ke `run()`, yang kemudian diteruskan ke konstruktor VM.

Pengaturan VM dan Fungsi Pembantu

Kita siap untuk implementasi VM yang sebenarnya. Kita mulai, seperti yang sering kita lakukan, dengan beberapa konstanta:

Chip8/vm.py

```
from array import array
from random import randint
import numpy as np
import pygame
import sys
RAM_SIZE = 4096 # in bytes, aka 4 kilobytes
SCREEN_WIDTH = 64
SCREEN_HEIGHT = 32
SPRITE_WIDTH = 8
WHITE = 0xFFFFFFFF
BLACK = 0
TIMER_DELAY = 1/60 # in seconds... about 60 Hz
FRAME_TIME_EXPECTED = 1/500 # for limiting VM speed
ALLOWED_KEYS = ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9",
                "a", "b", "c", "d", "e", "f"]

# The font set, hardcoded
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
]
```

Sebagian besar konstanta ini mudah dipahami dan sesuai dengan spesifikasi CHIP-8 asli. VM memiliki memori utama 4KB. Ia menentukan grafis dalam bentuk gambar keluaran hitam-putih dengan resolusi 64×32. Timer diperbarui 60 kali per detik. Sistem CHIP-8 asli memiliki 16 tombol yang dapat Anda tekan pada kontroler. Kita mungkin dapat mengaturnya dengan cara yang lebih ergonomis untuk bermain game dengan memetakannya ke tombol lain, tetapi dalam implementasi kita, kita akan membiarkan tombol-tombol tersebut tetap berada di tempatnya pada keyboard.

Mungkin konstanta yang paling tidak biasa di sini adalah `FONT_SET`. Ini adalah 80 byte data grafis untuk menampilkan angka `0–9` dan huruf `A–F`. Setiap karakter ditentukan oleh bit yang mewakili piksel karakter jika ditampilkan di layar. Anggap saja sebagai font primitif yang hanya memiliki 16 karakter. Beberapa game mengharapkan data ini berada di 80 byte pertama memori sehingga mereka dapat menulis pesan di layar kepada pengguna.

Selanjutnya, kita memiliki fungsi pembantu yang tidak terkait dengan status VM:

```
def concat_nibbles(*args: int) -> int:
    result = 0
    for arg in args:
        result = (result << 4) | arg
    return result
```

Fungsi `concat_nibbles()` mengambil sejumlah bilangan bulat dan menggabungkannya satu demi satu dengan menggeser setiap 4 bit ke kiri dan melakukan operasi OR bitwise dengan bilangan bulat berikutnya. Ini hanya akan berguna jika bilangan bulat itu sendiri berukuran 4 bit. Misalkan kita memiliki bilangan bulat `0111`. Menggesernya 4 bit ke kiri akan menyebabkan empat angka nol mengikuti 4 bit asli, seperti `01110000`. Sekarang misalkan kita memiliki bilangan bulat 4 bit lainnya, `1010`. Jika kita melakukan operasi OR dengan `01110000`, kita mendapatkan hasil `01111010`, penggabungan dari dua bilangan bulat 4 bit asli. Kita dapat terus melakukan ini untuk sejumlah bilangan bulat 4 bit untuk menggabungkannya.

Ingat bahwa bilangan bulat 4 bit dikenal sebagai nibble. Instruksi 16-bit dalam CHIP-8 dibagi menjadi empat nibble, dan setiap nibble seringkali memiliki arti yang berbeda. Secara default, kita akan membagi setiap instruksi menjadi empat nibble penyusunnya, tetapi untuk beberapa instruksi, kita perlu menggunakan nilai dari beberapa nibble gabungan. Oleh karena itu, fungsi pembantu `concat_nibbles()` sangat berguna.

Kelas VM dimulai dengan konstruktor yang menginisialisasi semua keadaan yang dapat diubah termasuk register, RAM, tumpukan, buffer tampilan (yang saat ini kita sebut VRAM atau RAM video), timer, dan beberapa variabel pembantu lainnya:

```
class VM:
  def __init__(self, program_data: bytes):
    # Initialized registers and memory constructs
    # General Purpose Registers - CHIP-8 has 16 of these registers
    self.v = array('B', [0] * 16)
    # Index Register
    self.i = 0
    # Program Counter
    # Starts at 0x200 because addresses below that were
    # used for the VM itself in the original CHIP-8 machines
    self.pc = 0x200
    # Memory - the standard 4k on the original CHIP-8 machines
    self.ram = array('B', [0] * RAM_SIZE)
    # Load the font set into the first 80 bytes
    self.ram[0:len(FONT_SET)] = array('B', FONT_SET)
    # Copy program into RAM starting at byte 512 by convention
    self.ram[512:(512 + len(program_data))] = array('B', program_data)
    # Stack - in real hardware this is typically limited to
    # 12 or 16 PC addresses for jumps, but since we're on modern hardware,
    # ours can just be unlimited and expand/contract as needed
    self.stack = []
    # Graphics buffer for the screen - 64 x 32 pixels
    self.display_buffer = np.zeros((SCREEN_WIDTH, SCREEN_HEIGHT),
                                   dtype=np.uint32)
    self.needs_redraw = False
    # Timers - really simple registers that count down to 0 at 60 hertz
    self.delay_timer = 0
    self.sound_timer = 0
    # These hold the status of whether the keys are down
    # CHIP-8 has 16 keys
    self.keys = [False] * 16
```

Beberapa variabel status ini memiliki nilai default yang penting. Misalnya, penghitung program (`pc`) harus selalu diatur ke lokasi `0x200` (512 dalam desimal) karena 512 byte pertama memori di mesin CHIP-8 awalnya digunakan untuk menyimpan VM CHIP-8 itu sendiri. Ini berarti program CHIP-8 tidak dapat menggunakan memori tersebut dan harus dimulai pada byte 512. Penulis telah memberikan komentar yang ekstensif pada konstruktor untuk menjelaskan setiap variabel saat dideklarasikan. Perhatikan bahwa sebagian besar VM kita hanya menggunakan pustaka standar Python untuk implementasinya, kecuali display_buffer, yang merupakan array NumPy. Ini adalah format yang diharapkan Pygame.

Selanjutnya, kita memiliki metode pembantu yang sederhana, decrement_timers(), dan properti dinamis yang sederhana, `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
```

Baik `decrement_timers()` maupun `play_sound` digunakan dalam loop eksekusi yang telah kita bahas sebelumnya di *`__main__.py`*.

Grafis

CHIP-8 melihat layar sebagai bidang 64×32 piksel dengan sistem koordinat Kartesius yang memiliki titik asal, lokasi `(0,0)`, di kiri atas, dan sumbu y berorientasi ke bawah. Dengan kata lain, koordinat x meningkat saat kita bergerak dari kiri ke kanan dan koordinat y meningkat saat kita bergerak dari atas ke bawah. Oleh karena itu, piksel kanan bawah berada di lokasi `(63,31)`. Tidak ada koordinat negatif, dan tidak mungkin untuk mengakses lokasi piksel di luar layar.

Setiap piksel direpresentasikan dalam memori sebagai satu bit. Dalam implementasi kami, 1 mewakili piksel putih dan 0 mewakili piksel hitam. Memori grafis (atau "buffer") terpisah dari memori program utama dan hanya dapat dimanipulasi secara tidak langsung menggunakan instruksi CHIP-8. Pygame menggunakan bilangan bulat 32-bit untuk merepresentasikan piksel di layar dalam format RGBA (A adalah untuk alfa, atau transparansi), sehingga setiap nilai piksel 1-bit kita harus menjadi bilangan bulat 32-bit ketika kita menyimpannya di `display_buffer`.

CHIP-8 menggambar menggunakan sprite, yang merupakan bitmap kecil (atau gambar, jika Anda suka) yang dapat bergerak di sekitar layar. Setiap sprite di CHIP-8 memiliki lebar 8 piksel dan tingginya dapat berkisar antara 1 hingga 15 piksel. Gambar 5-1 mengilustrasikan sprite 8×3 yang merepresentasikan kata HI yang digambar di layar pada lokasi `(28,15)`.

Gambar 5-1: Kata HI sebagai sprite 8×3

Karena setiap baris dalam sprite CHIP-8 tepat 8 piksel, maka direpresentasikan menggunakan 8 bit. Karena 8 bit sama dengan 1 byte, maka setiap baris sprite dapat direpresentasikan oleh satu byte. Karena sprite HI memiliki tiga baris, maka dapat direpresentasikan oleh 3 byte. Dalam biner, 3 byte tersebut akan terlihat seperti ini:

```
10100111
11100010
10100111
```

Perhatikan bagaimana setiap angka 1 dipetakan ke piksel putih dan setiap angka 0 dipetakan ke piksel hitam. Dengan informasi ini, semoga set font yang kita definisikan sebelumnya juga lebih masuk akal sekarang: setiap karakter dalam set font hanyalah sprite 8×5.

Menggambar sprite adalah satu-satunya cara untuk memodifikasi buffer tampilan, selain membersihkannya, jadi CHIP-8 VM memiliki satu instruksi gambar, Dxyn. Instruksi ini menggambar sprite dengan tinggi tertentu yang berada di lokasi memori yang ditentukan oleh register `i`. D dalam instruksi adalah nibble konstan, dan nibble `x` dan y mewakili indeks ke dalam register v di mana koordinat x dan y untuk kiri atas sprite harus berada. Dengan kata lain, koordinat x diambil dari register `v[x]` dan koordinat y dari register `v[y]`. Nibble n mewakili tinggi sprite. Inilah mengapa sprite tidak bisa lebih tinggi dari `15` piksel: satu nibble adalah 4 bit, dan 4 bit maksimal dapat merepresentasikan angka `15`.

Nibble pada Dxyn sesuai dengan parameter metode bantu `draw_sprite()`:

```
# Draw a sprite at *x*, *y* using data at *i* with a height of *height*
def draw_sprite(self, x: int, y: int, height: int):
   flipped_black = False # did drawing this flip any pixels?
   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 # ignore off-screen pixels
         new_bit = (row_bits >> (7 - col)) & 1
         old_bit = self.display_buffer[px, py] & 1
         if new_bit & old_bit: # if both set, flip white -> black
            flipped_black = True
         # CHIP-8 draws by XORing
         new_pixel = new_bit ^ old_bit
         self.display_buffer[px, py] = WHITE if new_pixel else BLACK
   # Set flipped flag for collision detection
   self.v[0xF] = 1 if flipped_black else 0
```

CHIP-8 menggambar sprite menggunakan operasi XOR. XOR, atau exclusive or, adalah operasi bitwise yang mengembalikan nilai 1 jika dua bit berbeda dan nilai 0 jika keduanya

sama. Python menggunakan operator ^ untuk XOR. Tabel 5-2 menunjukkan tabel kebenaran untuk XOR.

Tabel 5-2: Tabel Kebenaran XOR

0 ^ 0	0 ^ 1	1 ^ 0	1 ^ 1
0	1	1	0

Instruksi draw CHIP-8 mengambil sebuah sprite dan melakukan operasi XOR antara piksel-pikselnya dengan piksel-piksel yang sudah ada di layar pada lokasi yang ditentukan. Jika lokasi layar ini seluruhnya terdiri dari piksel hitam, maka instruksi ini hanya akan menggambar sprite. Namun, jika lokasi layar tersebut berisi beberapa piksel putih (1), maka piksel hitam akan digambar di tempat piksel putih sprite tumpang tindih dengan piksel putih layar. Hal ini karena 1 XOR 1 adalah 0. Instruksi draw CHIP-8 melacak apakah terjadi tumpang tindih (piksel putih layar diubah menjadi piksel hitam dengan menggambar sprite). Jika terjadi, instruksi ini akan mengatur register flag `(v[0xF])`.

Metode `draw_sprite()` merupakan kodifikasi dari proses ini. Kita mengulangi semua baris dan kolom dari sebuah sprite yang dimulai pada lokasi memori yang ditentukan oleh register `i`, mengambil setiap piksel dari sprite menggunakan operasi pergeseran kanan dan menyimpannya di `new_bit`. Operasi & pada data yang masuk ke new_bit memastikan bahwa hanya bit terakhir dari operasi pergeseran yang disimpan di `new_bit`. Kita membandingkan setiap `new_bit` dengan bit yang sudah ada di layar, old_bit, dan jika old_bit akan berubah dari putih menjadi hitam, kita mengatur register flag. Kita mengubah buffer tampilan dengan mengambil XOR dari `new_bit` dan `old_bit`.

Mengapa kita membutuhkan flag untuk melacak apakah menggambar sprite menyebabkan piksel layar yang sebelumnya menyala menjadi mati? Ini pada dasarnya adalah bentuk deteksi tabrakan. Jika sprite mengenai sesuatu yang sudah ada di layar, itu sangat membantu untuk diketahui dalam sebuah game. Misalnya, jika Anda memprogram game tenis, Anda ingin tahu kapan bola bergerak dan mengenai raket yang sudah ada di layar.

Eksekusi Instruksi

Sekarang saatnya untuk inti dari VM. Kita memiliki satu metode tersisa, tetapi ini adalah metode yang besar: kita perlu mengimplementasikan semua instruksi VM. Ini tidak jauh berbeda dengan mengeksekusi pernyataan dalam interpreter kita di Bab 1 dan 2. Baik mengeksekusi pernyataan interpreter, instruksi VM, atau opcode mikroprosesor dalam emulator, kita perlu melakukan sesuatu yang cukup sederhana: mengenali instruksi berikutnya dan kemudian mengeksekusi beberapa baris kode berbeda yang memanipulasi keadaan VM berdasarkan operasi yang dimaksudkan.

Misalnya, jika kita melihat instruksi penjumlahan, kita harus menjumlahkan dua angka yang ditentukan dan menyimpan hasilnya di lokasi yang ditentukan. Jika kita melihat instruksi lompatan, kita harus memindahkan eksekusi ke lokasi yang ditentukan di memori. Intinya adalah mengenali instruksi apa yang sedang dieksekusi dan mengubah beberapa variabel status yang mewakili memori, register, dan sejenisnya berdasarkan instruksi tersebut. Cara

paling sederhana untuk melakukan ini adalah dengan sejumlah besar pernyataan `if`. Pseudokodenya mungkin terlihat seperti ini:

```
if instruction == ADD:
  add some numbers together and store the sum
elif instruction == JUMP:
  jump to a location by changing the program counter
elif instruction == DRAW:
  draw the sprite where specified by changing the display buffer
etc.
```

Selain menggunakan banyak pernyataan `if`, ada tiga pola umum untuk menulis kode yang mengeksekusi instruksi. Yang pertama adalah pernyataan switch yang besar, sebuah konstruksi yang ada di banyak bahasa tetapi tidak sepenuhnya sama di Python dalam bentuk yang sama. Penulis berasumsi sebagian besar pembaca pernah melihat pernyataan switch sebelumnya dalam bahasa seperti C atau Java.

Jika belum, Anda dapat menganggapnya sebagai bentuk primitif dari pernyataan match Python seperti yang kita gunakan di Bab 1 dan 2. Kasus pernyataan switch yang dieksekusi bergantung pada instruksi. Ini agak mirip dengan pseudocode yang baru saja ditunjukkan. Bahkan, sebelum diperkenalkannya pernyataan match di Python 3.10, cara Anda mengimplementasikan pola ini di Python memang dengan banyak klausa if dan elif. Ini adalah cara paling sederhana untuk mengimplementasikan eksekusi instruksi, tetapi dapat menjadi sulit dikelola untuk kumpulan instruksi yang besar.

Pola selanjutnya adalah menggunakan tabel lompatan, yang terdiri dari array penunjuk fungsi. Kita mengindeks ke dalam array tergantung pada instruksi dan kemudian mengeksekusi fungsi yang sesuai yang dikembalikan. Instruksi hanyalah bilangan bulat, itulah sebabnya instruksi dapat digunakan sebagai indeks array. Jika instruksi berupa string karena suatu alasan, kita dapat menggunakan kamus di mana kuncinya adalah instruksi dan nilainya adalah penunjuk fungsi, meskipun ini sedikit kurang efisien. Karena pola ini membagi pekerjaan di banyak fungsi pembantu, umumnya menghasilkan kode yang lebih bersih daripada pernyataan switch yang besar dan mungkin lebih disukai untuk set instruksi yang lebih besar.

Pola ketiga adalah menggunakan kompilasi ulang dinamis, di mana kita menerjemahkan setiap instruksi ke dalam instruksi yang dipahami oleh perangkat keras yang mendasarinya (atau sesuatu yang dapat diterjemahkan lebih lanjut ke dalam hal tersebut). Misalnya, jika kita memiliki instruksi penjumlahan di VM yang berjalan pada mikroprosesor x86, kita dapat menerjemahkan instruksi penjumlahan VM ke dalam kode mesin untuk instruksi penjumlahan x86 yang setara. Ini adalah pola yang paling rumit untuk diimplementasikan karena membutuhkan pengetahuan mendalam tidak hanya tentang set instruksi asli tetapi juga set instruksi yang diterjemahkan. Namun, ini akan menghasilkan kinerja tercepat.

Dalam program ini, kita akan menggunakan pernyataan pencocokan raksasa karena set instruksi CHIP-8 relatif kecil. Ketika kita membuat emulator NES di bab berikutnya, kita akan

menggunakan tabel lompatan karena mikroprosesor 6502 memiliki set instruksi yang ukurannya kira-kira dua kali lipat (walaupun masih jauh lebih kecil daripada hampir semua mikroprosesor lainnya). Kompilasi ulang dinamis adalah teknik yang jauh lebih rumit dan di luar cakupan buku ini.

Metode `step()` bertanggung jawab untuk mengeksekusi instruksi, tetapi pertama-tama metode tersebut perlu mengambil instruksi berikutnya yang akan dieksekusi:

```
def step(self):
    # We look at the opcode in terms of its nibbles (4 bit pieces)
    # Opcode is 16 bits made up of next two bytes in memory
    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
```

Instruksi berikutnya terletak di alamat memori yang disimpan dalam penghitung program (pc). Karena instruksi terdiri dari 16 bit, kita mengambil 2 byte berikutnya di pc dan menyimpannya di `first2` dan `last2`. Seperti yang dibahas sebelumnya, akan lebih mudah untuk memikirkan setiap instruksi CHIP-8 sebagai kombinasi dari empat nibble, karena setiap nibble individual bermakna bagi banyak instruksi. Kita menyimpan nibble di `first`, `second`, `third`, dan `fourth`. Semua pencocokan pola di sekitar instruksi kita akan dalam bentuk nibble.

Saat kita mengeksekusi instruksi, kita juga akan melacak apakah instruksi tersebut memerlukan penggambaran ulang melalui needs_redraw dan apakah instruksi tersebut memodifikasi pc melalui jumped. Loop eksekusi menggunakan `needs_redraw` sebagai optimasi. Mengapa melakukan penggambaran jika tidak ada yang berubah? Melacak jumped memungkinkan beberapa kode umum berada di bagian bawah `step()`, mengurangi sedikit duplikasi kode.

Sekarang kita sampai pada instruksi sebenarnya. Pernyataan match yang besar ada di hadapan kita. Implementasi kami menggunakan sintaks match yang elegan dari Python untuk menangkap nibble yang diperlukan untuk eksekusi instruksi dalam variabel sementara. Detail eksekusi setiap instruksi mengikuti langsung dari deskripsinya sebelumnya di bab ini. Banyak instruksi dapat diimplementasikan hanya dalam satu baris kode. Akan sangat membosankan untuk menulis tentang masing-masing instruksi secara berurutan. Sebagai gantinya, berikut ini adalah reproduksi dari sisa `step()`, dengan komentar yang memberikan sedikit konteks tambahan.

Namun, sebelum Anda melihat kodenya, ini adalah tempat yang baik untuk berhenti dan mencoba mengimplementasikan instruksi sendiri. Anda tidak harus menggunakan pernyataan match. Anda dapat menggunakan serangkaian pernyataan `if...elif` seperti yang

penulis lakukan di Python 3.9 sebelum pernyataan match ada. (Saya telah menguji dan hampir tidak ada perbedaan kinerja antara keduanya.) Anda sudah memiliki semua pengaturan yang Anda butuhkan untuk dapat berkonsentrasi hanya pada apa yang seharusnya dilakukan setiap instruksi, alih-alih mengkonfigurasi memori sistem atau representasi register. Anda tidak perlu memikirkan tentang memuat file ROM atau apa yang seharusnya menjadi beberapa konstanta. Pikirkan saja tentang logika dan bagaimana setiap operasi akan memodifikasi keadaan VM.

Beberapa deskripsi instruksi sebelumnya dalam bab ini cukup singkat, tetapi Anda dapat menemukan instruksi yang lebih detail di salah satu dari banyak referensi CHIP-8 online. Namun, jangan terlalu lama menghabiskan waktu pada satu instruksi. Anda selalu dapat melihat implementasinya di sini jika Anda mengalami kesulitan. Setelah Anda mencoba menulis implementasi instruksi Anda sendiri, Anda dapat kembali ke kode buku ini untuk memeriksa ulang pekerjaan Anda. Melakukan pekerjaan ini sendiri terlebih dahulu akan memberi Anda gambaran yang baik tentang apa yang diperlukan untuk menulis VM atau emulator sederhana. Jangan takut: Anda akan kagum betapa mudahnya mengimplementasikan banyak instruksi. Ingat, VM CHIP-8 asli hanya membutuhkan 512 byte memori!

```
match (first, second, third, fourth):
    case (0x0, 0x0, 0xE, 0x0): # display clear
        self.display_buffer.fill(0)
        self.needs_redraw = True
    case (0x0, 0x0, 0xE, 0xE): # return from subroutine
        self.pc = self.stack.pop()
        jumped = True
    case (0x0, n1, n2, n3): # call program
        self.pc = concat_nibbles(n1, n2, n3) # go to start
        # Clear registers
        self.delay_timer = 0
        self.sound_timer = 0
        self.v = array('B', [0] * 16)
        self.i = 0
        # Clear screen
        self.display_buffer.fill(0)
        self.needs_redraw = True
        jumped = True
    case (0x1, n1, n2, n3): # jump to address
        self.pc = concat_nibbles(n1, n2, n3)
        jumped = True
    case (0x2, n1, n2, n3): # call subroutine
        self.stack.append(self.pc + 2) # put return place on stack
        self.pc = concat_nibbles(n1, n2, n3) # goto subroutine
        jumped = True
    case (0x3, x, _, _): # conditional skip v[x] equal last2
        if self.v[x] == last2:
            self.pc += 4
            jumped = True
    case (0x4, x, _, _): # conditional skip v[x] not equal last2
```

```
        if self.v[x] != last2:
            self.pc += 4
            jumped = True
    case (0x5, x, y, _): # conditional skip v[x] equal v[y]
        if self.v[x] == self.v[y]:
            self.pc += 4
            jumped = True
    case (0x6, x, _, _): # set v[x] to last2
        self.v[x] = last2
    case (0x7, x, _, _): # add last2 to v[x]
        self.v[x] = (self.v[x] + last2) % 256
    case (0x8, x, y, 0x0): # set v[x] to v[y]
        self.v[x] = self.v[y]
    case (0x8, x, y, 0x1): # set v[x] to v[x] | v[y]
        self.v[x] |= self.v[y]
    case (0x8, x, y, 0x2): # set v[x] to v[x] & v[y]
        self.v[x] &= self.v[y]
    case (0x8, x, y, 0x3): # set v[x] to v[x] ^ v[y]
        self.v[x] ^= self.v[y]
    case (0x8, x, y, 0x4): # add with carry flag
        try:
            self.v[x] += self.v[y]
            self.v[0xF] = 0 # indicate no carry flag
        except OverflowError:
            self.v[x] = (self.v[x] + self.v[y]) % 256
            self.v[0xF] = 1 # set carry flag
    case (0x8, x, y, 0x5): # subtract with borrow flag
        try:
            self.v[x] -= self.v[y]
            self.v[0xF] = 1 # indicate no borrow (yes, weird it's 1)
        except OverflowError:
            self.v[x] = (self.v[x] - self.v[y]) % 256
            self.v[0xF] = 0 # indicates there was a borrow
    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): # subtract with borrow flag (y - x in x)
        try:
            self.v[x] = self.v[y] - self.v[x]
            self.v[0xF] = 1 # indicate no borrow (yes, weird it's 1)
        except OverflowError:
            self.v[x] = (self.v[y] - self.v[x]) % 256
            self.v[0xF] = 0 # indicates there was a borrow
    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): # conditional skip if v[x] != v[y]
        if self.v[x] != self.v[y]:
            self.pc += 4
            jumped = True
    case (0xA, n1, n2, n3): # set i to address n1n2n3
        self.i = concat_nibbles(n1, n2, n3)
```

```
case (0xB, n1, n2, n3): # jump to n1n2n3 + v[0]
    self.pc = concat_nibbles(n1, n2, n3) + self.v[0]
    jumped = True
case (0xC, x, _, _): # v[x] = random number (0-255) & last2
    self.v[x] = last2 & randint(0, 255)
case (0xD, x, y, n): # draw sprite at (vx, vy) that's n high
    self.draw_sprite(self.v[x], self.v[y], n)
    self.needs_redraw = True
case (0xE, x, 0x9, 0xE): # conditional skip if keys(v[x])
    if self.keys[self.v[x]]:
        self.pc += 4
        jumped = True
case (0xE, x, 0xA, 0x1): # conditional skip if not keys(v[x])
    if not self.keys[self.v[x]]:
        self.pc += 4
        jumped = True
case (0xF, x, 0x0, 0x7): # set v[x] to delay_timer
    self.v[x] = self.delay_timer
case (0xF, x, 0x0, 0xA): # wait until next key then store in v[x]
    # Wait here for the next key then continue
    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): # set delay_timer to v[x]
    self.delay_timer = self.v[x]
case (0xF, x, 0x1, 0x8): # set sound_timer to v[x]
    self.sound_timer = self.v[x]
case (0xF, x, 0x1, 0xE): # add vx to i
    self.i += self.v[x]
case (0xF, x, 0x2, 0x9): # set i to location of character v[x]
    self.i = self.v[x] * 5 # built-in font set is 5 bytes apart
case (0xF, x, 0x3, 0x3): # store BCD at v[x] in i, i+1, i+2
    self.ram[self.i] = self.v[x] // 100 # 100s digit
    self.ram[self.i + 1] = (self.v[x] % 100) // 10 # 10s digit
    self.ram[self.i + 2] = (self.v[x] % 100) % 10 # 1s digit
case (0xF, x, 0x5, 0x5): # reg dump v0 to vx starting at i
    for r in range(0, x + 1):
        self.ram[self.i + r] = self.v[r]
case (0xF, x, 0x6, 0x5): # store i through i+r in v0 through 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 # increment program counter
```

Di akhir `step()`, kita menambah penghitung program jika kita tidak melakukan lompatan. Ini memastikan bahwa kita akan melanjutkan ke instruksi berikutnya pada saat `step()` dipanggil lagi. Karena setiap instruksi CHIP-8 berukuran 2 byte, penghitung program ditambah 2. Jika ada lompatan, maka eksekusi langsung dipindahkan ke instruksi berbeda tertentu di tempat lain dalam memori.

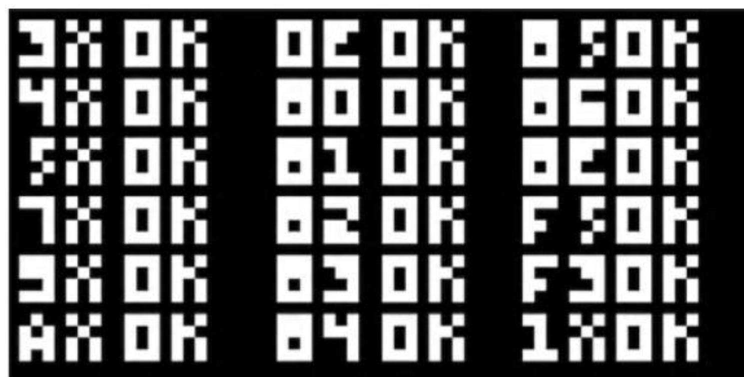
5.4 PENGGUNAAN VM

Cara paling detail untuk menguji VM adalah dengan menulis unit test kita sendiri untuk setiap instruksi. Untuk setiap pengujian, kita akan mencoba menjalankan instruksi dan kemudian memverifikasi bahwa keadaan internal VM selanjutnya benar. Meskipun ini ideal, demi menghemat waktu dan ruang, kita akan melakukan sesuatu yang lebih mirip dengan pengujian integrasi: kita akan melihat bagaimana VM kita bekerja saat menjalankan program CHIP-8 yang sebenarnya. Apakah program tersebut berjalan dengan benar?

Kebetulan, bahkan ada ROM pengujian yang menawarkan semacam solusi lengkap untuk menguji VM CHIP-8. Dua ROM uji tersebut disertakan dalam subdirektori Chip8/Tests dari repositori kode sumber buku ini. Kedua ROM uji tersebut dirilis di bawah lisensi terbuka oleh pengembangnya, dan lisensi tersebut disertakan dalam subdirektori. Mari kita jalankan ROM uji pertama dari direktori utama repositori:

```
% python3 -m Chip8 Chip8/Tests/chip8-test-rom/test_opcode.ch8
```

Jika VM berfungsi dengan benar, Anda akan melihat layar bertuliskan OK, seperti yang ditunjukkan pada Gambar 5-2.



Gambar 5-2: Menjalankan ROM uji pertama

Sekarang mari kita periksa pekerjaan kita dengan ROM uji kedua:

```
% python3 -m Chip8 Chip8/Tests/chip8-test-rom-2/chip8-test-rom.ch8
```

Yang ini hanya menampilkan OK satu kali di pojok kiri atas (lihat Gambar 5-3).

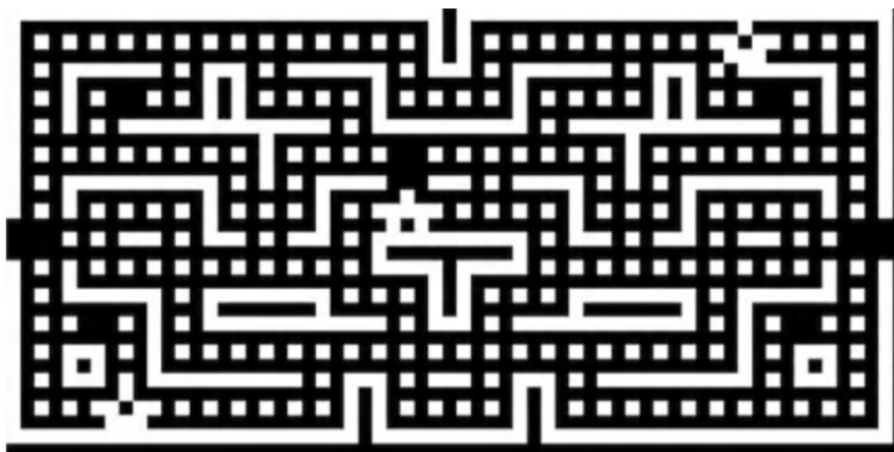
Gambar 5-3: Menjalankan ROM uji kedua

Uji coba ini tidak komprehensif, tetapi merupakan titik awal yang baik. Sekarang saatnya untuk uji integrasi utama: Dapatkah VM kita memainkan game dengan akurat?

Memainkan Game

Subdirektori Chip8/Games di repositori buku berisi pilihan ROM CHIP-8 yang telah ditempatkan ke domain publik. Jika Anda merasa skema kontrol beberapa di antaranya agak sulit digunakan, pertimbangkan untuk mengubah pengikatan kunci default. Saat ini, `ALLOWED_KEYS` dibaca langsung dari tombol masing-masing, jadi A di VM adalah tombol A pada keyboard. Namun, sistem tempat game ini dimainkan mungkin memiliki tata letak tombol yang sangat berbeda, jadi skema yang berbeda mungkin lebih baik untuk beberapa game.

Sebagian besar game cukup sederhana, yang masuk akal mengingat keterbatasan perangkat keras yang awalnya dimaksudkan untuk menjalankan VM. Ada klon game populer untuk sistem yang lebih mumpuni. Pertama, kita memiliki BLINKY, semacam klon Pac-Man (Gambar 5-4).

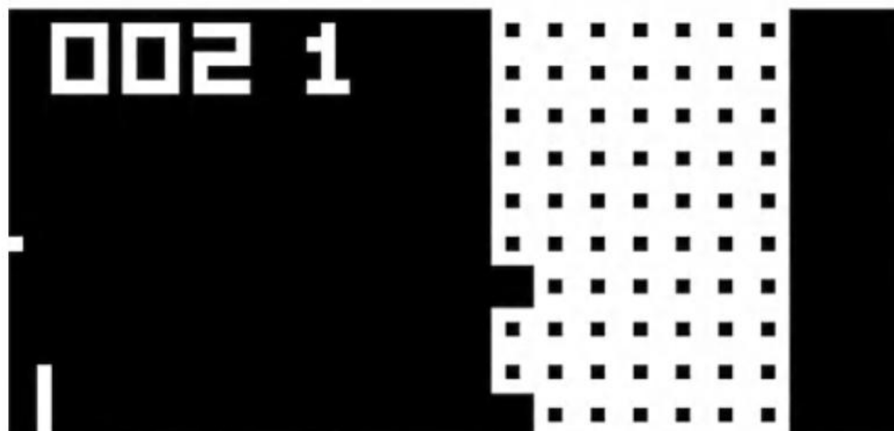


Gambar 5-4: Permainan BLINKY berjalan di VM

INVADERS adalah klon dari Space Invaders (Gambar 5-5).

Gambar 5-5: Game INVADERS berjalan di VM

VBRIX adalah bentuk vertikal dari Breakout (Gambar 5-6).



Gambar 5-6: Game VBRIX berjalan di VM

Dan kemudian ada PONG (Gambar 5-7).

Gambar 5-7: Game PONG berjalan di VM

Ada beberapa game lagi yang dapat Anda periksa yang disertakan dalam repositori kode sumber. Perhatikan ukuran file: sebagian besar game ini berukuran 500 byte atau kurang! Yang terbesar, BLINKY, hanya 2KB.

KODE BERTEMU KEHIDUPAN

Saya selalu tertarik untuk mengembangkan emulator saya sendiri, tetapi saya tidak merasa cukup percaya diri untuk membangunnya sampai jauh ke dalam kehidupan pemrograman saya. Ketika saya mulai meneliti cara menulis emulator, saran standar yang saya temukan adalah untuk mencoba menulis VM CHIP-8 terlebih dahulu karena hal itu lebih mudah daripada menulis hampir semua emulator tetapi membutuhkan semua elemen yang sama (menangani opcode, mensimulasikan memori dan register, grafis, dan sebagainya).

Saya menemukan tutorial online yang cukup bagus. Namun, saya memutuskan bahwa saya ingin membuatnya sedikit lebih menantang, jadi saya mengembangkan VM CHIP-8 awal saya dalam bahasa Swift yang saat itu masih baru, yang banyak saya gunakan dalam pekerjaan profesional saya. Itu adalah proyek akhir pekan, titik awal yang saya butuhkan untuk mulai mengembangkan emulator.

Aplikasi Dunia Nyata

VM (Virtual Machine) sangat umum dalam pengembangan perangkat lunak, baik historis maupun modern. Keunggulan utamanya adalah portabilitas. Program yang ditulis untuk VM akan berjalan di platform apa pun yang memiliki implementasi VM tersebut. VM juga menyediakan infrastruktur yang mengurangi beban penulis bahasa dengan menghilangkan kebutuhan untuk mengimplementasikan fitur runtime bahasa umum seperti pengumpulan sampah (garbage collection).

Contoh awal adalah kompilasi Pascal oleh beberapa kompiler pada tahun 1970-an dan 1980-an menjadi apa yang disebut p-code (sejenis bytecode) yang akan berjalan pada VM p-code. Dua lingkungan VM modern yang menonjol adalah JVM, yang disebutkan sebelumnya dalam bab ini, dan Common Language Runtime (CLR) milik Microsoft yang merupakan bagian dari platform .NET-nya. Baik JVM maupun CLR ditargetkan oleh beberapa bahasa pemrograman populer. Misalnya, C#, F#, dan Visual Basic adalah bahasa yang umumnya menargetkan CLR, tetapi ada juga implementasi bahasa populer seperti Python dan Swift untuk CLR.

Mengapa implementasi bahasa ini dikompilasi menjadi bytecode untuk CLR, bukan kode mesin? Setelah dikompilasi, bytecode tersebut dapat dijalankan di platform mana pun

yang memiliki CLR terinstal. Itu adalah semacam portabilitas instan pasca-kompilasi. Selain itu, VM canggih seperti CLR akan menyediakan layanan bahasa seperti pengumpulan sampah (garbage collection), multithreading, dan mekanisme keamanan. Terakhir, ketika VM seperti CLR mengkompilasi kode perantara menjadi kode mesin secara just-in-time (JIT), ia akan menerapkan optimasi yang tidak perlu dipikirkan oleh penulis bahasa.

Di luar mesin abstrak yang digunakan sebagai runtime bahasa, istilah mesin virtual juga secara membingungkan digunakan untuk merujuk pada seluruh implementasi perangkat keras dalam perangkat lunak—dengan kata lain, emulator. Membangun emulator adalah subjek bab berikutnya.

LATIHAN SOAL

1. Cobalah mengukur kinerja kode interpreter opcode utama menggunakan tiga metodologi berbeda: pernyataan match yang sudah diimplementasikan, serangkaian pernyataan if...elif, dan tabel lompatan. Tentukan metode mana yang tercepat menggunakan profiler atau timer sederhana. Anda mungkin perlu menonaktifkan kode pengaturan waktu di loop eksekusi utama untuk melakukan ini, atau Anda dapat melakukannya menggunakan serangkaian pengujian unit.
2. Ada versi CHIP-8 yang sedikit diperluas, yang dikenal sebagai SCHIP (Super-Chip). Ini membutuhkan implementasi beberapa opcode tambahan dan mengubah beberapa elemen dari VM CHIP-8 asli, seperti resolusinya. Cari dokumentasi untuk SCHIP dan coba ubah VM CHIP-8 kita menjadi VM SCHIP. Kemudian, coba mainkan beberapa game SCHIP!
3. Cobalah menulis game yang sangat sederhana yang hanya menampilkan beberapa huruf di layar menggunakan instruksi kode mesin CHIP-8. Anda akan membutuhkan editor heksadesimal untuk melakukan ini. Sangat menyenangkan melihat kode biner yang Anda tulis berjalan di VM yang Anda pahami.

BAB 6
MENGEMULSI KONSOL GAME NES

Pada bab ini, kita akan membangun emulator terbatas untuk konsol video game tahun 1980-an yang sangat disukai, Nintendo Entertainment System (NES). Dengan kata lain, kita akan membuat perangkat lunak yang berpura-pura menjadi perangkat keras NES sehingga perangkat lunak yang ditulis untuk NES dapat "ditipu" agar berjalan di platform modern. Membangun proyek ini memberikan pengalaman dalam mengemulasi sistem komputer lengkap. Meskipun NES adalah komputer yang relatif sederhana, ia memiliki semua komponen dasar yang sama (mikroprosesor, memori, grafis, dan sebagainya) yang dimiliki proyek emulasi yang lebih kompleks.

Dan tidak seperti CHIP-8 di Bab 5, kita akan mengemulasi lebih dari sekadar spesifikasi perangkat lunak—kita akan mensimulasikan perangkat keras nyata!

Emulator kami tidak akan menjadi implementasi ulang 100 persen akurat dari perangkat keras aslinya. Kami akan membuat beberapa penyederhanaan untuk membuat pembuatan emulator dapat dikelola dalam satu bab buku. Terlepas dari penyederhanaan ini, emulator kami masih akan mampu memainkan beberapa game NES dasar, termasuk beberapa game gratis dan sumber terbuka untuk hobi yang disertakan dalam repositori kode sumber buku. Kami tidak akan menguji emulator kami dengan game komersial apa pun, meskipun akan mampu memainkan beberapa game sederhana. Motivasi kami adalah untuk belajar tentang emulator, bukan untuk mencapai kompatibilitas game penuh. Penulis akan menyerahkannya sebagai latihan bagi pembaca untuk lebih meningkatkan kompatibilitas game emulator.

Kami akan membangun emulator dalam Python murni, yang pada saat penulisan ini belum cukup cepat pada PC modern untuk mengemulasi NES dengan kecepatan penuh. Kode yang kami hasilkan dapat ditingkatkan dengan Cython, ekstensi C, atau jenis lapisan kode asli lainnya untuk berjalan dengan kecepatan penuh. Ini juga akan diserahkan sebagai latihan bagi pembaca.

Ini adalah proyek paling menantang dalam buku ini. Penulis berasumsi dalam bab ini bahwa Anda sudah memiliki pengalaman menyelesaikan proyek-proyek di Bab 1, 2, dan terutama 5. Anda setidaknya harus menyelesaikan proyek CHIP-8 dari Bab 5 sebelum memulai bab ini, tetapi konsep yang dijelaskan di hampir semua bab sebelumnya muncul dalam proyek ini.

Jika Anda memiliki kekhawatiran hukum tentang penyelesaian proyek ini, teliti hukum di tempat Anda tinggal atau konsultasikan dengan pengacara. Perhatikan bahwa informasi yang digunakan untuk mengembangkan emulator NES dalam bab ini tidak didasarkan pada dokumen hak milik Nintendo. Ingatlah bahwa sebagian besar file ROM untuk game komersial dilindungi oleh undang-undang hak cipta. Tidak perlu mengunduh file ROM yang dilindungi

untuk menguji emulator Anda, karena repositori kode sumber buku ini mencakup beberapa ROM nonkomersial yang berada di bawah lisensi sumber terbuka atau dirilis ke domain publik.

6.1 GAMBARAN UMUM NES

NES adalah salah satu konsol video game terlaris sepanjang masa. Pertama kali dirilis di Jepang pada tahun 1983 sebagai Famicom, NES memikat generasi gamer global dengan perilisan internasionalnya pada tahun 1985. Pada saat debutnya, industri video game baru berusia sekitar satu dekade. Mikroprosesor dan perangkat keras lainnya di konsol video game masih cukup primitif. Meskipun demikian, NES berhasil menampilkan 60 FPS sprite dan latar belakang berwarna-warni, memainkan musik chip tune yang menarik, dan menjadi platform untuk beberapa game paling ikonik sepanjang masa.

Banyak spesifikasi perangkat keras untuk NES dan sebagian besar informasi tentang fungsinya yang penulis sajikan dalam bab ini berasal dari NesDev, komunitas pengembang homebrew NES dan penulis emulator yang ada di https://www.nesdev.org. Meskipun penulis pikir bab ini adalah gambaran umum dan tutorial terbaik tentang cara menulis emulator NES, penulis sangat menyarankan untuk memeriksa situs web NesDev untuk detailnya. Sebagai pengganti kumpulan kutipan besar dari situs tersebut, penulis menyajikan catatan penting ini kepada Anda. Selain itu, beberapa gambar dalam bab ini juga berasal dari NesDev dan telah dirilis ke domain publik.

Perangkat Keras

Unit pemrosesan pusat (CPU) di NES adalah klon dari mikroprosesor MOS Technology 6502, yang diproduksi oleh Ricoh, yang beroperasi pada kecepatan sedikit di bawah 2 MHz. 6502 adalah mikroprosesor yang sama yang ada di komputer rumahan populer pada saat itu, seperti Apple II dan Commodore 64.

Dibangun dari sekitar 3.500 transistor, 6502 adalah mikroprosesor yang sangat sederhana; misalnya, ia tidak memiliki instruksi untuk perkalian dan pembagian. Operasi aritmatika tersebut harus diimplementasikan dalam perangkat lunak dari banyak instruksi yang lebih sederhana seperti penjumlahan, pengurangan, dan pergeseran bit. Ketika Anda memikirkan betapa lambat dan sederhananya 6502 dibandingkan dengan mikroprosesor modern, sungguh luar biasa apa yang telah dicapai dengannya.

6502 pada NES dikombinasikan dengan chip audio dalam paket yang sama. Dalam istilah pengembangan NES, chip ini dikenal sebagai unit pemrosesan audio (APU). APU mendukung lima saluran suara yang berbeda. Agar lebih sederhana, kami tidak akan mengimplementasikan APU di emulator kami.

Saat mengimplementasikan audio, pengaturan waktu sangat penting, dan emulator kami tidak akan akurat dalam hal pengaturan waktu. CPU NES dapat mengakses 2KB RAM bawaan di mesin tersebut. Ya, Anda membaca dengan benar. Memori kerja CPU NES hanya 2KB, bahkan tidak cukup memori untuk menyimpan teks bagian bab ini. RAM-nya juga lebih sedikit daripada yang biasanya dimiliki sistem CHIP-8, yang keluar pada dekade sebelumnya. Beberapa cartridge menyertakan RAM tambahan.

Kunci kinerja NES adalah unit pemrosesan gambar (PPU), yang diproduksi oleh Ricoh sebagai 2C02, berdasarkan desain sebelumnya oleh Texas Instruments. PPU tidak hanya dapat menghasilkan grafik latar belakang ubin, tetapi juga memiliki dukungan bawaan untuk sprite. Fiturnya meliputi memori 2KB untuk informasi ubin latar belakang, memori 256 byte untuk melacak hingga 64 sprite, dan 28 byte untuk menyimpan informasi palet warna. NES mendukung 54 warna berbeda, tetapi hanya 25 yang dapat digunakan secara bersamaan. PPU bahkan memiliki dukungan primitif untuk deteksi tabrakan.

CPU berkomunikasi dengan APU dan PPU melalui register perangkat keras yang dipetakan ke memori. Ini adalah alamat memori tertentu yang, ketika ditulis, dapat memodifikasi operasi chip perangkat keras lain atau, ketika dibaca, akan memberikan pembaruan pada flag atau status chip lain saat ini. Misalnya, CPU dapat menulis data ke register PPU untuk mengubah lokasi sprite. Kemudian, ia dapat membaca dari register PPU yang berbeda untuk melihat apakah sprite bertabrakan dengan sesuatu. CPU juga memiliki register yang dipetakan ke memori untuk membaca dari pengontrol game.

Izinkan penulis membuat konsep register yang dipetakan ke memori ini lebih konkret dengan sebuah contoh. Ketika sebuah game perlu memeriksa status joypad pertama (pengontrol pemain 1), ia menggunakan register yang dipetakan ke memori. Register itu berada di alamat memori `0x4016.` Jika gim membaca dari `0x4016,` gim akan mendapatkan kembali 1 byte yang menunjukkan apakah tombol tertentu pada joypad ditekan. Alamat memori `0x4016` tidak dapat digunakan untuk hal lain; alamat tersebut terhubung secara perangkat keras ke jalur yang berasal dari joypad. Agar berfungsi dengan benar, emulator kita perlu melakukan hal yang tepat ketika beberapa alamat memori khusus ini dibaca atau ditulis. Beberapa hanya dapat dibaca, beberapa hanya dapat ditulis, dan beberapa dapat dibaca atau ditulis. Ini adalah register perangkat keras yang dipetakan ke memori.

Bagian penting lain dari perangkat keras NES adalah kartrid gim. Kartrid gim, terutama yang awal, sebagian besar terdiri dari chip ROM besar yang berisi grafik dan kode program untuk sebuah gim. Kartrid gim juga dapat memiliki RAM (kadang-kadang didukung oleh baterai sehingga status gim dapat disimpan), chip logika sederhana, dan bahkan yang disebut pengalihan bank untuk memungkinkan total memori (RAM + ROM) yang lebih besar daripada yang dapat diakses oleh 6502 dalam konfigurasi default-nya. Oleh karena itu, angka RAM 2KB tersebut sedikit menyesatkan karena kode program akan berada di cartridge ROM, bukan di memori konsol game yang terbatas. Sebaliknya, memori 2KB tersebut hampir seluruhnya dapat digunakan untuk menyimpan status. Cartridge game awal biasanya terdiri dari 24KB hingga 40KB ROM, sedangkan cartridge game selanjutnya mungkin memiliki sekitar 128KB ROM dan 8KB RAM. Cartridge utama terbesar untuk NES memiliki memori 768KB.

Perangkat Lunak

NES tidak memiliki BIOS atau sistem operasi. Perangkat keras NES yang polos tidak memiliki perangkat lunak yang menyertainya. Semua perangkat lunak disediakan oleh cartridge game. Program pada cartridge game akan langsung mengontrol CPU, PPU, dan APU, tanpa lapisan abstraksi di antara mereka dan perangkat keras.

Gim NES biasanya ditulis dalam bahasa assembly 6502. Itu mungkin terdengar rumit, tetapi itu lazim untuk era tersebut; sebagian besar program yang perlu berkinerja tinggi pada komputer pribadi atau konsol gim diprogram dalam bahasa assembly hingga awal tahun 1990-an. Selain assembler, alat pengembangan sering kali dibuat sendiri. Tidak ada NES Game Maker yang dapat diunduh.

Faktanya, ini adalah era sebelum unduhan menjadi hal yang umum. NES adalah beberapa generasi konsol sebelum konektivitas internet ada. Konsol mainstream pertama dengan modem bawaan adalah Sega Dreamcast, yang keluar pada akhir tahun 1990-an. Apa yang dikirimkan pada cartridge NES adalah versi final gim; tidak akan ada pembaruan. Jika ada bug, maka ada bug, jadi gim harus hampir sempurna untuk versi 1.0. Bandingkan itu dengan gim tipikal yang Anda beli hari ini, di mana pengembang sering mengerjakan patch utama pertama bahkan sebelum gim tersebut dirilis. Dahulu, dibutuhkan tingkat perhatian terhadap detail yang jauh lebih tinggi, tetapi di sisi lain, gim-gim tersebut jauh kurang kompleks dibandingkan saat ini.

Sungguh menakjubkan untuk berpikir bahwa dalam lingkungan primitif ini, beberapa gim paling berpengaruh dan mendefinisikan genre sepanjang masa dikembangkan. Kemampuan teknis yang dibutuhkan dari para programmer dalam tim berada di ceruk yang sangat berbeda dari apa yang ditempati oleh pengembang gim saat ini. Sebagian besar gim saat ini dibangun menggunakan kerangka kerja atau mesin yang sudah dikemas seperti Unreal atau Unity. Pengembang dapat menghabiskan sebagian besar waktu mereka untuk menulis mekanisme khusus gim.

Para pengembang NES harus menulis mesin mereka sendiri dalam bahasa assembly. Mereka harus secara langsung mengelola APU untuk memainkan setiap suara dan PPU untuk menampilkan setiap grafik, dan mereka harus memanfaatkan setiap siklus CPU untuk menyelesaikan apa pun. Perusahaan yang lebih besar membangun kerangka kerja dan alat internal mereka sendiri yang dapat digunakan kembali dari satu judul ke judul lainnya, tetapi para programmer masih bekerja pada tingkat yang relatif rendah.

6.2 PENGEMBANGAN EMULATOR

Saatnya untuk menulis beberapa kode. Namun sebelum itu, ada catatan tentang arah dan pengelolaan ekspektasi: emulator yang sedang kita tulis telah disederhanakan di setiap sudutnya. Seperti yang disebutkan sebelumnya, emulator ini tidak akan kompatibel dengan banyak game, karena PPU yang sangat disederhanakan. Emulator ini juga tidak akan memiliki suara, karena kita tidak mengimplementasikan APU. Dan emulator ini tidak akan berjalan cukup cepat untuk memainkan game pada kecepatan yang seharusnya. Namun, emulator ini akan menjalankan game sungguhan, dan game tersebut akan dapat dimainkan. Pekerjaan kita di sini juga akan memberikan fondasi yang kuat untuk melakukan perbaikan dan menambahkan lebih banyak fitur jika Anda memilihnya.

Perencanaan Struktur

Rencana umum untuk eksekusi emulator tidak jauh berbeda dengan VM CHIP-8 dari Bab 5. Seperti halnya CHIP-8, kita akan membaca setiap instruksi satu per satu dari file ROM

dan menginterpretasikannya. Seperti halnya CHIP-8, kita akan menggunakan Pygame untuk menampilkan grafik dan menangani input pengguna. Seperti halnya CHIP-8, kita akan memiliki satu loop besar yang mengambil setiap instruksi dan menanggapi setiap kejadian. Namun, struktur kodenya akan lebih canggih. Secara khusus, kita akan membagi emulator menjadi tiga kelas, masing-masing mewakili satu komponen fisik perangkat keras. Kita akan memiliki kelas untuk CPU, PPU, dan cartridge. Berikut adalah uraian setiap file yang akan kita tulis dan tujuannya:

__main.__py	Menangani argumen baris perintah dan mengimplementasikan loop emulator utama, yang mengirimkan instruksi, menampilkan grafik, dan menanggapi input pengguna.
rom.py	Membaca file ROM dan berpura-pura menjadi cartridge.
cpu.py	Memelihara status CPU, menginterpretasikan instruksi, dan menangani akses memori utama.
ppu.py	Mengelola status PPU dan menggambar latar belakang dan sprite.

Kita akan membahas file-file ini sesuai urutan yang tercantum di sini.

Membuat Loop Utama

File utama kita (*__main__.py*) adalah tempat berbagai komponen sistem (CPU, PPU, cartridge) berkumpul. "Loop run"-nya memberikan kehidupan pada emulator dengan menjaga agar semuanya terus berjalan dan berkoordinasi antara berbagai komponen, mendelegasikan ke Pygame sesuai kebutuhan untuk menampilkan grafik dan membaca input pengguna. Fungsi `run()` menerima objek ROM dan nama file ROM sebagai argumen. Dalam cuplikan pertama kita, kita menginisialisasi Pygame, mendapatkan jendela di layar, dan membuat objek CPU dan 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
```

Seperti yang ditunjukkan oleh panggilan ke konstruktornya, baik PPU maupun CPU perlu mengakses ROM. CPU perlu membaca instruksi program, dan PPU perlu membaca data grafis. CPU juga perlu mengakses PPU karena ketika alamat memori tertentu dibaca atau ditulis, alamat tersebut sebenarnya merupakan proksi untuk register PPU.

Variabel ticks melacak berapa banyak siklus yang telah dijalankan CPU. Untuk setiap siklus CPU, PPU berjalan tepat tiga kali. Dengan kata lain, PPU memiliki kecepatan clock tiga kali lebih cepat daripada CPU, jadi sementara CPU sekitar 1,8 MHz, PPU sekitar 5,4 MHz. Kode kita perlu mensimulasikan ini, jadi pada cuplikan berikutnya, yang merupakan loop game utama, kita melacak berapa banyak siklus (atau tick) yang dibutuhkan setiap instruksi CPU (instruksi yang berbeda membutuhkan jumlah siklus yang berbeda) dan kemudian menjalankan PPU selama tiga kali jumlah siklus tersebut:

```
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()
        # Draw, once per frame, everything onto the screen
        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
```

Di akhir setiap frame, grafik yang dilihat pengguna diperbarui melalui metode Pygame yang sama yang kita gunakan di Bab 5. Tetapi bagaimana kita tahu sebuah frame telah berakhir? NES memiliki resolusi 256 piksel lebar dan 240 piksel tinggi. Setiap baris piksel dikenal sebagai scanline, istilah yang berasal dari televisi tabung sinar katoda (CRT) yang akan dihubungkan ke NES. Pada NES asli, satu piksel akan diperbarui dengan setiap siklus PPU. Karena PPU berjalan tiga kali lebih cepat dari CPU (5,4 MHz versus 1,8 MHz), untuk setiap siklus CPU, PPU menggambar tiga titik. Ketika kita sampai pada titik ke-257 pada scanline ke-240, kita seharusnya sudah selesai dengan satu frame ❶.

Emulator NES yang sangat akurat mensimulasikan perilaku perangkat keras asli dengan melakukan apa yang seharusnya dilakukan PPU setiap siklus: menentukan warna titik berikutnya. Kami menggunakan teknik yang jauh lebih sederhana, yaitu hanya menggambar semua ubin dan sprite yang benar di tempat yang tepat sekali per frame. Dengan kata lain, alih-alih memikirkan satu titik setiap siklus, kami hanya memikirkan seperti apa tampilan seluruh layar sekali per frame. Meskipun teknik ini lebih cepat dan tidak mengharuskan kita

untuk meniru banyak detail tentang cara kerja internal PPU, teknik ini tidak akan berfungsi dengan setiap game. Game NES yang lebih canggih akan membuat perubahan pada grafis bahkan saat sebuah frame sedang digambar ke layar (yaitu, di antara scanline atau bahkan di antara titik).

Perhatikan bahwa PPU sebenarnya melakukan pemrosesan tambahan di antara scanline. Periode ini dikenal sebagai hblank. Yang lebih penting, PPU memiliki scanline tambahan di luar layar (hingga scanline 261, menghitung yang pertama sebagai scanline 0). Waktu selama pemrosesan scanline tambahan yang tidak dirender ini dikenal sebagai vblank. Selama vblank, aman bagi CPU untuk memodifikasi memori PPU mana pun, karena memori PPU tidak sedang diakses secara aktif untuk menampilkan scanline yang terlihat. PPU mengirimkan sinyal ke CPU ketika vblank dimulai (setelah selesai dengan semua scanline yang terlihat) untuk tujuan ini ❷. Sinyal tersebut adalah jenis interupsi non-maskable (NMI).

Anggap NMI sebagai interupsi pada program yang tidak dapat dihentikan. Dengan kata lain, ini adalah sinyal yang memberi tahu mikroprosesor, "Hentikan apa yang sedang Anda lakukan segera, karena kami sedang melakukan hal lain sekarang." Dalam kasus NES, hal lain tersebut adalah memperbarui tampilan grafis game. Setiap game NES memiliki handler NMI yang memperbarui grafis game selama vblank.

Bagian loop lainnya hanya menangani peristiwa:

```
for event in pygame.event.get():
    if event.type == pygame.QUIT:
        sys.exit()
    # Handle keyboard events as joypad changes
    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__":
    # Parse the file argument
    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)
```

Kita mengenali tombol-tombol tertentu yang ditekan sebagai padanan tombol pada joypad NES. Kita menandai tombol-tombol yang ditekan agar CPU dapat membacanya. File utama kita diakhiri dengan menangani argumen baris perintah untuk membaca file ROM. Pembacaan sebenarnya dilakukan oleh kelas `ROM`, yang akan kita bahas selanjutnya.

Mengemulasi Kartrid

Kartrid game NES sebagian besar terdiri dari chip ROM dalam cangkang plastik. Chip ROM tersebut menyimpan kode game dan aset grafis. Kode tersebut berada dalam chip ROM yang dikenal sebagai ROM PRG, dan grafisnya berada dalam chip ROM yang dikenal sebagai ROM CHR. Meskipun kartrid sebagian besar terdiri dari chip ROM, kartrid juga dapat memiliki lebih banyak komponen. Seperti yang telah disinggung sebelumnya, salah satu peningkatan yang paling umum adalah chip logika yang memungkinkan pengalihan bank, sebuah teknik untuk memiliki lebih banyak ROM daripada yang biasanya dapat diakses oleh NES, tetapi dialihkan dalam skema sehingga program dapat mengakses alamat yang dipetakan ke memori tertentu dan berkata, “Saya sudah selesai dengan 8KB pertama dari ROM CHR, tolong alihkan pembacaan memori penulis ke 8KB berikutnya.”

Beberapa kartrid bahkan melangkah lebih jauh dengan menyediakan RAM tambahan untuk melengkapi 2KB CPU utama yang minim. Ini dikenal sebagai RAM PRG. Beberapa kartrid bahkan memiliki baterai sehingga isi RAM tidak akan terhapus saat konsol dimatikan. Tidak ada cara lain untuk menyimpan data pengguna secara permanen di NES, karena tidak ada disk. RAM yang didukung baterai memungkinkan permainan yang lebih lama. Tidak ada yang ingin memainkan permainan selama 40 jam jika kemajuan mereka akan terhapus setelah mereka mematikan konsol mereka.

Semakin lama NES berada di pasaran, semakin canggih kartridnya. Desain kartrid canggih yang sama akan diproduksi untuk digunakan kembali di banyak game. Sebagai penulis emulator, untuk mendukung semua game, Anda perlu mendukung semua chipset berbeda yang dapat dimuat oleh kartrid. Namun, ada beberapa desain chipset yang sangat populer yang mencakup sebagian besar dari semua game.

Masing-masing desain chipset kartrid ini dikenal di dunia emulator NES sebagai mapper karena penggunaan utama chipset kartrid adalah untuk beralih antara bank memori yang berbeda, yang dalam istilah pemrograman dapat dianggap sebagai pemetaan alamat ke bank. Salah satu emulator NES pertama yang dikembangkan, iNES oleh Marat Fayzullin, mendefinisikan skema penomoran untuk banyak mapper. Selain itu, iNES mendefinisikan format file ROM yang digunakan oleh hampir semua emulator NES saat ini.

Tidak seperti CHIP-8, yang memiliki file ROM yang hanya terdiri dari memori game mentah, NES membutuhkan format file yang lebih canggih karena variabilitas kartrid game-nya. Secara khusus, format iNES mendefinisikan header yang harus kita perhatikan saat membaca file ROM. Untungnya, kita memiliki pengalaman sebelumnya dengan header berkat

pekerjaan kita dengan header MacBinary di Bab 3. Header format file iNES didefinisikan dalam Tabel 6-1.

Tabel 6-1: Header Format File iNES

Byte	Keterangan
$0-3$	Konstanta 0x4E45531A (ASCII "NES" diikuti oleh akhir file MS-DOS)
4	Ukuran ROM PRG dalam satuan 16KB
5	Ukuran ROM CHR dalam satuan 8KB (nilai 0 berarti papan menggunakan RAM CHR)
6	Flags 6: mapper, mirroring, baterai, trainer
7	Flags 7: mapper, VS/Playchoice, NES 2.0
8	Flags 8: Ukuran RAM PRG (ekstensi yang jarang digunakan)
9	Flags 9: Sistem TV (ekstensi yang jarang digunakan)
10	Flags 10: Sistem TV, keberadaan RAM PRG (ekstensi tidak resmi, jarang digunakan)
$11-15$	Ruang kosong yang tidak terpakai (seharusnya diisi dengan angka nol, tetapi beberapa penipu mencantumkan nama mereka di sini)

Seperti yang Anda lihat, sebagian dari header mendefinisikan nomor mapper game. Setiap byte flag dapat berisi beberapa bit flag individual, itulah sebabnya beberapa deskripsi mencantumkan beberapa hal. Emulator kita tidak akan menggunakan informasi apa pun di flag 8, 9, atau 10.

Format file iNES telah diperluas dengan format yang lebih baru yang dikenal sebagai NES 2.0. Sebagian besar bidang header iNES masih valid di NES 2.0, dengan informasi tambahan di byte 11 hingga 15 dan beberapa perubahan pada byte flag.

Emulator yang kita buat dalam bab ini hanya mampu memainkan game yang menggunakan mapper paling sederhana, mapper 0, yang dikenal sebagai NROM. Kartrid NROM tidak memiliki fitur pengalihan bank, sehingga paling mudah untuk diemulasi. Kartrid NROM akan selalu memiliki 16KB atau 32KB ROM PRG dan 8KB ROM CHR. Secara opsional dapat memiliki RAM PRG.

Dalam kode kita, kita mendefinisikan sebuah namedtuple bernama Header untuk menyimpan isi header file ROM, serta mendeklarasikan beberapa konstanta standar:

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
```

Konstruktor kelas ROM pertama-tama menggunakan fungsi unpack() dari modul pustaka standar struct untuk membaca header dari file ROM dan mendistribusikannya ke dalam bidang-bidang dengan ukuran yang sesuai, yang ditentukan oleh string format:

```
class ROM:
  def __init__(self, file_name: str | Path):
    with open(file_name, "rb") as file:
      # Read header and check signature "NES"
      self.header = Header._make(unpack("!LBBBBBBB5s",file.read(HEADER_SIZE)))
```

Metode kelas `._make()` pada `namedtuple` dapat digunakan untuk membuat instance dari `namedtuple` tersebut dari sebuah `iterable`, seperti yang kita dapatkan dari `unpack()`. Tipe item spesifik dalam string format `unpack()` tercantum dalam Tabel 6-2. Setiap item dalam string format sesuai dengan elemen header dari Tabel 6-1. Untuk detail lebih lanjut tentang string format.

Tabel 6-2: Karakter String Format untuk Modul `struct`

Item	# of bytes	Tipe C	Tipe Python
!	N/A	Menunjukkan bahwa apa yang mengikuti selanjutnya dalam format big-endian.	N/A
L	4	Unsigned long	int
B	1	Unsignet chat	int
5s	5	Char[]	byte

Setelah proses ini, self.header berisi bagian-bagian yang tepat dari header iNES dalam segmen-segmen yang diberi label dengan rapi. Selanjutnya, kita periksa beberapa informasi dari header tersebut:

```
if self.header.signature != 0x4E45531A:
  print("Invalid ROM Header Signature")
else:
  print("Valid ROM Header Signature")
# Untangle Mapper - one nibble in flags6 and one nibble in 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")
```

Setiap header iNES seharusnya dimulai dengan tanda tangan 4-byte yang sama. Sementara itu, nomor mapper dibangun dari bagian flag 7 dan 8. Emulator kami hanya berfungsi dengan game yang menggunakan mapper 0.

Berikut adalah sisa konstruktornya:

```
    self.read_cartridge = self.read_mapper0
    self.write_cartridge = self.write_mapper0
    # Check if there's a trainer (4th bit flags6) and read it
    self.has_trainer = bool(self.header.flags6 & 4)
    if self.has_trainer:
      self.trainer_data = file.read(TRAINER_SIZE)
    # Check mirroring from flags6 bit 0
    self.vertical_mirroring = bool(self.header.flags6 & 1)
    print(f"Has vertical mirroring {self.vertical_mirroring}")
    # Read PRG_ROM & CHR_ROM, in multiples of 16K and 8K, respectively
    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
```

Kode ini berkaitan dengan pengaturan properti lain dari cartridge game. Bagaimana cara kita membaca dan menulis darinya (ini dapat berbeda tergantung pada mapper, meskipun kita hanya mendukung mapper 0)? Apakah ia memiliki trainer (fitur esoterik yang akan kita abaikan)? Apakah ia menggunakan jenis mirroring tertentu untuk grafiknya? Terakhir, berdasarkan ukuran yang ditunjukkan oleh header, jumlah data yang sesuai untuk ROM PRG, ROM CHR, dan (opsional) RAM PRG dibaca.

Perhatikan bagaimana `read_cartridge` dan `write_cartridge` ditetapkan sebagai alias untuk metode `read_mapper0()` dan `write_mapper0()`. Jika kita mendukung lebih dari satu mapper, kita akan menanganinya secara berbeda. Seperti yang ada sekarang, berikut adalah definisi untuk metode 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
```

Dengan melihat `read_mapper0()`, Anda akan melihat tiga area memori yang berbeda pada cartridge. Alamat di bawah 0x2000 dipetakan ke CHR ROM, yang diakses langsung oleh PPU. CPU mengakses PRG ROM dengan alamat lebih besar dari atau sama dengan 0x8000, dan dapat membaca atau menulis ke PRG RAM (jika cartridge memilikinya) dengan alamat lebih besar dari atau sama dengan 0x6000 tetapi di bawah 0x8000.

Itulah bagian cartridge dari kode kita. Singkatnya, file ROM dikonversi menjadi area CHR ROM dan PRG ROM yang masing-masing dapat diakses oleh PPU dan CPU kita. Inilah mengapa kelas PPU dan CPU dalam emulator kita perlu dapat mengakses kelas ROM.

Mengemulasi CPU

CPU pada akhirnya dapat dianggap sebagai mesin keadaan terbatas yang canggih. Mereka mempertahankan keadaan dalam register mereka dan jumlah memori terbatas yang dapat mereka akses. Mereka melakukan transisi keadaan melalui instruksi yang dapat mereka tangani. Wawasan ini menjelaskan pekerjaan utama yang perlu dilakukan oleh emulator CPU kita: memelihara register, mengakses memori, dan memodifikasi register dan memori dengan benar berdasarkan instruksi.

6502 adalah salah satu inti CPU paling sederhana yang pernah diterima secara luas di industri, dan versi 6502 di NES bahkan lebih sederhana daripada 6502 standar. Ia tidak memiliki instruksi untuk BCD yang dimiliki sebagian besar 6502. Hanya ada 56 jenis instruksi berbeda yang perlu kita implementasikan agar CPU NES berfungsi, dan banyak di antaranya dapat diimplementasikan hanya dalam beberapa baris kode. Selain itu, 6502 hanya memiliki tiga register utama (A, X, dan Y) bersama dengan beberapa register khusus (SP, PC, dan berbagai flag). Satu-satunya kompleksitas nyata dalam 6502 berasal dari berbagai metode akses memori yang dapat digunakan oleh berbagai instruksi, tetapi kita akan mengabstraksikannya dalam fungsi pembantu.

Pengaturan Awal

Kode untuk implementasi 6502 kita dimulai dengan menyiapkan beberapa konstruksi dan konstanta pembantu:

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")
```

```python
@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
```

Enum `MemMode` mencantumkan semua berbagai skema akses memori yang berbeda di 6502. Pada beberapa mikroprosesor dasar, mengambil byte dari memori semudah menentukan alamat dan mendapatkan kembali byte yang tersimpan di sana. Misalnya, jika penulis mengatakan "read 0x1940," penulis akan mendapatkan kembali byte apa pun yang tersimpan di memori pada alamat 0x1940. 6502 dapat melakukan ini dengan mode memori `ABSOLUTE`-nya, tetapi ia memiliki mode memori lain yang berguna dalam situasi tertentu. Beberapa mode ini mengakses alamat memori yang dihitung secara langsung, bukan ditentukan secara harfiah.

Misalnya, mode `ABSOLUTE_X` menambahkan nilai dalam register X ke alamat yang diberikan dan mengakses lokasi memori yang dihasilkan. Dalam mode ini, jika kita mengeksekusi instruksi yang sama lagi setelah menaikkan X, kita akan secara otomatis membaca byte berikutnya di memori. Saat memprogram pada tingkat rendah, ini dapat menjadi kemudahan nyata dan bahkan meningkatkan kinerja jika perangkat keras dioptimalkan untuk mode akses tertentu. Kita akan membahas mode memori NES secara lebih detail nanti di bab ini—untungnya, banyak di antaranya cukup mirip satu sama lain.

Enum `InstructionType` mencantumkan semua jenis instruksi yang dapat ditangani oleh 6502. Beberapa jenis instruksi ini adalah untuk operasi BCD yang, seperti yang disebutkan sebelumnya, tidak dimiliki oleh versi NES dari 6502. Beberapa di antaranya adalah jenis instruksi "tidak resmi" yang sebenarnya tidak didokumentasikan sebagai bagian dari 6502

tetapi ditemukan keberadaannya melalui uji coba dan kesalahan. Sangat sedikit game yang menggunakannya. 56 sisanya adalah jenis instruksi yang akan kita implementasikan. Kita mencantumkan semua jenis instruksi yang mungkin dalam enum ini—bahkan yang belum diimplementasikan—karena kita akan menggunakan tabel yang dibuat secara otomatis dari semua 256 kemungkinan opcode 6502, dan kita ingin setiap entri dalam tabel memiliki nilai jenis instruksi yang valid.

Sebuah Instruksi mengacu pada salah satu dari 256 kemungkinan opcode yang dapat dipahami oleh 6502. Setiap instruksi memiliki informasi tentang tipenya (`type`), fungsi terkaitnya dalam program kita yang menanganinya (`method`), mode akses memorinya (`mode`), jumlah byte yang diharapkan (`length`), jumlah siklus CPU yang dibutuhkan untuk menjalankannya (`ticks`), dan jumlah siklus tambahan yang dibutuhkan untuk menjalankannya jika halaman memori dilewati saat dieksekusi (`page_ticks`). Halaman memori adalah bagian dari RAM yang dapat diakses oleh pengontrol memori secara berurutan dengan cepat ke bagian lain. Pada 6502, halaman memori berukuran 256 byte. Jika sebuah instruksi melewati batas 256 byte tersebut, eksekusinya mungkin membutuhkan waktu lebih lama.

Fakta bahwa setiap instruksi memiliki fungsi terkait untuk menanganinya merupakan petunjuk bahwa kita akan menggunakan desain yang cukup berbeda dari proyek CHIP-8. Untuk VM CHIP-8, kami menggunakan pernyataan pencocokan raksasa untuk memproses setiap instruksi, tetapi untuk 6502 dan rangkaian instruksinya yang sedikit lebih kompleks, kami akan menggunakan desain yang lebih bersih. Alih-alih mengaktifkan setiap instruksi, kita akan mencarinya dalam sebuah array berdasarkan opcode-nya dan kemudian mengeksekusi fungsi yang terkait. Desain semacam ini merupakan variasi dari pola umum yang dikenal sebagai tabel lompatan. Pada intinya, kita mengindeks ke dalam array instruksi berdasarkan opcode untuk menemukan fungsi mana yang akan dilompati.

Kelas Joypad mewakili keadaan joypad selama eksekusi program. CPU dapat langsung melakukan polling joypad melalui beberapa register yang dipetakan ke memori, jadi tampaknya tepat untuk menempatkan Joypad di sini dalam modul cpu. Ingat bahwa loop utama kita mengatur keadaan joypad berdasarkan peristiwa yang terdeteksi oleh Pygame. Berikut adalah rincian konstanta pembantu yang tersisa dalam daftar kode sebelumnya:

`STACK_POINTER_RESET`	Alamat memori yang ditunjuk oleh penunjuk tumpukan CPU pada awalnya.
`STACK_START`	Di mana tumpukan dimulai dalam memori, yang menariknya merupakan alamat yang berbeda dari `STACK_POINTER_RESET`.
`RESET_VECTOR`	Alamat dalam memori yang berisi alamat lain dalam memori tempat eksekusi program dimulai. ROM PRG dari setiap game NES memiliki semacam kode awal untuk memulai eksekusi di alamat yang tercantum di `RESET_VECTOR`.
`NMI_VECTOR`	Sama seperti RESET_VECTOR tetapi untuk NMI dan vblank. Ketika vblank terjadi, kontrol akan dipindahkan ke alamat dalam memori yang tercantum di `NMI_VECTOR`.

IRQ_BRK_VECTOR	Alamat untuk jenis interupsi yang kurang umum digunakan yang tidak akan memengaruhi game yang kita uji dengan program kita. Ini disertakan di sini untuk kelengkapan.
MEM_SIZE	Ukuran, dalam byte, dari RAM utama yang dapat diakses oleh CPU NES.

Selanjutnya, mari kita lihat bagian awal konstruktor kelas CPU kita, yang mengatur memori, register, dan variabel status yang dapat dikonfigurasi:

```
class CPU:
    def __init__(self, ppu: PPU, rom: ROM):
        # Connections to Other Parts of the Console
        self.ppu: PPU = ppu
        self.rom: ROM = rom
        # Memory on the CPU
        self.ram = array('B', [0] * MEM_SIZE)
        # Registers
        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)
        # Flags
        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
        # Miscellaneous State
        self.jumped: bool = False
        self.page_crossed: bool = False
        self.cpu_ticks: int = 0
        self.stall: int = 0 # number of cycles to stall
        self.joypad1 = Joypad()
```

Untuk lebih memahami kode pengaturan ini, mari kita telusuri lebih dalam register-register pada 6502. Tabel 6-3 mencantumkan semuanya.

Karena Python hanya memiliki satu tipe untuk semua bilangan bulat terlepas dari ukurannya, kita merepresentasikan semua register ini kecuali flag dengan tipe int. Alih-alih mengutak-atik bit individual untuk setiap flag dalam register status, kita membaginya menjadi Boolean terpisah menggunakan nomenklatur huruf yang umum dalam dokumentasi 6502. Ini adalah variabel anggota C, Z, I, D, B, V, dan N.

Sebagian besar diatur sebagai hasil dari operasi aritmatika, I diatur ketika program ingin tidak diinterupsi oleh sinyal IRQ, dan B diatur ketika flag didorong ke tumpukan setelah

instruksi break. Flag D, yang digunakan untuk kode BCD, tidak relevan dengan NES, karena NES tidak memiliki instruksi BCD.

Tabel 6-3: Register 6502

Nama	Ukuran (byte)	Kegunaan
A	1	Register utama yang digunakan untuk operasi aritmatika. Terkadang dikenal sebagai akumulator.
X	1	Register indeks, sering digunakan sebagai penghitung loop. Register ini juga dapat digunakan sebagai register serbaguna, meskipun tidak semua instruksi bekerja dengannya seperti halnya dengan A.
Y	1	Sama seperti X
PC	2	Penghitung program (Program Counter), yang melacak di mana dalam memori instruksi berikutnya yang akan dieksekusi berada. Ukurannya 2 byte karena 6502 dapat mengakses hingga 64KB memori (tanpa pengalihan bank).
SP	1	Penunjuk tumpukan (Stack Pointer), yang melacak di mana program saat ini berada di tumpukan. Karena hanya 1 byte, tumpukan dapat menampung maksimum 256 byte.
P	1	Register status atau flag. Bit-bit individualnya menunjukkan hal yang berbeda, seperti sesuatu tentang operasi aritmatika (misalnya, apakah hasilnya nol?) atau apakah terjadi break atau interupsi diaktifkan (IRQ).

Variabel `jumped` melacak apakah instruksi jump mengubah register PC, dan `page_crossed` digunakan untuk pencatatan saat mengakses memori di seluruh halaman memori, yang, seperti yang telah dibahas, dapat lebih mahal daripada mengakses memori yang berdekatan. CPU NES mungkin perlu menunggu sejumlah siklus tertentu agar beberapa tugas selesai. Inilah tujuan dari penundaan `(stall).` Dalam emulator kami, penundaan hanya digunakan ketika terjadi transfer akses memori langsung (DMA) untuk mengirimkan sejumlah data dari memori utama ke memori atribut objek (OAM), tempat PPU menyimpan informasi tentang sprite.

6.3 TABEL LOMPATAN

Selanjutnya, kita akan mendeklarasikan tabel lompatan, daftar yang berisi semua instruksi potensial yang dapat diproses oleh 6502. Karena 6502 menggunakan opcode 1-byte, dan ada 256 nilai yang mungkin untuk satu byte, maka secara potensial ada 256 instruksi yang berbeda. Nantinya, dalam metode `step(),` kita akan mengindeks daftar ini untuk mendapatkan instruksi spesifik dan fungsi yang sesuai untuk opcode tertentu yang kita dekode. Kita tidak akan benar-benar mengimplementasikan setiap instruksi (beberapa adalah BCD atau tidak resmi), jadi beberapa di antaranya dilampirkan ke metode `self.unimplemented().` Seluruh 256 baris tabel lompatan disertakan di sini untuk kelengkapan:

```
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),
]
```

Membuat tabel lompatan ini secara manual akan sangat membosankan.

Sebagai gantinya, penulis menulis skrip eksternal untuk secara otomatis membuat tabel dari sumber publik. Skrip tersebut adalah sebuah trik yang penulis buat untuk menghasilkan tabel, jadi penulis tidak memasukkannya ke dalam repositori. Namun, terkadang skrip cepat dan sederhana seperti itu dapat menghemat banyak waktu pengetikan!

Instruksi

Selanjutnya, kita perlu mendeklarasikan semua metode yang menghidupkan instruksi 6502. Seperti yang telah dibahas, kita memiliki 56 metode unik untuk diimplementasikan, mulai dari ADC hingga TYA, yang menangani tugas-tugas seperti aritmatika, aliran kontrol, dan sejenisnya.

Seperti halnya proyek CHIP-8, ini adalah tempat yang tepat bagi Anda untuk berhenti dan mencoba menulis beberapa metode sendiri sebelum melihat implementasi di sini. Untuk melakukan itu, Anda memerlukan referensi instruksi 6502 yang bagus. Ada banyak yang tersedia secara online, dan tautan https://nesdev.org yang disebutkan sebelumnya mengarah ke beberapa di antaranya. Referensi yang baik harus mencakup hal-hal berikut:

- Nama instruksi, termasuk mnemonik umumnya
- Kode operasi untuk berbagai bentuk instruksi
- Mode memori yang didukungnya
- Bendera apa, jika ada, yang dipengaruhi oleh instruksi
- Berapa banyak siklus yang dibutuhkan
- Register apa yang dioperasikannya
- Contoh dari apa yang dilakukannya

Jika Anda memilih untuk mengimplementasikan instruksi sendiri, Anda pertama-tama perlu melihat sisa kelas CPU untuk melihat metode pembantu apa yang tersedia untuk Anda. Ada metode untuk memodifikasi tumpukan, membaca dari memori, dan menulis ke memori, dan ada beberapa metode utilitas lainnya juga.

Anda akan menemukan bahwa banyak instruksi cukup sederhana. Misalnya, AND adalah operasi logika AND yang persis seperti yang Anda harapkan. Kita mengambil akumulator (`self.A`), melakukan operasi AND bitwise antara akumulator tersebut dan apa pun yang kita baca dari memori, lalu menyimpan hasilnya kembali ke akumulator:

```
def AND(self, instruction: Instruction, data: int):
    src = self.read_memory(data, instruction.mode)
    self.A = self.A & src
    self.setZN(self.A
```

Perhatikan bagaimana dua hal telah diabstraksikan. Membaca dari memori dilakukan dengan metode lain, `self.read_memory()`, yang menerima mode memori instruksi. Kita akan kembali ke implementasi metode itu nanti. Kedua, banyak instruksi berbeda memengaruhi flag, jadi kita memiliki metode seperti `self.setZN()` untuk menangani perubahan flag. Ini adalah prinsip klasik jangan ulangi diri sendiri (DRY).

Berikut ini adalah implementasi untuk semua 56 metode yang dibutuhkan. Kita menulis dalam Python apa yang akan dilakukan 6502 dalam perangkat keras, dan itu sebenarnya bukan hal yang rumit. Python memiliki operator untuk menyelesaikan sebagian besar tugas. Keterampilan lain yang paling membantu dalam pekerjaan semacam ini adalah pemahaman yang kuat tentang operator bitwise, karena ada beberapa tempat di mana instruksi secara eksplisit memintanya atau kita perlu memotong hasilnya untuk memastikan hasilnya masih 8 bit sehingga sesuai dengan register.

```
# Add memory to accumulator with carry
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)

# Bitwise AND with accumulator
def AND(self, instruction: Instruction, data: int):
    src = self.read_memory(data, instruction.mode)
    self.A = self.A & src
    self.setZN(self.A)

# Arithmetic shift left
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)

# Branch if carry clear
def BCC(self, instruction: Instruction, data: int):
    if not self.C:
        self.PC = self.address_for_mode(data, instruction.mode)
        self.jumped = True

# Branch if carry set
def BCS(self, instruction: Instruction, data: int):
    if self.C:
        self.PC = self.address_for_mode(data, instruction.mode)
        self.jumped = True

# Branch on result zero
def BEQ(self, instruction: Instruction, data: int):
    if self.Z:
```

```
            self.PC = self.address_for_mode(data, instruction.mode)
            self.jumped = True

    # Bit test bits in memory with accumulator
    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)

    # Branch on result minus
    def BMI(self, instruction: Instruction, data: int):
        if self.N:
            self.PC = self.address_for_mode(data, instruction.mode)
            self.jumped = True

    # Branch on result not zero
    def BNE(self, instruction: Instruction, data: int):
        if not self.Z:
            self.PC = self.address_for_mode(data, instruction.mode)
            self.jumped = True

    # Branch on result plus
    def BPL(self, instruction: Instruction, data: int):
        if not self.N:
            self.PC = self.address_for_mode(data, instruction.mode)
            self.jumped = True

    # Force break
    def BRK(self, instruction: Instruction, data: int):
        self.PC += 2

    # Push PC to stack
        self.stack_push((self.PC >> 8) & 0xFF)
        self.stack_push(self.PC & 0xFF)

    # Push status to stack
        self.B = True
        self.stack_push(self.status)
        self.B = False
        self.I = True
        # Set PC to reset vector
        self.PC = (self.read_memory(IRQ_BRK_VECTOR, MemMode.ABSOLUTE)) | \
                  (self.read_memory(IRQ_BRK_VECTOR + 1, MemMode.ABSOLUTE) << 8)
        self.jumped = True

    # Branch on overflow clear
    def BVC(self, instruction: Instruction, data: int):
        if not self.V:
            self.PC = self.address_for_mode(data, instruction.mode)
            self.jumped = True
```

```python
# Branch on overflow set
def BVS(self, instruction: Instruction, data: int):
    if self.V:
        self.PC = self.address_for_mode(data, instruction.mode)
        self.jumped = True

# Clear carry
def CLC(self, instruction: Instruction, data: int):
    self.C = False

# Clear decimal
def CLD(self, instruction: Instruction, data: int):
    self.D = False

# Clear interrupt
def CLI(self, instruction: Instruction, data: int):
    self.I = False

# Clear overflow
def CLV(self, instruction: Instruction, data: int):
    self.V = False

# Compare accumulator
def CMP(self, instruction: Instruction, data: int):
    src = self.read_memory(data, instruction.mode)
    self.C = self.A >= src
    self.setZN(self.A - src)

# Compare X register
def CPX(self, instruction: Instruction, data: int):
    src = self.read_memory(data, instruction.mode)
    self.C = self.X >= src
    self.setZN(self.X - src)

# Compare Y register
def CPY(self, instruction: Instruction, data: int):
    src = self.read_memory(data, instruction.mode)
    self.C = self.Y >= src
    self.setZN(self.Y - src)

# Decrement memory
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)

# Decrement X
def DEX(self, instruction: Instruction, data: int):
    self.X = (self.X - 1) & 0xFF
    self.setZN(self.X)
```

```
# Decrement Y
def DEY(self, instruction: Instruction, data: int):
    self.Y = (self.Y - 1) & 0xFF
    self.setZN(self.Y)

# Exclusive or memory with accumulator
def EOR(self, instruction: Instruction, data: int):
    self.A ^= self.read_memory(data, instruction.mode)
    self.setZN(self.A)

# Increment memory
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)

# Increment X
def INX(self, instruction: Instruction, data: int):
    self.X = (self.X + 1) & 0xFF
    self.setZN(self.X)

# Increment Y
def INY(self, instruction: Instruction, data: int):
    self.Y = (self.Y + 1) & 0xFF
    self.setZN(self.Y)

# Jump
def JMP(self, instruction: Instruction, data: int):
    self.PC = self.address_for_mode(data, instruction.mode)
    self.jumped = True

# Jump to subroutine
def JSR(self, instruction: Instruction, data: int):
    self.PC += 2

# Push PC to stack
    self.stack_push((self.PC >> 8) & 0xFF)
    self.stack_push(self.PC & 0xFF)

# Jump to subroutine
    self.PC = self.address_for_mode(data, instruction.mode)
    self.jumped = True

# Load accumulator with memory
def LDA(self, instruction: Instruction, data: int):
    self.A = self.read_memory(data, instruction.mode)
    self.setZN(self.A)

# Load X with memory
def LDX(self, instruction: Instruction, data: int):
    self.X = self.read_memory(data, instruction.mode)
```

```
        self.setZN(self.X)

    # Load Y with memory
    def LDY(self, instruction: Instruction, data: int):
        self.Y = self.read_memory(data, instruction.mode)
        self.setZN(self.Y)

    # Logical shift right
    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)

    # No op
    def NOP(self, instruction: Instruction, data: int):
        pass

    # Or memory with accumulator
    def ORA(self, instruction: Instruction, data: int):
        self.A |= self.read_memory(data, instruction.mode)
        self.setZN(self.A)

    # Push accumulator
    def PHA(self, instruction: Instruction, data: int):
        self.stack_push(self.A)

    # Push status
    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

    # Pull accumulator
    def PLA(self, instruction: Instruction, data: int):
        self.A = self.stack_pop()
        self.setZN(self.A)

    # Pull status
    def PLP(self, instruction: Instruction, data: int):
        self.set_status(self.stack_pop())

    # Rotate one bit left
    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)

# Rotate one bit right
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)

# Return from interrupt
    def RTI(self, instruction: Instruction, data: int):
    # Pull status out
    self.set_status(self.stack_pop())
    # Pull PC out
    lb = self.stack_pop()
    hb = self.stack_pop()
    self.PC = ((hb << 8) | lb)
    self.jumped = True

# Return from subroutine
def RTS(self, instruction: Instruction, data: int):
    # Pull PC out
    lb = self.stack_pop()
    hb = self.stack_pop()
    self.PC = ((hb << 8) | lb) + 1 # 1 past last instruction
    self.jumped = True

# Subtract with carry
def SBC(self, instruction: Instruction, data: int):
    src = self.read_memory(data, instruction.mode)
    signed_result = self.A - src - (1 - self.C)
    # Set overflow
    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)

# Set carry
def SEC(self, instruction: Instruction, data: int):
```

```
        self.C = True

    # Set decimal
    def SED(self, instruction: Instruction, data: int):
        self.D = True

    # Set interrupt
    def SEI(self, instruction: Instruction, data: int):
        self.I = True

    # Store accumulator
    def STA(self, instruction: Instruction, data: int):
        self.write_memory(data, instruction.mode, self.A)

    # Store X register
    def STX(self, instruction: Instruction, data: int):
        self.write_memory(data, instruction.mode, self.X)

    # Store Y register
    def STY(self, instruction: Instruction, data: int):
        self.write_memory(data, instruction.mode, self.Y)

    # Transfer A to X
    def TAX(self, instruction: Instruction, data: int):
        self.X = self.A
        self.setZN(self.X)

    # Transfer A to Y
    def TAY(self, instruction: Instruction, data: int):
        self.Y = self.A
        self.setZN(self.Y)

    # Transfer stack pointer to X
    def TSX(self, instruction: Instruction, data: int):
        self.X = self.SP
        self.setZN(self.X)

    # Transfer X to A
    def TXA(self, instruction: Instruction, data: int):
        self.A = self.X
        self.setZN(self.A)

    # Transfer X to SP
    def TXS(self, instruction: Instruction, data: int):
        self.SP = self.X

    # Transfer Y to 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.")
```

Meskipun sebagian besar instruksi cukup sederhana, penulis menemukan penanganan penambahan dengan carry (ADC) dan pengurangan dengan carry (SBC) agak rumit. Register utama 6502 hanya 8 bit, jadi carry akan sering terjadi dan Anda perlu mengatur flag dengan benar. Tetapi kita punya trik: tipe int Python tidak terbatas pada 8 bit. Oleh karena itu, kita dapat melakukan aritmatika seolah-olah kita bekerja dengan nilai int normal dan kemudian hanya melakukan modulo pada apa pun di atas 255.

6.4 METODE STEP()

Dengan implementasi untuk semua instruksi yang sudah ada, kita siap untuk melangkah melalui eksekusi kode mesin 6502 yang sebenarnya. Metode `step()` membaca opcode berikutnya di PC dan mengambil instruksi dari tabel lompatan. Berikut adalah awal dari metode tersebut:

```
  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))
```

Sebagian besar instruksi 6502 juga memiliki beberapa data yang menyertainya, dan jumlah byte dapat bervariasi. Misalnya, instruksi TAY (transfer A ke Y) tidak memerlukan data, jadi hanya 1 byte, tetapi instruksi apa pun yang membaca dari memori akan memerlukan data tambahan untuk menentukan alamat memori. Jumlah data yang akan dibaca ditentukan oleh `instruction.length`. Metode `step()` berlanjut sebagai berikut:

```
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}:
  # Branch instructions are +1 ticks if they succeeded
  self.cpu_ticks += 1
self.cpu_ticks += instruction.ticks
```

```
if self.page_crossed:
   self.cpu_ticks += instruction.page_ticks
```

Kita memanggil metode sebenarnya dari instruksi untuk mengeksekusi instruksi, kemudian menaikkan penghitung program sesuai kebutuhan. Kita akhiri dengan beberapa pencatatan mengenai siklus CPU (tick).

Akses Memori

Beberapa metode berikutnya yang akan kita tulis membantu dalam membaca dan menulis ke memori. Akses memori adalah salah satu area yang lebih rumit dari 6502 karena memiliki selusin mode akses memori yang berbeda. Metode `address_for_mode()` bertanggung jawab untuk menerjemahkan data yang terkait dengan instruksi ke alamat memori tertentu berdasarkan mode instruksi (`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 for zero-page wrapping in next two lines
         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 for zero-page wrapping in next two lines
         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
```

Untuk memahami kode ini, berikut adalah uraian mode akses memori 6502 dan hubungannya dengan data yang terkait dengan instruksi:

ABSOLUTE	Alamatnya adalah data.
ABSOLUTE_X	Register X ditambahkan ke data untuk membentuk alamat.
ABSOLUTE_Y	Register Y ditambahkan ke data untuk membentuk alamat.
ACCUMULATOR	Register A sedang digunakan, bukan memori. Kita menangani mode ini secara langsung dalam metode instruksi individual, sehingga tidak muncul di address_for_mode().
IMMEDIATE	data adalah item terakhir; kita sebenarnya tidak mengakses memori. Kita menangani mode ini secara langsung dalam metode untuk membaca dan menulis ke memori.
INDEXED_INDIRECT	Alamat 2-byte berada di RAM pada data + X.
INDIRECT	Alamat 2-byte berada di RAM pada data.
INDIRECT_INDEXED	Alamat pada data di RAM ditambahkan ke register Y untuk membentuk alamat akhir.
RELATIVE	data ditambahkan ke PC untuk membentuk alamat.
ZEROPAGE	Ini seperti ABSOLUTE, tetapi dalam 256 byte pertama memori (halaman nol).
ZEROPAGE_X	Ini seperti ABSOLUTE_X, tetapi dalam 256 byte pertama memori.
ZEROPAGE_Y	Ini seperti ABSOLUTE_Y, tetapi dalam 256 byte pertama memori.

Mode `ZEROPAGE` mungkin tampak berlebihan dibandingkan mode `ABSOLUTE`, tetapi ada optimasi yang bagus di sini: karena halaman nol dalam memori hanya 256 byte, ia hanya membutuhkan satu byte data untuk menentukan alamat di dalamnya. Ini menghemat satu siklus waktu CPU (byte alamat kedua tidak perlu diambil) saat mengeksekusi instruksi yang menggunakan halaman nol.

Oleh karena itu, instruksi dalam mode `ZEROPAGE` lebih cepat daripada instruksi lain yang mengakses memori. Bahkan, programmer 6502 terkadang memperlakukan 256 slot memori di halaman nol seperti register tambahan karena sangat cepat untuk dibaca dan ditulis. Ini membantu menutupi sedikitnya register aktual yang dimiliki 6502.

Sekarang kita dapat membentuk alamat memori, kita lebih dekat untuk membaca dan menulis memori, tetapi kita juga membutuhkan pengetahuan tentang berbagai wilayah memori NES—alamat mana yang dipetakan ke RAM, mana yang ke PPU, dan seterusnya. Penting untuk dicatat bahwa banyak wilayah memori NES berisi pencermian yang ekstensif.

Misalnya, 2KB pertama dalam peta memori, atau alamat hingga 0x800 dalam heksadesimal, dipetakan ke RAM CPU, tetapi alamat apa pun yang diakses di bawah 0x2000 dipetakan ke RAM yang sama karena 2KB diulang empat kali hingga 0x2000. Dengan kata lain, alamat 0x801 sama dengan alamat 0x001, begitu pula alamat 0x1001 dan 0x1801 (2 × 0x800 dalam heksadesimal adalah 0x1000, dan 3 × 0x800 adalah 0x1800).

Pencermian ini merupakan hasil dari kekhasan perangkat keras NES untuk mengurangi biaya. Misalnya, tidak semua jalur perangkat keras memori 6502, yang dapat Anda bayangkan sebagai kabel, diperlukan untuk mengakses RAM 2KB yang sedikit itu, sehingga beberapa jalur perangkat keras diabaikan begitu saja. Sebagai ilustrasi, 0x800 adalah 2.048 dalam desimal dan 100000000000 dalam biner.

Setiap digit dalam biner dapat dianggap sebagai sinyal yang dibawa oleh jalur perangkat keras. Untuk mengakses 2.048 alamat pertama, Anda hanya memerlukan 11 jalur perangkat keras, yang sesuai dengan 11 angka 0 di belakang angka kita. 11 jalur perangkat keras tersebut cukup untuk 2.048 nilai berbeda, nilai desimal 0 hingga 2.047. Jalur perangkat keras ke-12, yang diwakili oleh angka 1 dalam biner, dapat diabaikan. Dengan kata lain, 0x800 dipetakan ke 12 jalur perangkat keras, tetapi hanya 11 dari jalur perangkat keras tersebut yang benar-benar ada, sehingga 0x800 pada dasarnya sama dengan 0x0.

Tabel 6-4 menunjukkan peta memori CPU NES, seperti yang dilihat oleh emulator kita. Ini mencakup wilayah dan alamat yang dipetakan ke memori secara individual. Wilayah dan alamat yang diabaikan atau tidak diimplementasikan oleh emulator kita tidak ditampilkan. Tabel ini juga menunjukkan apakah suatu wilayah atau alamat dapat dibaca, ditulis, atau keduanya. Terakhir, tabel ini menyebutkan apakah ada pencermian (mirroring).

Terdapat banyak alamat dan detail yang hilang dari Tabel 6-4. Misalnya, emulator sederhana kita tidak memiliki APU, tetapi NES asli memiliki alamat yang dipetakan ke memori dalam rentang 0x4000 untuk APU dan perangkat I/O lainnya (seperti joypad kedua). Kita juga tidak menentukan register PPU individual antara 0x2000 dan 0x2007, yang akan kita bahas kembali saat mengimplementasikan PPU. Ruang memori cartridge dapat sangat bervariasi tergantung pada mapper. Itu di luar cakupan bagian ini. Terakhir, RAM CPU yang hanya 2KB sebenarnya memiliki dua subwilayah penting: wilayah halaman nol cepat dari 0x0000 hingga 0x00FF yang telah dibahas, dan ruang yang biasanya digunakan untuk tumpukan dari 0x0100 hingga 0x01FF.

Tabel 6-4: Peta Memori NES yang Disederhanakan

Alamat / Wilayah	Panjang	Keterangan	Read / Write	Mirroring??
0x0000–0x1FFF	0x800	Memory CPU Utama 2KB	RW	Ya, 0x800 pertama dicerminkan hingga 0x2000
0x2000–0x3FFF	0x8	8 PPU Registers	Bervariasi	0x2000 hingga 0x2007 dicerminkan setiap 8 byte

0x4014	0x1	Transfer data sprite melalui DMA	W	Tidak
0x4016	0x1	Status Joypad 1	RW	Tidak
0x6000–0xFFFF	Bervariasi berdasarkan Cardridge	Memori Kartrid	Bervariasi	Bervariasi

Sekarang setelah kita memahami bagaimana memori dibagi, kita dapat melihat metode untuk membaca dan menulisnya. Kita akan mulai dengan metode `read_memory()`. Metode ini menerima lokasi yang akan dibaca dan mode memori, dan mengembalikan satu byte (direpresentasikan sebagai int di Python) dari lokasi tersebut:

```
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)

    # Memory map at https://wiki.nesdev.org/w/index.php/CPU_memory_map
    if address < 0x2000: # main RAM 2KB goes up to 0x800
        return self.ram[address % 0x800] # mirrors for next 6KB
    elif address < 0x4000: # 2000-2007 is PPU, mirrors every 8 bytes
        temp = ((address % 8) | 0x2000) # get data from PPU register
        return self.ppu.read_register(temp)
    elif address == 0x4016: # joypad 1 status
        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 # unimplemented other kinds of IO
    else: # addresses from 0x6000 to 0xFFFF are from the cartridge
        return self.rom.read_cartridge(address)
```

Jika mode memori adalah IMMEDIATE, itu berarti data aktual yang terkait dengan instruksi adalah apa yang dimaksudkan untuk "dibaca." Ingat, kode kita diabstraksikan sampai pada titik di mana sebuah metode untuk satu instruksi seharusnya bekerja dengan mode memori apa pun. Dalam kasus mode IMMEDIATE, kita tidak perlu melakukan pencarian aktual apa pun, jadi kita hanya mengembalikan data yang terkait dengan instruksi. (Nama lokasi agak aneh di sini, tetapi itu adalah nama yang bagus untuk semuanya kecuali mode IMMEDIATE.) Jika tidak, lokasi yang terkait dengan instruksi dikonversi ke alamat memori menggunakan address_for_mode().

Setelah mendapatkan alamat memori, kita menggunakan serangkaian pernyataan if untuk menentukan dari mana memori sebenarnya harus dibaca, seperti pada Tabel 6-4. Kita memperhitungkan mirroring dengan menggunakan modulus. Tergantung pada wilayahnya, CPU, PPU, joypad, atau cartridge dapat diakses. NES memiliki cara yang aneh untuk membaca joypad. Setiap kali alamat 0x4016 dibaca, status tombol joypad yang berbeda akan dikembalikan. Oleh karena itu, Anda perlu melakukan delapan kali pembacaan untuk mengetahui status setiap tombol. Selanjutnya, mari kita lihat penulisan ke memori:

```
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)
    # Memory map at https://wiki.nesdev.org/w/index.php/CPU_memory_map
    if address < 0x2000: # main RAM 2KB goes up to 0x800
        self.ram[address % 0x800] = value # mirrors for next 6KB
    elif address < 0x3FFF: # 2000-2007 is PPU, mirrors every 8 bytes
        temp = ((address % 8) | 0x2000) # write data to PPU register
        self.ppu.write_register(temp, value)
    elif address == 0x4014: # DMA transfer of sprite data
        from_address = value * 0x100 # address to start copying from
        for i in range(SPR_RAM_SIZE): # copy all 256 bytes to sprite RAM
            self.ppu.spr[i] = self.read_memory((from_address + i),
                                               MemMode.ABSOLUTE)
        # Stall for 512 cycles while this completes
        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 # unimplemented other kinds of IO
    else: # addresses from 0x6000 to 0xFFFF are from the cartridge
        # We haven't implemented support for cartridge RAM
        return self.rom.write_cartridge(address, value)
```

Metode write_memory() cukup mirip dengan read_memory(). Metode ini menangani mode IMMEDIATE, kemudian menukar lokasi dengan alamat untuk mode lainnya dan menulis ke lokasi yang sesuai.

6.5 METODE PEMBANTU

Kita akan melengkapi kelas CPU dengan serangkaian metode pembantu, dimulai dengan tiga metode ini:

```
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)]
```

Banyak instruksi perlu mengatur flag nol (Z) dan negatif (N). Alih-alih mengulangi beberapa baris tersebut dalam metode instruksi, kita memiliki setZN(). Metode stack_push() menempatkan nilai baru pada tumpukan. Ini melibatkan penempatan nilai pada tumpukan di alamat tulis dan memindahkan penunjuk tumpukan. Demikian pula, stack_pop() mengambil nilai kembali dari penunjuk tumpukan, memindahkan penunjuk tumpukan, dan mengembalikan nilai tersebut.

Untuk kenyamanan, kelas CPU kita menyimpan berbagai flag status dalam tujuh variabel Boolean terpisah, tetapi 6502 yang sebenarnya memiliki satu register status 8-bit di mana setiap flag adalah satu bit. Instruksi BRK, PHP, PLP, dan RTI perlu bekerja dengan register status dalam format bit-sentrisnya, sehingga metode berikut, status() dan set_status(), menggunakan operasi bitwise untuk menerjemahkan antara kedua format tersebut:

```
@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)
```

Terakhir, kita memiliki metode untuk menangani apa yang terjadi ketika NMI dipicu, dan metode `log()` untuk debugging:

```
  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
    # Set PC to NMI vector
    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}"
```

Sebuah NMI mengirimkan eksekusi ke blok kode pada alamat yang ditentukan dalam `NMI_VECTOR`. Ketika sebuah NMI dipicu, mirip dengan instruksi BRK dan JSR, kita perlu meletakkan penanda agar kita dapat kembali ke tempat kita berada sebelum NMI membawa kita pergi. Itulah tujuan dari mendorong PC dan status ke tumpukan.

Mengimplementasikan CPU melibatkan penulisan banyak kode sesuai spesifikasi—spesifikasi yang panjang tetapi terdiri dari banyak bagian kecil yang relatif mudah dicerna. Bagian terakhir dari emulator yang perlu kita implementasikan adalah PPU, dan akan terasa sangat berbeda. Masih ada spesifikasi dengan banyak detail, tetapi detail-detail tersebut pada dasarnya hanya melakukan dua hal besar: menggambar ubin latar belakang dan menggambar sprite.

Memahami PPU

Mengimplementasikan PPU adalah bagian paling kompleks dari proyek emulasi NES. PPU bertanggung jawab untuk menggambar grafik di layar. Untuk tujuan emulator sederhana kita, kita dapat menganggap grafik memiliki dua aspek: latar belakang dan sprite.

Latar belakang adalah lapisan belakang grafis di mana masing-masing bagian biasanya tidak bergerak sama sekali atau hanya bergulir bersama sebagai satu kelompok. Perangkat keras NES menggambar latar belakang menggunakan ubin. Dalam permainan platform, misalnya, sebuah ubin mungkin merupakan bagian dari platform, bagian dari latar belakang

artistik (mungkin gunung), tangga, atau pintu. Elemen-elemen ini biasanya tidak dapat bergerak sendiri.

Sebaliknya, sprite adalah objek permainan individual yang dapat bergerak ke mana saja di layar secara independen. Bayangkan pemain dan musuh dalam permainan platform. NES memiliki perangkat keras khusus untuk menangani hingga 64 sprite 8×8 atau 8×16 (dalam piksel) di layar sekaligus.

Ada banyak cara berbeda untuk mengimplementasikan PPU. Yang paling akurat adalah mensimulasikan apa yang dilakukan PPU sebenarnya: menghasilkan setiap piksel layar, satu per satu. Jika implementasi ini dilakukan dengan benar, setiap game yang ditulis untuk NES seharusnya berfungsi dengan benar. Pendekatan ini juga yang paling intensif kinerja. Alternatif populer adalah mengimplementasikan grafis satu baris pemindaian pada satu waktu. Alih-alih melakukan pembaruan untuk setiap piksel, pembaruan terjadi ketika pemrosesan setiap scanline selesai.

Kami tidak akan melakukan salah satu dari hal tersebut. Seperti yang dibahas sebelumnya, kami akan mengimplementasikan PPU menggunakan pendekatan yang lebih sederhana, yaitu memperbarui seluruh layar satu frame per satu frame. Ini adalah teknik yang paling tidak akurat, karena jika game entah bagaimana mengubah grafis antar piksel atau antar scanline, pembaruan tersebut tidak akan muncul. Kompatibilitas akan tetap baik, tetapi tidak akan berlaku untuk semua game.

Meskipun metode kami paling jauh dari cara kerja PPU yang sebenarnya, ini adalah teknik yang paling berkinerja tinggi, karena membutuhkan pembaruan paling sedikit per frame (hanya satu pembaruan besar alih-alih banyak pembaruan kecil). Ini juga yang paling mudah secara konseptual, karena tidak memerlukan pemahaman semua detail tentang cara kerja PPU yang sebenarnya. Karena emulator kami ditulis dalam bahasa pemrograman yang relatif lambat, Python, dan dimaksudkan untuk sesederhana mungkin untuk tujuan demonstrasi, rendering per frame bisa dibilang pilihan terbaik.

Meskipun kita tidak perlu memahami setiap detail PPU untuk mengimplementasikan pendekatan frame-by-frame kita, kita tetap perlu memahami beberapa dasar tentang dari mana data untuk latar belakang dan sprite berasal. Kita akan membahasnya lebih lanjut di bagian selanjutnya. Beberapa informasi sudah tersebar di bagian-bagian sebelumnya dari bab ini, tetapi di sini semuanya dirangkai bersama dengan banyak ide dan detail baru untuk menjelaskan secara kohesif bagaimana PPU beroperasi.

CHR ROM

Data untuk ubin latar belakang dan sprite awalnya terletak di kartrid dalam wilayah memori yang dikenal sebagai CHR ROM. Ukuran memori ini bisa sangat bervariasi. Game-game awal biasanya hanya memiliki 8KB CHR ROM, tetapi game-game selanjutnya dengan mapper dapat memiliki ratusan kilobyte, dengan kemampuan untuk menukar wilayah 8KB lainnya untuk yang pertama agar kompatibel dengan harapan perangkat keras PPU (yang hanya dapat mengakses 8KB CHR ROM secara langsung).

Beberapa game mengganti CHR ROM dengan CHR RAM, yang dapat dimodifikasi selama pengoperasian game. Namun, sebagian besar game memiliki CHR ROM tetap. Jika

sebuah game memiliki CHR RAM, game tersebut harus memuat grafis ke dalam CHR RAM sesuai kebutuhan dari PRG ROM, alih-alih selalu ada di sana. Beberapa game langka memiliki CHR ROM dan CHR RAM di bank yang berbeda.

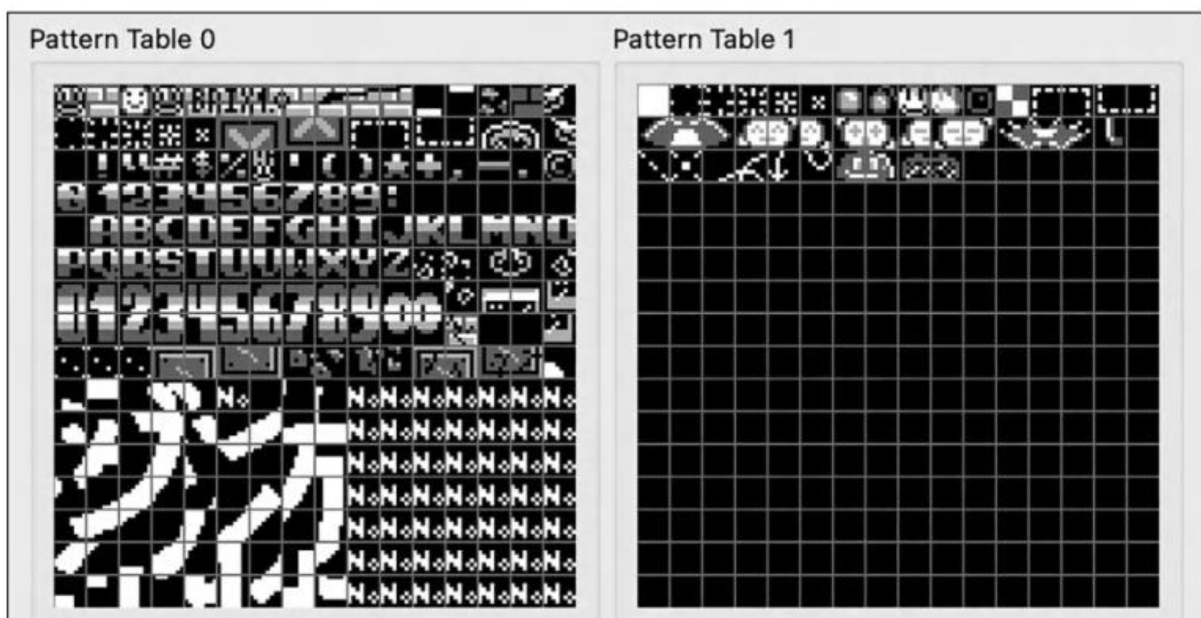
6.6 TABEL POLA DAN UBIN

Pada waktu tertentu, PPU dapat "dihubungkan" ke salah satu dari dua bagian 4KB dari ROM CHR pada kartrid. Bagian-bagian ini dikenal sebagai tabel pola. NES mengambil semua data grafisnya untuk bingkai tertentu dari tabel pola yang dipilih. Perhatikan bahwa beberapa dokumentasi merujuk pada semua 8KB ROM CHR yang dapat diakses sebagai satu tabel pola keseluruhan, alih-alih menyebut setiap bagian 4KB sebagai tabel pola terpisah.

Setiap tabel pola dibagi menjadi 256 ubin 16-byte, dan setiap ubin mendefinisikan potensi grafis untuk wilayah layar 8×8 piksel. Bersama-sama, ubin yang telah ditentukan sebelumnya dalam tabel pola mewakili semua hal berbeda yang mungkin Anda lihat dalam permainan. Misalnya, Gambar 6-1 menunjukkan tabel pola untuk game sumber terbuka BrickBreaker karya Aleff Correa, yang akan kita jalankan di emulator kita nanti di bab ini.

Anda dapat melihat bagaimana beberapa ubin dalam tabel pola mewakili sprite, beberapa mewakili teks, dan beberapa mewakili pola latar belakang. Banyak ubin dalam tabel pola kosong karena BrickBreaker tidak perlu menggunakannya (ia memiliki lebih banyak ruang di ROM CHR untuk aset grafis daripada yang sebenarnya dibutuhkan).

Gambar 6-1 dihasilkan menggunakan fitur PPU Viewer dari emulator FCEUX, yang memungkinkan Anda untuk melihat isi ROM CHR, terpisah dari gameplay sebenarnya. Menambahkan fitur debug seperti penampil tabel pola dapat sangat membantu saat menulis emulator. Misalnya, Anda dapat membandingkan output PPU Anda dengan output emulator yang sudah mapan seperti FCEUX.



Gambar 6-1: Tabel pola BrickBreaker, seperti yang ditampilkan oleh PPU Viewer emulator FCEUX

Seperti yang disebutkan, setiap ubin dalam tabel pola berukuran 16 byte. Dengan 16 byte yang mendefinisikan 8 × 8 = 64 piksel, hanya tersisa 2 bit per piksel, dan 2 bit hanya dapat mewakili empat nilai berbeda (00, 01, 10, 11). Oleh karena itu, PPU hanya mendukung empat warna dalam satu tile. Bahkan, salah satunya selalu merupakan warna latar belakang yang telah ditentukan sebelumnya atau transparan (00), sehingga pada dasarnya hanya ada tiga warna yang dipilih programmer yang dapat muncul dalam tile tertentu. Palet warna ini diatur untuk wilayah empat tile sekaligus dan dikontrol di bagian memori yang terpisah dari tabel pola itu sendiri. Inilah mengapa semua tile muncul dalam skala abu-abu pada Gambar 6-1; palet warna setiap tile tidak ditentukan oleh tabel pola. Kita akan kembali membahas bagaimana warna dipilih sebentar lagi.

Sayangnya, tile menjadi sedikit lebih rumit: nilai 2-bit yang mendefinisikan setiap piksel tidak disusun secara berurutan. Sebaliknya, bit ke-nol dari setiap piksel ditempatkan dalam 8 byte pertama dari ubin, dan bit pertama dari setiap piksel ditempatkan dalam 8 byte kedua dari ubin. Setiap 8 byte membentuk bidang bit, dan kedua bidang tersebut digabungkan, dua bit sekaligus, untuk menentukan warna setiap piksel.

Izinkan penulis menjelaskannya dengan cara lain. 8 byte pertama (bidang bit pertama) dari sebuah ubin dapat dianggap sebagai 64 bagian dari nilai warna untuk 64 piksel dalam ubin 8×8. Nilai-nilai tersebut ditempatkan secara berurutan. Bidang bit 8 byte kedua mendefinisikan 64 bagian lain dari nilai warna untuk 64 piksel yang sama. Nilai-nilai tersebut juga ditempatkan secara berurutan. Kedua bidang tersebut perlu digabungkan untuk mendapatkan 64 nilai warna akhir. Sebagai contoh, pertimbangkan 16 byte data ubin berikut:

```
    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
    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
```

Bagian-bagian yang cocok dari kedua bidang bit bergabung membentuk pola piksel yang ditunjukkan dalam daftar dan diilustrasikan pada Gambar 6-2: gambar pecahan 1/2.

	Plane 0									Plane 1	
41	01000001	00	01	00	00	00	00	00	11	00000001	01
C2	11000010	01	01	00	00	00	00	11	00	00000010	02
44	01000100	00	01	00	00	00	11	00	00	00000100	04
48	01001000	00	01	00	00	11	00	00	00	00001000	08
10	00010000	00	00	00	11	00	10	10	00	00010110	16
20	00100000	00	00	11	00	00	00	00	10	00100001	21
40	01000000	00	11	00	00	00	00	10	00	01000010	42
80	10000000	11	00	00	00	00	10	10	10	10000111	87

Gambar 6-2: Bagaimana sebuah ubin terbentuk dari dua bidang bit

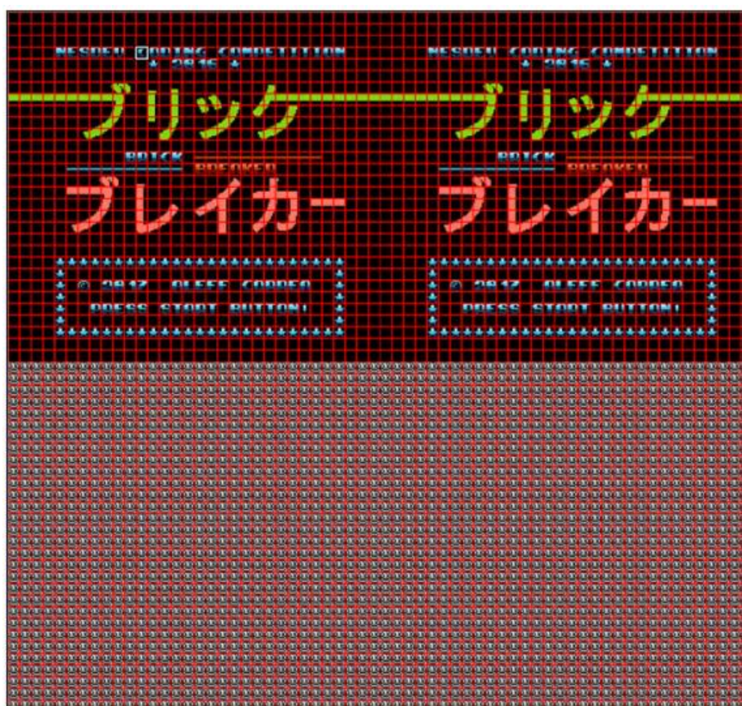
Jika ada angka 1 di bidang bit pertama dan angka 0 di bidang bit kedua, bit-bit tersebut bergabung membentuk 01, atau warna 1 dalam skema warna. Demikian pula, angka 0 dari bidang bit pertama dan angka 1 dari bidang bit kedua membentuk 10, atau warna 2, dan angka 1 di kedua bidang bit menghasilkan 11, atau warna 3.

Tabel Nama

Tabel nama adalah tempat ubin sebenarnya untuk latar belakang layar disusun. Seperti apa tampilan latar belakang layar saat ini? Terdiri dari ubin apa saja, dan warna apa yang digunakan di masing-masing ubin? Itulah tugas dari tabel nama dan tabel atribut yang menyertainya (dibahas selanjutnya).

Setiap tabel nama, yang mewakili satu layar permainan, memiliki lebar 32 ubin dan tinggi 30 ubin. Setiap ubin dalam tabel nama ditentukan oleh satu byte—indeks ubin dalam tabel pola saat ini. Dengan cara ini, ubin tabel pola dipetakan langsung ke tabel nama, sehingga Anda dapat menganggap tabel nama sebagai urutan spesifik ubin dari tabel pola.

Seberapa besar tabel nama? Nah, 32 × 30 = 960, jadi ada 960 lokasi dalam tabel nama. Dan setiap lokasi dalam tabel ditempati oleh indeks 1-byte, jadi tabel nama berukuran 960 byte. Sekarang kita memiliki cukup informasi untuk memahami mengapa NES memiliki resolusi 256×240. Setiap ubin memiliki lebar 8 piksel dan tinggi 8 piksel. Jika tabel nama mewakili latar belakang layar dan memiliki lebar 32 ubin, maka 8 × 32 = 256. Dan tinggi layar dalam piksel ditemukan dengan 8 × 30 = 240.



Gambar 6-3: Tabel nama layar judul BrickBreaker

PPU hanya memiliki memori 2KB, yang hanya cukup untuk dua tabel nama (dan tabel atributnya). Kedua tabel nama

tersebut dapat dicerminkan, sehingga total ada empat tabel nama logis. Gambar 6-3 menunjukkan tabel nama untuk layar judul BrickBreaker yang disusun.

Hanya satu tabel nama yang diisi dengan data yang bermakna pada Gambar 6-3. Anda dapat mengamati pencermian horizontal di mana separuh kanan gambar merupakan salinan dari separuh kiri.

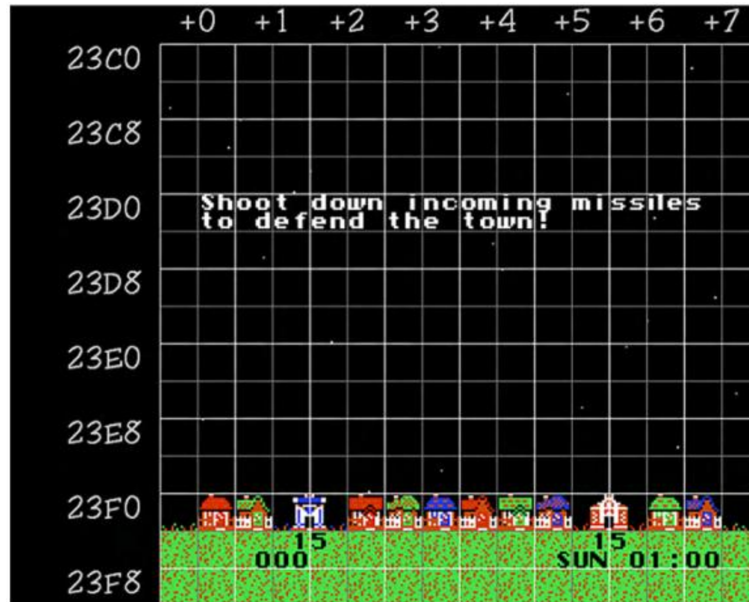
6.7 PALET WARNA DAN TABEL ATRIBUT

PPU memiliki memori palet untuk empat palet warna latar belakang yang berbeda. Ingat dari "Tabel Pola dan Ubin" pada halaman 178 bahwa setiap ubin terdiri dari piksel dalam salah satu dari empat warna, dengan salah satunya adalah warna latar belakang atau transparan. Itu berarti hanya tiga warna yang sebenarnya spesifik untuk sebuah ubin. Oleh karena itu, setiap palet dalam memori palet PPU mendefinisikan satu set tiga warna. (Kita akan membahas memori palet lebih lanjut di bagian berikutnya.)

Untuk menentukan palet warna mana yang berlaku untuk ubin mana, setiap tabel nama diikuti oleh tabel atribut. Tabel atribut berukuran 8×8, dan setiap entri di dalamnya hanya 1 byte. Bagaimana mungkin ada 960 ubin dan hanya 64 entri tabel atribut 1-byte? Setiap entri sebenarnya mendefinisikan warna untuk satu set area 2×2, dan setiap area tersebut mewakili kisi ubin 2×2. Oleh karena itu, setiap entri 1-byte dalam tabel atribut sebenarnya mencakup 16 ubin. Bagaimana cara kerjanya?

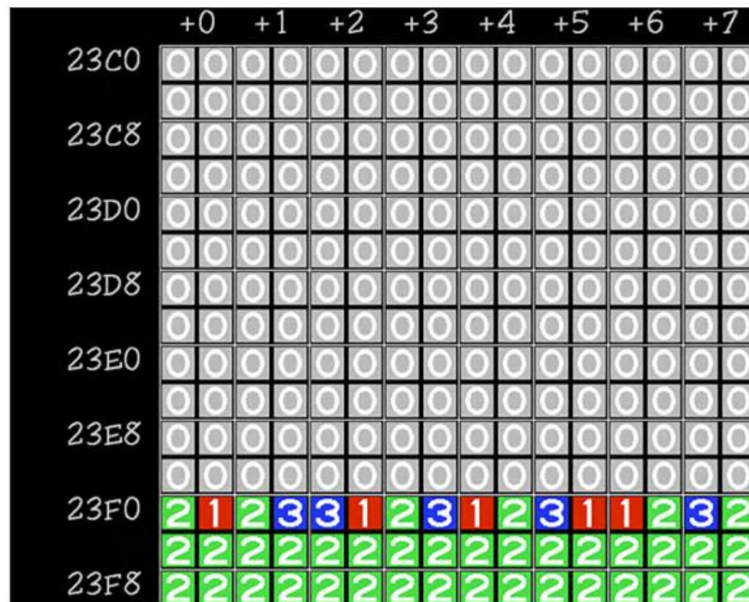
Setiap 2 bit dari entri 1-byte tersebut adalah untuk satu area, dan 2 bit dapat mewakili hingga empat nilai. Setiap nilai 2-bit adalah pemilih antara salah satu dari empat palet warna latar belakang yang berbeda. Oleh karena itu, setiap wilayah ubin 2×2 di layar dapat memiliki satu palet dari empat palet latar belakang yang mungkin. Itu berarti keempat ubin di setiap area harus menggunakan tiga warna yang sama (dan latar belakang/transparan).
Itu bukan tugas yang mudah. Seniman yang mengerjakan grafis untuk NES harus melukis area layar yang signifikan (area empat ubin) hanya menggunakan tiga warna non-latar belakang. Dan kemudian setiap area yang hanya menggunakan tiga warna non-latar belakang tersebut harus berbaur dengan area yang berdekatan yang mungkin menggunakan palet tiga warna yang berbeda.

Gambar 6-4, 6-5, dan 6-6, yang dibuat untuk https://nesdev.org, menampilkan tabel atribut yang sedang beraksi. (Jika Anda membaca ini dalam bentuk cetak, lihat direktori gambar dari repositori pendamping untuk versi berwarna dari gambar-gambar tersebut.) Pertama, Gambar 6-4 menunjukkan latar belakang sebuah game (Thwaite karya Damian Yerrick) yang dibagi menjadi area-area yang dapat dipilih warnanya oleh tabel atribut.



Gambar 6-4: Tata letak tabel atribut di layar pada Thwaite

Gambar 6-5 menunjukkan pemilihan palet sebenarnya (palet latar belakang mana, 0–3) untuk setiap area dari Gambar 6-4.



Gambar 6-5: Pemetaan palet warna tabel atribut di Thwaite

Terakhir, Gambar 6-6 menunjukkan empat palet latar belakang yang dapat dipilih. Terdapat beberapa tumpang tindih warna antar palet, yang memungkinkan area layar yang berbeda untuk berbaur satu sama lain.

3F00 +	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Values	0f	00	10	20	0f	17	16	20	0f	17	2a	20	0f	17	12	20
Colors																
Color set	0				1				2				3			

Gambar 6-6: Palet warna latar belakang di Thwaite

Setiap tabel atribut berukuran 64 byte, dan setiap tabel nama berukuran 960 byte. Satu tabel nama dan tabel atribut yang sesuai karenanya membutuhkan 960 + 64 = 1.024 byte, atau 1KB. Inilah cara 2KB RAM pada PPU terisi dengan dua kombinasi tabel pola/tabel atribut.

Memori Palet

Memori palet PPU memiliki ruang untuk empat palet latar belakang dan empat palet sprite. Seperti yang telah kita bahas, sebuah palet terdiri dari tiga warna (bersama dengan warna latar belakang/transparan). Palet ini dapat digunakan untuk mewarnai area latar belakang layar empat ubin (lihat bagian sebelumnya tentang tabel atribut) atau sebuah sprite. Dengan kata lain, setiap area latar belakang empat ubin dapat diwarnai menggunakan satu dari empat palet, dan setiap sprite dapat diwarnai menggunakan satu dari empat palet.

Sebuah palet didefinisikan menggunakan 3 byte. Setiap byte menentukan salah satu dari tiga warna palet, meskipun hanya 6 bit dari setiap byte yang sebenarnya digunakan untuk memilih warna. Karena 6 bit dapat memilih dari 64 nilai, ini memberi tahu kita bahwa para seniman NES hanya memiliki 64 warna untuk digunakan. Dalam praktiknya, 10 dari 64 warna yang mungkin ini pada dasarnya sama dengan hitam, jadi sebenarnya NES memiliki 54 warna.

Gambar 6-7 menunjukkan warna-warna ini untuk NES NTSC. Warna-warna tersebut sedikit berbeda untuk mesin NES PAL. (Negara-negara yang berbeda menggunakan standar video yang berbeda—NTSC di Amerika Utara versus PAL di sebagian besar Eropa dan Asia, misalnya—yang akan memengaruhi cara kerja PPU NES.)

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F
0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17	0x18	0x19	0x1A	0x1B	0x1C	0x1D	0x1E	0x1F
0x20	0x21	0x22	0x23	0x24	0x25	0x26	0x27	0x28	0x29	0x2A	0x2B	0x2C	0x2D	0x2E	0x2F
0x30	0x31	0x32	0x33	0x34	0x35	0x36	0x37	0x38	0x39	0x3A	0x3B	0x3C	0x3D	0x3E	0x3F

Gambar 6-7: Warna yang tersedia pada NES NTSC

Sebagai contoh, katakanlah area latar belakang Anda dilukis menggunakan palet latar belakang 1, dan palet latar belakang 1 menentukan warna 0x11, 0x0A, dan 0x3D. Itu berarti Anda dapat melukis area tersebut menggunakan nuansa biru, hijau, dan abu-abu, serta warna latar belakang.

Memori Atribut Objek (Object Attribute Memory/OAM)

OAM menyimpan informasi tentang sprite di layar. Sprite pada dasarnya adalah objek bergerak dalam permainan—misalnya, pemain, musuh, dan proyektil. NES mendukung 64

sprite sekaligus dan menggunakan 4 byte untuk mendeskripsikan masing-masing sprite, sehingga terdapat 256 byte OAM.

Gambar untuk sprite berasal dari ROM CHR yang sama pada kartrid dengan ubin latar belakang—yaitu, tabel pola. Namun, tidak seperti latar belakang, sprite tidak dibatasi pada lokasi ubin; sprite dapat digambar di posisi mana pun di layar. Setiap sprite juga dapat dibalik secara horizontal atau vertikal, ditempatkan di depan atau di belakang latar belakang, dan diwarnai menggunakan salah satu dari empat palet warna yang berbeda. Keempat byte untuk setiap sprite dan fungsinya dijelaskan secara singkat dalam Tabel 6-5.

Tabel 6-5: Spesifikasi Sprite dalam OAM

Byte #	Keterangan
0	Posisi sumbu y dari sprite
1	Indeks ke dalam tabel pola tempat data grafis sprite berada.
2	Atribut sprite termasuk palet (bit 0-1), depan atau belakang (bit 5), pembalikan horizontal (bit 6), dan pembalikan vertikal (bit 7)
3	Posisi sumbu x dari sprite

Mengapa byte untuk posisi sumbu y dan posisi sumbu x tidak bersebelahan adalah pertanyaan yang ingin penulis ajukan kepada pencipta NES. Mungkin ini berkaitan dengan urutan pengkabelan fisik jalur di PPU.

Pembuatan dan Pengaturan Waktu Frame

PPU di NES asli menggambar layar dari kiri atas ke kanan bawah, satu piksel pada satu waktu. Ini mencerminkan "senjata" di televisi CRT yang akan dihubungkan ke NES. Sinar elektronnya "ditembakkan" di belakang layar dari kiri ke kanan, satu garis pemindaian pada satu waktu dari atas ke bawah. NES NTSC menggambar seluruh layar 60 kali per detik (60 frame per detik, atau 60 FPS). NES PAL menggambar dengan kecepatan lebih lambat, yaitu 50 FPS. Satu frame adalah perjalanan senjata yang menyelesaikan seluruh layar. Seperti yang disebutkan sebelumnya dalam bab ini, senjata juga terkadang akan berada di luar layar untuk sementara selama periode hblank (antara garis pemindaian) dan vblank (antara frame). Oleh karena itu, ini adalah waktu teraman bagi program NES untuk mengubah memori tempat piksel dibaca.

Karena resolusi NES adalah 256×240, kita tahu bahwa setidaknya harus ada 256 titik yang digambar per scanline dan total 240 scanline. Ada scanline pra-render, yang akan kita sebut scanline 0. Kemudian, scanline 1 hingga 240 adalah scanline "terlihat" yang mewakili apa yang sebenarnya ditampilkan program di layar. Scanline 241 hingga 261 terjadi selama fase vblank. NMI, yang disebutkan sebelumnya dalam bab ini, oleh karena itu dipicu pada awal scanline 241 untuk memberi tahu program bahwa aman untuk mengubah memori PPU. Titik 0 hingga 255 mewakili 256 titik yang terlihat pada setiap scanline. Titik 256 hingga 340 adalah titik "diam" di mana fase hblank antara scanline terjadi.

Setiap titik mewakili satu siklus PPU. Jika Anda menghitungnya, 341 titik per garis pindai dikalikan 262 garis pindai berarti dibutuhkan 89.342 siklus PPU untuk menggambar setiap

frame. Ingatlah bahwa ada 3 siklus PPU untuk setiap 1 siklus CPU. Jika kita membagi siklus PPU dengan 3 dan membulatkan, kita mendapatkan 29.781 siklus CPU per frame. CPU di NES berjalan sekitar 1,79 MHz, atau 1.790.000 siklus per detik.

Jika Anda membagi 1.790.000 dengan 29.781, Anda mendapatkan angka yang mendekati 60. Itulah 60 frame per detik! Untuk membuat emulator NES yang akurat per siklus, penting untuk memahami detail bagaimana PPU menentukan warna apa yang akan digambar untuk setiap piksel, satu piksel pada satu waktu. Karena kita menggunakan pendekatan yang lebih sederhana namun kurang akurat, yaitu menggambar satu frame pada satu waktu, kita dapat mengabaikan detail tersebut karena berada di luar cakupan proyek kita. Yang kita ketahui berdasarkan pengaturan waktu yang baru saja dibahas adalah bahwa pada suatu titik selama setiap 89.342 siklus PPU, kita perlu menggambar seluruh frame.

Bagi saya, waktu yang logis untuk melakukan itu adalah ketika scanline yang terlihat selesai. Dalam kode yang akan kita lihat sebentar lagi, Anda akan melihat semua latar belakang dan semua sprite digambar sekaligus pada waktu scanline 240, titik 256 (titik terakhir yang terlihat pada scanline terakhir yang terlihat). Renderer sederhana kita tidak melakukan penggambaran kecuali selama siklus PPU tersebut, sekali per frame.

Mengimplementasikan PPU

Mengimplementasikan PPU kita dimulai dengan beberapa konstanta penting, termasuk berbagai ukuran memori, resolusi layar, dan palet warna lengkap yang tersedia:

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]
```

Warna ditentukan dalam nilai RGB heksadesimal. Berbagai situs web memiliki sedikit variasi tentang nilai warna yang tepat, dan ada perbedaan nyata dalam warna-warna ini antara berbagai variasi perangkat keras NES (NTSC versus PAL, misalnya). Oleh karena itu, game yang sama mungkin memiliki warna yang sedikit berbeda ketika dimainkan pada perangkat keras

NES yang berbeda atau emulator yang berbeda. Kelas PPU memiliki variabel instance untuk memori sprite (OAM), memori nametable, dan memori palet:

```
class PPU:
  def __init__(self, rom: ROM):
    self.rom = rom
    # PPU memory
    self.spr = array('B', [0] * SPR_RAM_SIZE) # sprite RAM
    self.nametables = array('B', [0] * NAMETABLE_SIZE) # nametable RAM
    self.palette = array('B', [0] * PALETTE_SIZE) # palette RAM
```

Bagian selanjutnya dari konstruktor kelas PPU mengatur nilai default untuk berbagai register PPU yang dapat diatur oleh program dan banyak variabel pembantu (kita akan membahas lebih detail tentang fungsi beberapa variabel ini seiring berjalannya implementasi kita):

```
# Registers
self.addr = 0 # main PPU address register
self.addr_write_latch = False
self.status = 0
self.spr_address = 0
# Variables controlled by PPU control registers
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
# Internal helper variables
self.buffer2007 = 0
self.scanline = 0
self.cycle = 0
# Pixels for screen
self.display_buffer = np.zeros((NES_WIDTH, NES_HEIGHT), dtype=np.uint32)
```

Selanjutnya, kita memiliki metode rendering, `step()`:

```
  def step(self):
    # Our simplified PPU draws just once per frame
    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 off, clear sprite zero, clear sprite overflow
        self.status |= 0b00011111

    self.cycle += 1
    if self.cycle > 340:
        self.cycle = 0
        self.scanline += 1
        if self.scanline > 261:
            self.scanline = 0
```

Setiap panggilan ke `step()` dari loop utama emulator mewakili satu siklus PPU. Karena strategi kita adalah menggambar semuanya sekali per frame, dan bukan untuk akurat hingga piksel atau scanline, `step()` sangat sederhana. Ia hanya menggambar latar belakang dan sprite sekaligus di akhir setiap bagian yang terlihat dari sebuah frame. Ia juga mengatur register status ketika vblank dimulai dan berakhir sehingga CPU dapat berkoordinasi dengan PPU. Terakhir, ia melakukan beberapa pencatatan untuk melacak scanline saat ini dan siklus saat ini pada setiap scanline.

Pekerjaan berat dilakukan oleh metode `draw_background()` dan `draw_sprites()`, yang dipanggil oleh `step()`. Kita akan melihat metode-metode ini selanjutnya.

Menggambar Latar Belakang

Kita memulai metode latar belakang `draw_background()` dengan menghitung alamat tabel atribut:

```
def draw_background(self):
    attribute_table_address = self.nametable_address + 960
```

Tabel atribut selalu berada tepat setelah tabel nama, dan ingat dari bagian sebelumnya dalam bab ini bahwa tabel nama berukuran 960 byte. Penulis juga menyebutkan sebelumnya dalam bab ini bahwa tabel nama terdiri dari 960 byte karena menggunakan 1 byte untuk merepresentasikan indeks dari masing-masing 960 ubin. Layar memiliki lebar 32 ubin dan tinggi 30 ubin, dengan setiap ubin mewakili area 8×8 piksel. Kita menggambar ubin-ubin ini dari kiri atas ke kanan bawah layar, satu baris demi satu baris, dari kiri ke kanan:

```
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)
```

Setiap `tile_address` dihitung dengan menambahkan nametable_address dasar ke offset tile saat ini. Karena setiap baris memiliki panjang 32 tile, kita mengalikan baris (y) dengan 32 dan menambahkan komponen x, yang dapat dianggap sebagai kolom. Untuk mendapatkan byte indeks sebenarnya (`nametable_entry`), kita membaca satu byte memori di `tile_address`. Nantinya dalam metode ini, kita akan menggunakan `nametable_entry` untuk mengambil

konten piksel tile dari tabel pola. Selanjutnya, kita mendapatkan entri tabel atribut yang sesuai dengan entri nametable saat ini:

```
        attrx = x // 4
        attry = y // 4
        attribute_address = attribute_table_address + attry * 8 + attrx
        attribute_entry = self.read_memory(attribute_address)
```

Karena setiap entri dalam tabel atribut 8×8 adalah untuk 16 ubin, kita membagi x dan y dengan 4 untuk mendapatkan entri atribut yang terhubung ke ubin yang dimaksud. Karena setiap attribute_entry adalah untuk empat area ubin 2×2 (sehingga total 16 ubin), kita perlu mempersempit pencarian ke area ubin tertentu:

```
        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")
```

`attribute_entry` berukuran 1 byte, dan setiap 2 bit dari byte tersebut sesuai dengan area ubin yang berbeda. Variabel `block` mewakili area ubin untuk ubin saat ini; nilainya bisa 0, 1, 2, atau 3. Tergantung pada nilai `block`, kita menggunakan operasi bitwise yang sesuai untuk mengambil dua bit spesifik untuk area ubin tersebut dari `attribute_entry` dan menyimpannya di `attribute_bits`.

Sekarang kita perlu mengambil piksel individual dari setiap ubin dari tabel pola:

```
   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
```

Ingat kembali dari “Tabel Pola dan Ubin” bahwa setiap tabel pola terdiri dari ubin 16-byte. Oleh karena itu, untuk menghitung alamat ubin, kita perlu mengalikan indeksnya

(nametable_entry) dengan 16 dan menambahkannya ke background_pattern_table_address. Selain itu, ubin tabel pola dibagi menjadi dua bidang bit, dengan setiap warna berukuran 2 bit dan masing-masing dari 2 bit tersebut berjarak 8 byte di bidang yang terpisah (lihat Gambar 6-2).

Strategi kita adalah membaca 2 byte, satu untuk bidang low_order dan satu untuk bidang high_order. Setiap byte menyimpan setengah dari entri piksel untuk satu baris ubin. Kita menggunakan fine_y untuk mewakili setiap baris ubin, kemudian fokus pada bit individual menggunakan fine_x, yang mewakili setiap kolom. Kombinasi bit dari setiap bidang dengan attribute_bits menghasilkan alamat di memori palet tempat warna piksel saat ini disimpan. Terakhir, kita menggambar setiap piksel dari ubin satu per satu di lokasi layar yang sesuai menggunakan warna yang telah ditentukan sebelumnya di NES_PALETTE:

```
        x_screen_loc = x * 8 + fine_x
        y_screen_loc = y * 8 + fine_y
        transparent = ((pixel & 3) == 0)
        # If the background is transparent, use the first color in the palette
        color = self.palette[0] if transparent else self.palette[pixel]
        self.display_buffer[x_screen_loc, y_screen_loc] = NES_PALETTE[color]
```

Mengatur piksel untuk layar berarti mengatur nilai dalam display_buffer, yang merupakan array NumPy karena itulah yang diterima Pygame.

Menggambar Sprite

Menggambar sprite memiliki beberapa kemiripan dengan menggambar ubin latar belakang. Namun, alih-alih membaca dari nametable, kita membaca dari OAM (self.spr). Setiap entri sprite dalam memori adalah 4 byte, yang mewakili posisi y sprite, indeks tabel pola, atribut, dan posisi x (lihat Tabel 6-5). Ada ruang hingga 64 entri sprite di OAM. Jika posisi y adalah 0xFF, maka entri tersebut tidak digunakan. Kita memulai draw_sprites() dengan memindahkan 4 byte sekaligus melalui OAM untuk menemukan semua entri yang valid:

```
  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]
```

Kita mengambil posisi y dan posisi x dari setiap sprite yang valid. Kita juga melihat bit 5 dari atributnya untuk melihat apakah itu sprite latar belakang. Sprite latar belakang hanya digambar jika latar belakangnya transparan. Perhatikan bahwa kita menelusuri memori sprite secara terbalik karena, seperti yang akan kita lihat di akhir bagian ini, sprite ke-nol memiliki arti khusus. Sama seperti kita menggambar ubin latar belakang satu piksel pada satu waktu, kita melakukan hal yang sama untuk sprite:

```
    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
```

Di sini, x dan y analog dengan `fine_x` dan `fine_y` dalam `draw_background()`. Kita berhati-hati agar tidak menggambar piksel yang berada di luar layar. Atribut lain yang dapat dimiliki sprite adalah kemampuan untuk dibalik secara vertikal (`flip_y`), yang ditentukan oleh bit ketujuh dalam byte atribut sprite:

```
          flip_y = bool((self.spr[i + 2] >> 7) & 1)
          sprite_line = y - y_position
          if flip_y:
            sprite_line = 7 - sprite_line
```

Jika sebuah sprite dibalik secara vertikal, kita membaca pikselnya dalam urutan vertikal terbalik. Kita menggunakan angka ajaib 7 karena setiap sprite berukuran 8×8 piksel. Membaca bit piksel sebenarnya dari tabel pola, berdasarkan `indeks` tabel pola, sangat mirip dengan pekerjaan yang dilakukan di `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
```

Saya menggunakan beberapa terminologi yang berbeda di sini (`bit0s_address` dan `bit1s_address` sebagai pengganti `low_order` dan `high_order`) karena penulis pikir penamaan yang berbeda mungkin akan lebih mudah dipahami oleh pembaca yang berbeda. Bit warna atribut adalah bit 0 dan 1 dalam byte atribut sprite, dan disimpan dalam `bit3and2` untuk warna akhir. Sprite juga dapat dibalik secara horizontal berdasarkan bit 6 dalam byte atribut:

```
    flip_x = bool((self.spr[i + 2] >> 6) & 1)
    x_loc = x - x_position # position within sprite
    if not flip_x:
      x_loc = 7 - x_loc
```

Kita menggabungkan dua bidang bit dan melewati proses menggambar piksel yang transparan:

```
    bit1and0 =(((bit1s >> x_loc) & 1) << 1) | (
                ((bit0s >> x_loc) & 1) << 0)
    if bit1and0 == 0: # transparent pixel... skip
          continue
```

PPU melacak apakah sprite ke-nol (entri pertama dalam OAM) bertabrakan dengan piksel latar belakang yang tidak transparan. Ini disebut benturan sprite-nol. Kami mengimplementasikan bentuk sederhana deteksi tabrakan ini di sini:

```
# This is not transparent. Is it a sprite-zero hit therefore?
# Check that left 8 pixel clipping is not off.
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
# Need to do this after sprite-zero checking so we still count background
# sprites for sprite-zero checks
if background_sprite and not background_transparent:
   continue # background sprite shouldn't draw over opaque pixels
```

Ketika terjadi benturan pada sprite nol, kita menandainya di register status. Dalam bagian kode ini, kita juga melewatkan penggambaran sprite latar belakang jika latar belakang tidak transparan. Ada flag yang dapat diatur di PPU untuk memotong 8 piksel kiri dari ubin atau sprite latar belakang. Flag tersebut juga diperiksa di bagian ini untuk memastikan jika flag tersebut aktif, tidak ada benturan sprite nol yang salah. Terakhir, kita mengambil warna piksel individual:

```
            color = bit3and2 | bit1and0
            color = self.read_memory(0x3F10 + color) # from palette
            self.display_buffer[x, y] = NES_PALETTE[color]
```

Untuk mengambil warna, kita menggabungkan bit3 dan 2 dengan bit1 dan 0 dan membaca dari lokasi yang sesuai di memori palet. Kemudian, kita menempatkan piksel ini di layar. Alih-alih membaca langsung dari palet, kita menggunakan `read_memory()` di sini karena kebutuhan untuk menggabungkan pencermian alamat.

Mengakses Register

PPU memiliki beberapa register yang dipetakan ke memori, dan ada beberapa teknis dan kekhasan saat membaca atau menulisnya. Kita akan mengatasi hal ini melalui metode `read_register()` dan `write_register()`. Dalam `read_register()`, pertama-tama kita menangani alamat 0x2002, yang digunakan untuk membaca register status:

```
def read_register(self, address: int) -> int:
   if address == 0x2002:
      self.addr_write_latch = False
      current = self.status
```

```
        self.status &= 0b01111111 # clear vblank on read to 0x2002
        return current
```

Saat register status dibaca, `self.addr_write_latch` diatur ke False, yang memodifikasi cara alamat ditulis ke 0x2006 (akan dibahas nanti). Selain itu, vblank dikosongkan saat pembacaan ke register status. Selanjutnya, `self.spr_address` saat ini di OAM dapat dibaca melalui 0x2004:

```
    elif address == 0x2004:
        return self.spr[self.spr_address]
```

Memori PPU di self.addr dapat dibaca dan ditulis melalui register 0x2007. Namun, pembacaannya dilakukan melalui buffer (`self.buffer2007`), dengan detail yang bervariasi tergantung pada alamat yang dibaca:

```
    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)
        # Every read to 0x2007 there is an increment
        self.addr += self.address_increment
        return value
    else:
        raise LookupError(f"Error: Unrecognized PPU read {address:X}")
```

Perhatikan bagaimana `self.address_increment` ditambahkan ke `self.addr` setelah setiap pembacaan. Ini memungkinkan pembacaan selanjutnya untuk secara otomatis mendapatkan entri berikutnya, baik 1 byte atau 32 byte lebih jauh.

Dalam `write_register()`, kita mengubah operasi PPU dengan menulis ke berbagai register yang dipetakan ke memori. Pertama, register 0x2000 dan 0x2001 disebut register kontrol. Register ini digunakan untuk mengubah berbagai nilai internal yang telah kita lihat digunakan di seluruh implementasi PPU lainnya:

```
  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)
```

Selanjutnya, kita akan menangani register 0x2003 hingga 0x2007:

```
    elif address == 0x2003:
        self.spr_address = value
    elif address == 0x2004:
        self.spr[self.spr_address] = value
        self.spr_address += 1
    elif address == 0x2005: # scroll
        pass
    elif address == 0x2006:
        # Based on https://wiki.nesdev.org/w/index.php/PPU_scrolling
        if not self.addr_write_latch: # first write
            self.addr = (self.addr & 0x00FF) | ((value & 0xFF) << 8)
        else: # second write
            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}")
```

Register 0x2003 mengatur `self.spr_address`. Register 0x2004 mengatur nilai di self.spr_address dan kemudian menambah `self.spr_address` sebesar 1. Register 0x2005 adalah register scroll; kami belum mengimplementasikannya di PPU sederhana kami, tetapi implementasi penuh akan membutuhkannya. Saat ini, emulator kami tidak akan berfungsi dengan game yang membutuhkan scrolling. Register 0x2006 digunakan untuk memodifikasi `self.addr`. Di sinilah `self.addr_write_latch` berperan: kita membutuhkan latch karena self.addr berukuran 16 bit (2 byte) tetapi hanya dapat ditulis ke 1 byte pada satu waktu. Terakhir, 0x2007 digunakan untuk menulis ke `self.addr`.

Mengakses Memori

Implementasi PPU kami pada dasarnya sudah selesai, tetapi kami membutuhkan metode pembantu untuk membaca dan menulis ke memori PPU. Metode `read_memory()` dan `write_memory()` ini cukup mirip dengan analognya di CPU:

```
    def read_memory(self, address: int) -> int:
        address = address % 0x4000 # mirror >0x4000
        if address < 0x2000: # pattern tables
            return self.rom.read_cartridge(address)
        elif address < 0x3F00: # nametables
            address = (address - 0x2000) % 0x1000 # 3000-3EFF is a mirror
            if self.rom.vertical_mirroring:
                address = address % 0x0800
            else: # horizontal mirroring
                if (address >= 0x400) and (address < 0xC00):
```

```
                address = address - 0x400
            elif address >= 0xC00:
                address = address - 0x800
        return self.nametables[address]
    elif address < 0x4000: # palette memory
        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 # mirror >0x4000
    if address < 0x2000: # pattern tables
        return self.rom.write_cartridge(address, value)
    elif address < 0x3F00: # nametables
        address = (address - 0x2000) % 0x1000 # 3000-3EFF is a mirror
        if self.rom.vertical_mirroring:
            address = address % 0x0800
        else: # horizontal mirroring
            if (address >= 0x400) and (address < 0xC00):
                address = address - 0x400
            elif address >= 0xC00:
                address = address - 0x800
        self.nametables[address] = value
    elif address < 0x4000: # palette memory
        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}")
```

Dalam metode ini, berbagai wilayah memori dipetakan ke area masing-masing—tabel pola (yang sebenarnya ada di kartrid), tabel nama, dan sejenisnya. Satu-satunya komplikasi adalah tabel nama dan memori palet dapat dicerminkan. Kita menangani ini menggunakan operator mod (%), seperti yang kita lakukan pada CPU.

6.8 PENGUJIAN EMULATOR GAME NES

Banyak ROM uji telah dibuat untuk orang-orang yang mengembangkan emulator NES. Beberapa di antaranya disertakan dalam repositori untuk buku ini. Ini dapat menguji CPU 6502 serta PPU. Terima kasih kepada Shay Green dan Kevin Horton atas pengembangan pengujian ini.

10 pengujian unit kami menjalankan ROM ini dan kemudian memeriksa bahwa nilai-nilai tertentu dalam memori NES virtual diatur dengan benar, seperti yang ditentukan oleh pembuat ROM uji. Seperti semua pengujian untuk buku ini, file untuk pengujian unit ini muncul di direktori pengujian di akar repositori kode sumber:

```
# 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):
        # Create machinery that we are testing
        rom = ROM(self.test_folder / "nestest" / "nestest.nes")
        ppu = PPU(rom)
        cpu = CPU(ppu, rom)
        # Set up tests
        cpu.PC = 0xC000 # special starting location for tests
        with open(self.test_folder / "nestest" / "nestest.log") as f:
            correct_lines = f.readlines()
        log_line = 1
        # Check every line of the log against our own produced logs
        while log_line < 5260: # go until first unofficial opcode test
            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):
    # Create machinery that we are testing
    rom = ROM(self.test_folder / "instr_test-v5" / "rom_singles" / "01-basics.nes")
    ppu = PPU(rom)
    cpu = CPU(ppu, rom)
    # Tests run as long as 0x6000 is 80, and then 0x6000 is result code; 0 means success
    rom.prg_ram[0] = 0x80
    while rom.prg_ram[0] == 0x80: # go until first unofficial opcode test cpu.step()
    self.assertEqual(0, rom.prg_ram[0], "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")]) # message ends with null terminator

def test_blargg_instr_test_v5_implied(self):
    # Create machinery that we are testing
    rom = ROM(self.test_folder / "instr_test-v5" / "rom_singles" / "02-implied.nes")
    ppu = PPU(rom)
```

```
        cpu = CPU(ppu, rom)
        # Tests run as long as 0x6000 is 80, and then 0x6000 is result code; 0 means success
        rom.prg_ram[0] = 0x80
        while rom.prg_ram[0] == 0x80: # go until first unofficial opcode test 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")]) # message ends with null terminator

    def test_blargg_instr_test_v5_branches(self):
        # Create machinery that we are testing
        rom = ROM(self.test_folder / "instr_test-v5" / "rom_singles" / "10-branches.nes")
        ppu = PPU(rom)
        cpu = CPU(ppu, rom)
        # Tests run as long as 0x6000 is 80, and then 0x6000 is result code; 0 means success
        rom.prg_ram[0] = 0x80
        while rom.prg_ram[0] == 0x80: #go until first unofficial opcode test  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")]) # message ends with null terminator

    def test_blargg_instr_test_v5_stack(self):
        # Create machinery that we are testing
        rom = ROM(self.test_folder / "instr_test-v5" / "rom_singles" / "11-stack.nes")
        ppu = PPU(rom)
        cpu = CPU(ppu, rom)
        # Tests run as long as 0x6000 is 80, and then 0x6000 is result code; 0 means success
        rom.prg_ram[0] = 0x80
        while rom.prg_ram[0] == 0x80: # go until first unofficial opcode test 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")]) # message ends with null terminator

    def test_blargg_instr_test_v5_jmp_jsr(self):
        # Create machinery that we are testing
        rom = ROM(self.test_folder / "instr_test-v5" / "rom_singles" / "12-jmp_jsr.nes")
        ppu = PPU(rom)
        cpu = CPU(ppu, rom)
        # Tests run as long as 0x6000 is 80, and then 0x6000 is result code; 0 means success
        rom.prg_ram[0] = 0x80
        while rom.prg_ram[0] == 0x80: # go until first unofficial opcode test 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")]) # message ends with null terminator

    def test_blargg_instr_test_v5_rts(self):
        # Create machinery that we are testing
        rom = ROM(self.test_folder / "instr_test-v5" / "rom_singles" / "13-rts.nes")
        ppu = PPU(rom)
```

```
    cpu = CPU(ppu, rom)
    # Tests run as long as 0x6000 is 80, and then 0x6000 is result code; 0 means success
    rom.prg_ram[0] = 0x80
    while rom.prg_ram[0] == 0x80: # go until first unofficial opcode test 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")]) # message ends with null terminator

  def test_blargg_instr_test_v5_rti(self):
    # Create machinery that we are testing
    rom = ROM(self.test_folder / "instr_test-v5" / "rom_singles" / "14-rti.nes")
    ppu = PPU(rom)
    cpu = CPU(ppu, rom)
    # Tests run as long as 0x6000 is 80, and then 0x6000 is result code; 0 means success
    rom.prg_ram[0] = 0x80
    while rom.prg_ram[0] == 0x80: # go until first unofficial opcode test 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")]) # message ends with null terminator

  def test_blargg_instr_test_v5_brk(self):
    # Create machinery that we are testing
    rom = ROM(self.test_folder / "instr_test-v5" / "rom_singles" / "15-brk.nes")
    ppu = PPU(rom)
    cpu = CPU(ppu, rom)
    # Tests run as long as 0x6000 is 80, and then 0x6000 is result code; 0 means success
    rom.prg_ram[0] = 0x80
    while rom.prg_ram[0] == 0x80: # go until first unofficial opcode test cpu.step()
    message = bytes(rom.prg_ram[4:]).decode("utf-8")
    print(message[0:message.index("\0")]) # message ends with null terminator
    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):
    # Create machinery that we are testing
    rom = ROM(self.test_folder / "instr_test-v5" / "rom_singles" / "16-special.nes")
    ppu = PPU(rom)
    cpu = CPU(ppu, rom)
    # Tests run as long as 0x6000 is 80, and then 0x6000 is result code; 0 means success
    rom.prg_ram[0] = 0x80
    while rom.prg_ram[0] == 0x80: # go until first unofficial opcode test cpu.step()
    message = bytes(rom.prg_ram[4:]).decode("utf-8")
    print(message[0:message.index("\0")]) # message ends with null terminator
    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()
```

Penting untuk memiliki pengujian otomatis seperti ini saat mengembangkan emulator. Bahkan ketika Anda berpikir CPU Anda sempurna, Anda mungkin melewatkan bug kecil yang mengacaukan seluruh program. Anda juga perlu tahu bahwa mengubah satu bagian emulator Anda tidak akan merusak bagian lain.

Memainkan Game

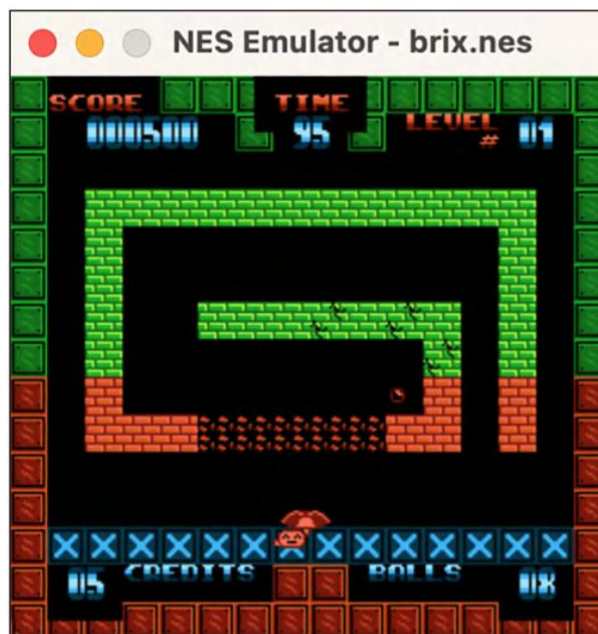
Pengujian unit adalah satu hal, tetapi pengujian sebenarnya dari emulator kita adalah apakah ia dapat memainkan perangkat lunak NES yang sebenarnya. Karena alasan hukum, kami tidak akan menguji perangkat lunak komersial apa pun di emulator NES kami. Karena kesederhanaannya, emulator kami tidak akan mampu memainkan sebagian besar pustaka NES.

Sebaliknya, repositori kode sumber buku ini mencakup beberapa game sumber terbuka atau domain publik yang mampu dijalankan oleh emulator kami. Ini adalah game sungguhan dalam arti bahwa game tersebut dapat dijalankan di konsol NES sungguhan.

Mari kita mulai dengan BrickBreaker, game mirip Breakout karya Aleff Correa yang penulis sebutkan sebelumnya. Dengan asumsi Anda telah menginstal Pygame dan NumPy, Anda dapat memainkan game ini hanya dengan menjalankan perintah ini dari direktori utama repositori:

```
% python3 -m NESEmulator NESEmulator/Games/brix.nes
```

Terlihat cukup bagus (lihat Gambar 6-8).



Gambar 6-8: BrickBreaker karya Aleff Correa

Selanjutnya, mari kita coba Chase karya Shiru:

```
% python3 -m NESEmulator NESEmulator/Games/Chase.nes
```

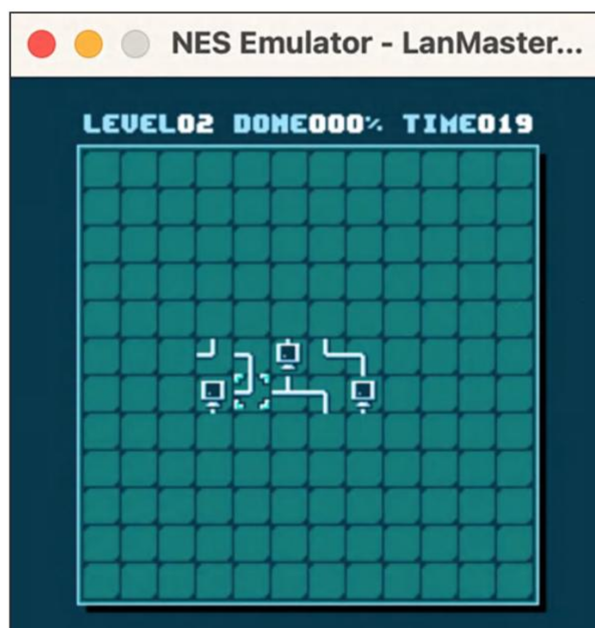
Gambar 6-9 menunjukkan permainan tersebut.

Gambar 6-9: Chase karya Shiru

Terakhir, mari kita coba Lan Master karya Shiru:

```
% python3 -m NESEmulator NESEmulator/Games/LanMaster.nes
```

Ini adalah permainan teka-teki, seperti yang ditunjukkan pada Gambar 6-10.



Gambar 6-10: Lan Master karya Shiru

Lan Master sangat lancar dimainkan di emulator kami, tetapi dua game lainnya tidak. Mengapa? Nah, Anda mungkin telah memperhatikan bahwa semuanya berjalan cukup lambat. Di MacBook Air M1 penulis yang menjalankan CPython 3.13, misalnya, game-game tersebut berjalan sekitar 14 FPS. Itu sekitar seperempat kecepatan NES asli. Emulator kami menjalankan game-game ini dengan benar, hanya saja sangat lambat.

Apa pelajarannya? Python, dan khususnya versi utama Python, CPython, lambat. Dalam beberapa tahun terakhir telah ada upaya untuk meningkatkan kinerja CPython, tetapi masih sangat lambat dibandingkan dengan sebagian besar implementasi bahasa pemrograman lainnya dan tidak dioptimalkan sejak awal untuk menulis program tingkat rendah seperti emulator. Untuk menulis program yang lebih berkinerja, Anda perlu melakukan beberapa langkah: Anda dapat menggunakan pustaka tertentu yang diimplementasikan dalam bahasa tingkat rendah, menggunakan Cython, menulis ekstensi C, atau menggunakan interpreter Python alternatif seperti PyPy.

Saya akan menyerahkan percepatan emulator menggunakan sesuatu seperti Cython sebagai latihan bagi pembaca. Penulis yakin bahwa dengan solusi yang tepat, Anda dapat membuat emulator NES ini mencapai 60 FPS seperti NES asli.

KODE BERTEMU KEHIDUPAN

Ketika saya memulai perjalanan pemrograman emulator saya dengan menulis VM CHIP-8 di Swift, impian saya yang sebenarnya adalah menulis emulator untuk konsol game pertama yang saya miliki saat kecil, NES. Dua tahun kemudian saya berhasil, menulis emulator NES dasar tanpa suara di C. Sementara VM CHIP-8 hanya membutuhkan waktu satu atau dua hari, emulator NES membutuhkan waktu sekitar 30 hari, tersebar selama setahun penelitian dan pemrograman yang dilakukan secara berkala. Saya menemukan penulisan bagian CPU cukup mudah, tetapi menulis renderer latar belakang yang presisi piksel jauh lebih menantang. Saya akhirnya memindahkan renderer latar belakang PPU dari emulator NES Michael Fogleman yang luar biasa dari Go ke C dan menggabungkannya dengan kode rendering sprite saya sendiri.

Jika emulator NES membutuhkan waktu sekitar 30 hari untuk ditulis dan VM CHIP-8 membutuhkan waktu sekitar dua hari, apakah proyek ini 15 kali lebih sulit? Saya rasa tidak. Tantangan saya dalam menulis PPU adalah mengingat semua detail teknis sekaligus. Saya terlalu fokus pada penulisan renderer yang sempurna piksel padahal seharusnya saya mulai dengan renderer per-frame seperti yang kita lakukan di bab ini. Pada saat itu juga tidak ada tutorial yang bagus, meskipun dokumentasi di NesDev sangat berharga. Saya pertama kali ingin menulis emulator NES saat remaja, jadi menyelesaikannya adalah mimpi yang menjadi kenyataan, meskipun cukup mendasar dan tanpa suara. (Saya

kemudian menambahkan suara, serta lebih banyak mapper daripada hanya NROM.)

Di bab ini, saya mencoba memberikan tutorial yang saya harapkan ada ketika saya mengembangkan emulator NES saya. Saya tahu rendering yang sempurna piksel akan terlalu rumit dan terlalu penuh dengan detail register internal yang esoteris untuk penulis emulator pemula, jadi saya kembali dan menulis ulang PPU emulator NES C saya sesederhana mungkin. Itulah renderer yang saya porting ke Python untuk bab ini.

Setelah menyelesaikan emulator NES, saya melanjutkan untuk menulis emulator untuk IBM PC asli. Itu adalah proyek yang jauh lebih rumit, sebagian besar karena Intel 8088/8086 jauh lebih rumit daripada MOS 6502. Dalam proyek itu, tidak menulis pengujian otomatis sejak awal menjadi bumerang bagi saya. Akhirnya saya berhasil membuatnya berfungsi pada tingkat dasar, tetapi seharusnya saya menulis pengujian jauh lebih awal. Semakin rumit mikroprosesor yang Anda emulasikan, semakin Anda membutuhkan pengujian otomatis sedini mungkin.

Aplikasi Dunia Nyata

Emulator mungkin paling sering digunakan untuk memainkan video game untuk sistem yang sudah tidak diproduksi lagi, tetapi emulator juga telah lama digunakan pada titik-titik penting dalam sejarah komputasi. Misalnya, ketika Bill Gates dan Paul Allen memulai Microsoft pada tahun 1975 dengan menulis interpreter BASIC untuk Altair 8800, seperti yang dibahas dalam Bab 2, mereka sebenarnya tidak memiliki Altair 8800 yang tersedia. Sebaliknya, mereka menulis emulator untuk mikroprosesor Intel 8080 Altair pada salah satu minikomputer di Harvard, tempat Gates kuliah.8

Apple telah melakukan transisi keluarga mikroprosesor yang digunakan dalam jajaran komputer Macintosh-nya sebanyak tiga kali: dari Motorola 68K ke Motorola/IBM PowerPC, dari PowerPC ke Intel x86, dan akhirnya dari Intel x86 ke Apple Silicon turunan ARM milik Apple sendiri. Apakah itu berarti Apple harus meminta pengembang untuk mengkompilasi ulang atau menulis ulang semua perangkat lunak mereka beberapa kali? Dalam jangka panjang, ya, tetapi dalam jangka pendek selama masa transisi, Apple menyediakan emulator. Mac PowerPC dapat menjalankan perangkat lunak Mac 68K, Mac Intel (pada awalnya) dapat menjalankan perangkat lunak Mac PowerPC, dan Mac Apple Silicon juga dapat menjalankan perangkat lunak Mac Intel. Apple adalah pengembang emulator yang luar biasa.

Faktanya, Mac PowerPC akan menjalankan perangkat lunak 68K lebih cepat daripada Mac 68K. Hal yang sama terkadang berlaku untuk Mac Apple Silicon yang menjalankan perangkat lunak Intel. Emulator juga penting untuk pelestarian perangkat lunak. Apa yang terjadi ketika sangat sulit untuk mendapatkan perangkat keras asli yang digunakan untuk

menulis perangkat lunak tersebut? Dalam kasus tersebut, emulator mungkin menjadi satu-satunya pilihan. Di sisi lain, emulator terkadang digunakan dalam fase desain platform komputasi baru. Sebelum platform tersebut ada, para perancang dapat menggunakan emulator untuk mensimulasikan seperti apa platform tersebut dan membantu mengembangkan fitur-fiturnya dalam lingkungan yang realistis.

Terakhir, penulisan emulator sangat edukatif. Ini adalah salah satu cara terbaik untuk mengajarkan cara kerja komputer pada tingkat rendah, seperti yang saya harap telah Anda temukan dalam bab ini.

LATIHAN SOAL

1. Cobalah untuk meningkatkan performa emulator kita hingga setara dengan NES asli—dengan kata lain, 60 FPS. Anda mungkin perlu menggunakan sesuatu seperti Cython, atau Python yang dikombinasikan dengan bahasa tingkat rendah seperti C atau Rust melalui ekstensi. Hampir tidak mungkin untuk menjalankan Python murni pada 60 FPS pada CPython 3.13 dan mikroprosesor era 2025.
2. Tambahkan dukungan untuk scrolling ke emulator kita menggunakan dokumentasi di https://nesdev.org.
3. Implementasikan mapper lain. Saat ini emulator kita hanya mengimplementasikan NROM, mapper paling dasar. Dua mapper populer lainnya adalah MMC1 dan UxROM.
4. Dan sekarang untuk tantangan terbesar dari semuanya: cobalah menulis APU untuk emulator kita sehingga Anda dapat memainkan game dengan suara.

BAB 7
KLASIFIKASI DENGAN K-NEAREST TETANGGA

Bab ini akan memperkenalkan k-nearest neighbors (KNN), sebuah algoritma pembelajaran mesin yang sangat sederhana namun sangat efektif untuk beberapa aplikasi. Pertama kali dikembangkan pada tahun 1950-an dan 1960-an,[1] KNN dapat digunakan untuk klasifikasi (menentukan kategori sesuatu) dan regresi (memprediksi nilai). Bagi pembaca yang merasa kesulitan dengan kompleksitas pembelajaran mesin, KNN menyediakan titik masuk yang mudah diakses namun tetap relevan dengan dunia nyata. Dalam bab ini, kita akan menggunakan KNN untuk menyelesaikan dua masalah klasifikasi dengan tingkat akurasi yang tinggi: membedakan berbagai jenis ikan dan mengenali angka tulisan tangan. Kemudian, di Bab 8, kita akan memperluas KNN ke beberapa masalah regresi terkait.

7.1 KEBANGKITAN PEMBELAJARAN MESIN

Sejak tahun 1950-an, penelitian kecerdasan buatan (AI) tradisional terutama berkaitan dengan pemanfaatan algoritma untuk memodelkan kecerdasan manusia. Namun, mulai tahun 1990-an, dan terutama sejak munculnya jaringan saraf yang didukung oleh komputasi GPU pada tahun 2000-an, sebagian besar fokus penelitian dan produk AI telah beralih ke subbidang pembelajaran mesin, yang menggunakan kumpulan data besar untuk melatih model yang dapat membuat keputusan tanpa perlu merujuk pada metode pemecahan masalah manusia. Untuk memahami pergeseran ini, bayangkan perbedaan antara program catur yang mengevaluasi posisi berdasarkan heuristik yang dipahami dengan baik dari grandmaster dan program catur yang mengevaluasi posisi berdasarkan bobot yang disetel secara otomatis yang diasimilasi dari analisis statistik jutaan permainan. Memang, pembelajaran mesin sangat bergantung pada statistik.

Semua aplikasi pembelajaran mesin yang menarik yang kita kenal, termasuk LLM, pengenalan gambar, dan asisten digital, dibangun menggunakan jaringan saraf multilayer canggih yang dilatih pada GPU atau prosesor saraf khusus. Ini dikenal sebagai pembelajaran mendalam. Untuk memprogram kerangka kerja pembelajaran mendalam dari awal, Anda memerlukan pemahaman yang signifikan tentang kalkulus dan statistik. Bahkan jika Anda menghindari sebagian dari itu dengan menggunakan pustaka, Anda biasanya masih membutuhkan kumpulan data yang besar, yang seringkali sulit diperoleh, dan perangkat keras yang canggih.

Karena hambatan-hambatan ini, para programmer yang tertarik untuk memulai dengan pembelajaran mesin terkadang merasa terintimidasi. Mereka takut matematika akan terlalu sulit atau mereka akan kekurangan sumber daya yang dibutuhkan untuk mengembangkan aplikasi yang mereka minati. Terlebih lagi, beberapa programmer yang sedang belajar suka membangun proyek mereka dari awal; mereka tidak ingin hanya menginstal pip untuk mendapatkan solusi yang tidak memberi mereka pemahaman tentang bagaimana proses yang mendasarinya sebenarnya bekerja.

Namun, seperti yang akan Anda lihat dalam bab ini, pembelajaran mesin dapat memiliki titik awal yang sangat mudah didekati jika Anda tidak langsung terjun ke dalam pembelajaran mendalam. Anda dapat memprogram algoritma KNN dari awal dan menggunakannya untuk memecahkan masalah nyata, dan Anda tidak memerlukan latar belakang matematika di luar tingkat sekolah menengah untuk memahami apa yang Anda lakukan. Satu-satunya konsep statistik yang dibutuhkan untuk mengimplementasikan KNN adalah gagasan tentang rata-rata (mean), dan satu-satunya rumus lain yang Anda perlukan adalah teorema Pythagoras untuk menemukan jarak Euclidean antara dua titik. Itu tidak terlalu sulit, bukan?

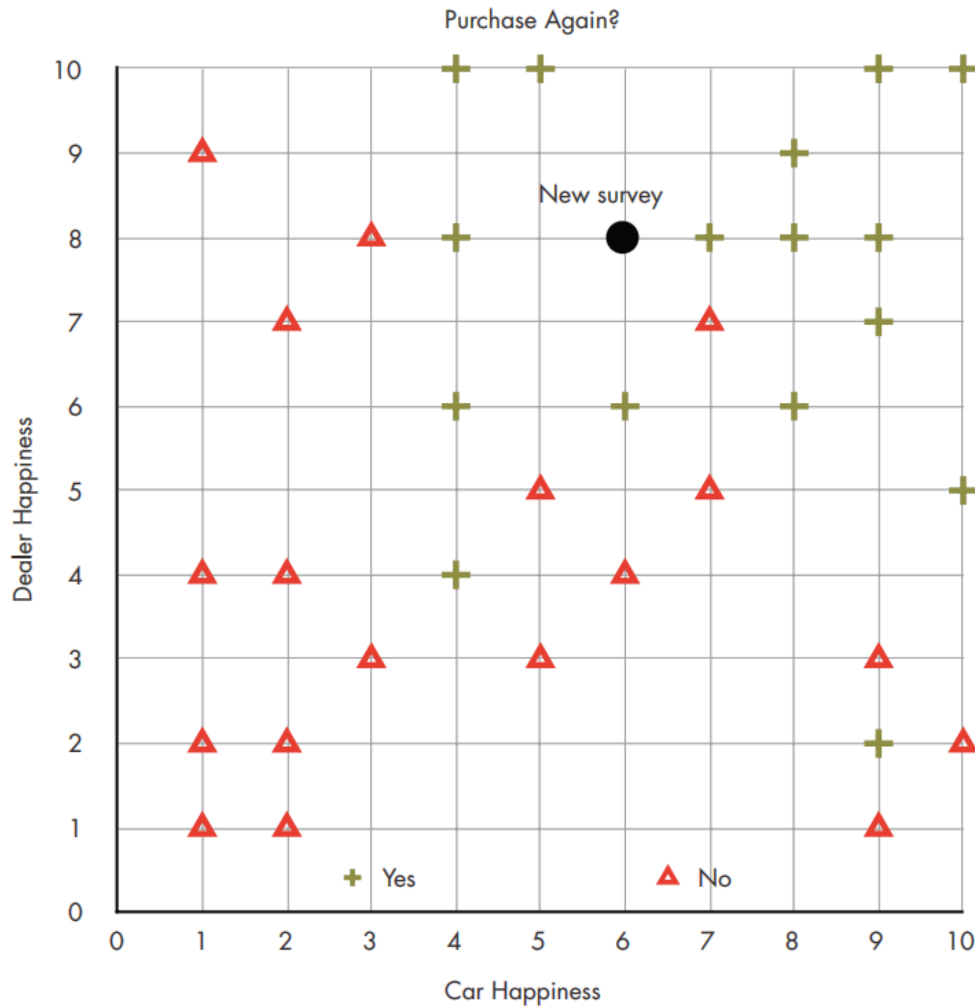
7.2 CARA KERJA KNN

Algoritma KNN membuat asumsi sederhana: tetangga dari suatu titik data kemungkinan besar adalah titik data lain yang memiliki kesamaan paling banyak dengannya. Misalnya, jika penulis mencoba mencari tahu penyakit yang diderita pasien, mungkin pasien lain dengan gejala dan tanda vital yang sama adalah petunjuk terbaik. Kebetulan, inilah mengapa Anda mungkin tidak menginginkan dokter yang baru lulus dari sekolah kedokteran. Dokter yang lebih berpengalaman dapat menggunakan heuristik yang jelas, yaitu "Saya pernah melihat pasien lain seperti ini sebelumnya," untuk menawarkan panduan awal yang lebih baik.

Cara lain untuk menjelaskannya adalah bahwa titik data yang paling dekat dengan nilai yang tidak diketahui adalah yang paling mungkin memberi tahu kita apa nilai yang tidak diketahui tersebut. Mungkin Anda adalah dealer mobil, dan Anda ingin tahu apakah Anda harus menghabiskan lebih banyak uang pemasaran untuk menarik pelanggan tetap potensial dengan mengirimkan surat promosi lebih lanjut. Pelanggan telah mengisi survei kepuasan pelanggan yang menilai berbagai aspek bisnis Anda. Anda memiliki banyak data dari pelanggan sebelumnya yang mengisi survei yang sama, dan Anda tahu apakah mereka akhirnya membeli mobil lain dari Anda atau tidak.

Anda dapat membandingkan peringkat survei pelanggan potensial yang berulang ini dengan peringkat pelanggan sebelumnya untuk menemukan pelanggan yang paling mirip dengannya dalam hal watak. Jika pelanggan sebelumnya dengan peringkat serupa akhirnya membeli mobil lain, Anda tahu bahwa mungkin layak untuk mengeluarkan uang untuk mengirimkan lebih banyak materi pemasaran kepadanya. Pada dasarnya itulah analisis yang dilakukan KNN: ia membiarkan data sebelumnya "memilih" nilai yang mungkin terkait dengan beberapa data baru.

Mari kita lihat contoh terakhir secara visual sepanjang dua dimensi. Gambar 7-1 adalah kumpulan data fiktif dari peringkat responden pada survei dealer untuk "Kebahagiaan Mobil" dan "Kebahagiaan Dealer". Tanda silang adalah responden survei yang membeli mobil lain dari dealer, segitiga adalah responden survei yang tidak, dan titik bulat adalah responden survei baru.



Gambar 7-1: Data responden survei dealer mobil

Bagaimana kita mengklasifikasikan responden survei baru kita? Apakah dia kemungkinan akan membeli mobil lain atau tidak? Menggunakan KNN, kita perlu terlebih dahulu memilih nilai untuk k, meta-heuristik. Ini hanyalah jumlah tetangga yang akan kita lihat. Jika kita menetapkan k menjadi 1, maka kita hanya melihat titik data yang paling dekat dengan nilai yang tidak diketahui. Pada Gambar 7-1, dengan responden survei kita di (6, 8), titik data terdekat berikutnya untuk dibandingkan adalah di (7, 8), dan itu adalah seseorang yang membeli mobil lain. Dengan k ditetapkan menjadi 1, oleh karena itu kita akan menyimpulkan bahwa layak untuk mengeluarkan uang untuk mengirimkan materi pemasaran tambahan kepada responden baru kita.

Tetapi bagaimana jika k ditetapkan ke angka yang lebih besar dari 1? Solusi umum dalam algoritma KNN adalah "memilih". Misalnya, jika k adalah 3 dalam skenario kita, Anda dapat melihat bahwa tiga titik data terdekat adalah dua yang membeli mobil baru dan satu yang tidak. Karena mayoritas membeli mobil lain, kita masih dapat menyimpulkan bahwa pengeluaran untuk pemasaran kepada responden survei baru tersebut tetap layak. Jika k ditetapkan menjadi 2, kita akan mengalami masalah karena satu responden memutuskan

untuk membeli mobil lain dan satu lagi tidak. Maka, kita harus memiliki semacam kriteria pemecah kebuntuan.

Dan itu saja. Itulah keseluruhan algoritma KNN untuk klasifikasi. Kita melihat k titik data yang dekat dengan titik data yang dimaksud, dan kita membiarkan mereka memilih apa yang seharusnya menjadi data yang tidak diketahui. Dengan demikian, kita dapat meringkas algoritma KNN untuk klasifikasi sebagai berikut:

1. Pilih k, jumlah tetangga yang akan dibandingkan dengan titik data yang belum diklasifikasi.
2. Temukan k tetangga terdekat dengan titik data tersebut.
3. Berikan suara pada klasifikasi titik data berdasarkan kelas dari k tetangga terdekat.

Algoritma sederhana ini membutuhkan tiga klarifikasi: Apa artinya menjadi tetangga "terdekat"? Bagaimana pemenang pemungutan suara ditentukan? Dan berapa nilai k yang tepat? Semua pertanyaan ini mungkin memiliki jawaban yang berbeda untuk berbagai aplikasi KNN.

“Terdekat” paling umum ditentukan menggunakan jarak Euclidean.

Dengan kata lain, jika kita menggambar garis lurus pada Gambar 7-1 antara titik yang belum diklasifikasikan (titik bulat) dan semua titik data lainnya pada grafik, garis terpendek akan menentukan tetangganya. Namun, untuk beberapa aplikasi yang tidak memiliki titik data numerik, sesuatu seperti jarak Hamming (menghitung perbedaan) mungkin berlaku. Fungsi jarak selain jarak Euclidean berada di luar cakupan bab ini.

Pemungutan suara biasanya berarti menentukan kelas mana yang dimiliki oleh mayoritas tetangga terdekat. Anda memerlukan semacam kriteria pemecah seri, meskipun Anda tetap menggunakan angka ganjil untuk k. Ini karena banyak aplikasi memiliki lebih dari dua kelas. Misalnya, bagaimana jika Anda memiliki tiga kelas dan k ditetapkan menjadi 5? Anda mungkin berakhir dengan dua tetangga terdekat yang termasuk dalam kelas A, dua yang termasuk dalam kelas B, dan satu yang termasuk dalam kelas C. Haruskah titik data yang dimaksud diklasifikasikan sebagai A atau B?

Menentukan nilai k yang tepat sebenarnya cukup mudah. Kecuali kita memiliki pengetahuan domain khusus yang memberi tahu kita sebaliknya, kita harus menggunakan nilai yang telah ditemukan paling akurat dalam pengujian. Kita dapat menguji KNN dengan beberapa nilai k yang berbeda dengan dataset dan kumpulan titik uji tertentu dan melihat nilai mana yang paling bermanfaat.

Itulah dasar-dasar klasifikasi dengan KNN. Jelas, ada banyak pilihan dan peningkatan yang dapat dilakukan di luar garis besar sederhana ini, tetapi Anda sudah cukup tahu untuk mengimplementasikan KNN!

7.3 MENGIMPLEMENTASIKAN KLASIFIKASI DENGAN KNN

Seperti algoritma itu sendiri, kode kita untuk mengimplementasikan KNN akan cukup sederhana. Tetapi sebelum kita dapat sampai ke algoritma, kita membutuhkan tipe generik

untuk merepresentasikan titik data. Kita akan membuat kelas DataPoint yang merupakan protokol, artinya tidak akan ada instance dari tipe ini sendiri, tetapi hanya dari subkelasnya. Ini adalah templat abstrak yang menguraikan fungsionalitas yang harus dimiliki oleh tipe titik data yang lebih konkret:

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: ...
```

Protokol kami menetapkan bahwa sebuah titik data harus memiliki atribut jenis (atau kelas dalam istilah klasifikasi), metode `from_string_data()` untuk mengkonversi satu baris dari file CSV (nilai yang dipisahkan koma) ke instance kelas, dan `distance()` untuk menemukan jarak antara dua titik data dengan jenis yang sama. Kami akan membuat subkelas DataPoint untuk dua set data konkret yang kami gunakan dalam bab ini. Implementasi KNN utama kami adalah melalui kelas yang, tidak mengherankan, disebut 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)

# Read a CSV file and return a list of data points
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))
```

Sintaks petunjuk tipe kelas `KNN[DP: DataPoint]:` menyatakan bahwa tipe generik, DP, dikaitkan dengan KNN dan bahwa DP harus merupakan subkelas dari DataPoint. Kita akan memuat semua dataset kita dari file CSV. Metode `_read_csv()` kelas KNN kita menggunakan modul csv bawaan Python untuk memuat file-file ini. Setiap baris dalam file CSV digunakan

untuk menginisialisasi salah satu subkelas DataPoint kita melalui metode kelas `from_string_data()`. Kita akan kembali ke detail spesifik file CSV sebentar lagi ketika kita melihat dataset pertama kita.

Sekarang kita telah memuat dataset di kelas `KNN`, kita siap untuk mengimplementasikan algoritma `KNN` yang sebenarnya. Dari mana kita mulai? Diberikan sebuah titik yang ingin kita klasifikasikan, hal pertama yang kita butuhkan adalah mengidentifikasi k-tetangga terdekatnya. Semua titik data kita memiliki metode `distance()` bawaan, jadi kita cukup menghitung jarak dari titik data yang belum diklasifikasikan ke setiap titik data dalam dataset dan menemukan k titik terdekat:

```
def nearest(self, k: int, data_point: DP) -> list[DP]:
  return sorted(self.data_points, key=data_point.distance)[:k]
```

Ya, ini hanya satu baris kode. Kita hanya menggunakan hasil metode distance() untuk mengurutkan semua titik data, dan kemudian kita menyimpan k nilai terendah (titik-titik dengan jarak terkecil dari `data_point`). Inti dari algoritma KNN memang sesederhana itu.

Apakah ini cara paling efisien untuk menemukan tetangga terdekat? Tidak. Pengurutan adalah operasi $O(n \log n)$, di mana n adalah jumlah titik data dalam dataset. Jika dataset sangat besar, ini akan menjadi hambatan yang signifikan. Kita dapat sedikit meningkatkan ini dengan menulis kode untuk mengevaluasi jarak semua titik data secara manual menggunakan struktur data tambahan untuk hanya menyimpan k nilai terkecil. Untuk meningkatkan kinerja lebih jauh, kita mungkin memerlukan struktur data yang lebih canggih daripada daftar yang tidak diurutkan untuk menyimpan dataset. Singkatnya, ada pertukaran di sini antara kinerja algoritma dan kompleksitas struktur data.

Alternatif lain adalah tidak benar-benar mencari tetangga di seluruh dataset setiap kali. Ada berbagai pendekatan untuk membatasi pencarian, baik melalui pra-komputasi subset titik data yang lebih terbatas yang mewakili keseluruhan atau pengambilan sampel dataset selama pencarian melalui apa yang disebut pencarian perkiraan. Meskipun demikian, teknik pengurutan sederhana kami cukup cepat untuk aplikasi kami dan sesuai dengan judul bagian buku ini, "Pembelajaran Mesin Super Sederhana."

Selanjutnya, kita perlu melakukan "pemungutan suara." Itu melibatkan penghitungan berapa banyak dari setiap kelas (atau jenis) yang ada di tetangga terdekat dan mengembalikan kelas yang paling banyak:

```
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]
```

Pertama, kita mencari `neighbors`. Kemudian, kita menggunakan tipe koleksi `Counter` bawaan Python untuk menemukan jenis yang paling umum di antara tetangga. Panggilan `most_common(1)` mengembalikan item tunggal yang paling umum di `Counter`, dan `[0][0]` mengatakan ambil item pertama itu dari koleksi dan ambil label jenisnya. Counter secara

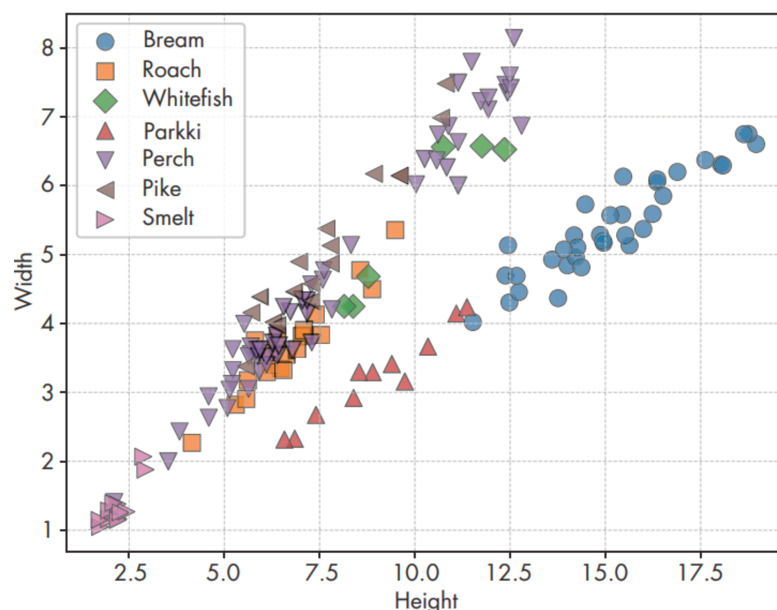
internal akan terstruktur sebagai pasangan kunci-nilai yang terlihat seperti `[("amphibian", 3), ("reptile", 4), ("mammal", 1)]`. Dalam contoh itu, baris tersebut akan menemukan pasangan kunci-nilai dengan nilai tertinggi, `("reptile", 4)`, dan hanya mengembalikan kuncinya, `"reptile"`. Kita tidak menangani pemecah seri secara sistematis di sini—kita hanya menyerahkannya pada keinginan `Counter`. Sekali lagi, ingat judul bagiannya.

Itu saja. Berkat beberapa rutin pustaka standar Python yang bagus, pekerjaan algoritmik sebenarnya di balik KNN secara efektif hanya tiga baris kode antara `nearest()` dan `classify()`. Sudah kubilang itu akan "sangat sederhana." Dengan algoritma yang sudah ada, mari kita terapkan KNN pada dua masalah klasifikasi.

Klasifikasi Ikan

Misalkan Anda bekerja sebagai programmer untuk sebuah perusahaan yang membuat alat pencari ikan untuk para pemancing. Alat ini terdiri dari kamera di ujung tiang yang bergerak di bawah perahu. Alat ini memiliki pengenalan gambar bawaan, sehingga ketika ikan melewati kamera bawah air, kamera dapat secara otomatis mengambil gambar ikan tersebut dan mengenali persegi panjang di dalam foto yang berisi ikan. Kamera juga dapat memperkirakan dimensi ikan dalam foto tersebut. Tugas Anda adalah menulis lapisan perangkat lunak di atas sistem pengenalan gambar ini yang memberi tahu pemancing jenis ikan apa itu. Lagipula, tidak semua ikan legal untuk ditangkap.

Untungnya, kita memiliki dataset di domain publik yang akan membantu kita dalam tugas klasifikasi ikan ini. Data ini awalnya berasal dari makalah Finlandia tahun 1917 karya Pekka Brofeldt yang dalam bahasa Inggris berjudul “Contribution to the Knowledge of Fish Stocks in Dangerous Lakes.” Data ini berisi dimensi dan berat 159 ikan dari sebuah danau, yang diklasifikasikan berdasarkan spesies (sehingga program kami mungkin hanya berfungsi di danau tersebut). Gambar 7-2 menunjukkan ikan dalam dataset kami hanya dalam dua dimensi, tinggi dan lebar.



Gambar 7-2: Ikan dikategorikan berdasarkan spesies dan diplot sepanjang dimensi tinggi dan lebar

Seperti yang mungkin Anda duga, ikan dari spesies yang serupa cenderung berdekatan jika dilihat dari segi tinggi dan lebar. Itu merupakan indikator yang baik bahwa KNN mungkin merupakan algoritma yang berguna untuk masalah ini. Mari kita lihat seperti apa data mentahnya. Berikut adalah contoh beberapa baris pertama dari 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
```

Baris pertama adalah judul yang menjelaskan setiap kolom. Tiga dimensi panjang, yang mewakili jarak dari hidung setiap ikan ke berbagai bagian tubuh yang berbeda, dalam sentimeter, dan berat dalam gram. Ada beberapa ambiguitas dalam sumber tentang satuan tinggi dan lebar. Namun, selama satuannya konsisten antar sampel, data tersebut tetap berguna, meskipun kita tidak tahu apakah itu sentimeter, semacam persentase, atau lainnya.

Kelas Ikan

Untuk mengimplementasikan pengklasifikasi ikan kita, kita perlu membangun subkelas DataPoint untuk merepresentasikan ikan:

```
# 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
```

Secara teknis, kita tidak perlu secara eksplisit menjadikan Fish sebagai subkelas dari DataPoint untuk kesesuaian protokol dalam petunjuk tipe Python. Dengan memenuhi semua persyaratan protokol DataPoint, Fish dapat menggantikan DataPoint tanpa perlu membuat subkelasnya berkat konsep subtipe implisit. Namun demikian, kita mendeklarasikan Fish sebagai subkelas dari DataPoint secara eksplisit karena memberikan kejelasan kepada pembaca kode kita dan membantu dalam pemeriksaan tipe.

Setiap baris CSV disediakan dari kelas KNN sebagai daftar string untuk kelas Fish untuk dikonversi menjadi instance Fish. Metode `from_string_data()` melakukan konversi ini. Metode `distance()` menghitung jarak Euclidean antara instance saat ini dan Fish lainnya. Kita menemukan perbedaan antara setiap dimensi dalam dataset, mengkuadratkan perbedaan tersebut, dan menjumlahkannya. Kemudian, kita mengembalikan akar kuadrat (** 0,5) dari jumlah tersebut. Perhatikan bahwa kita tidak mempertimbangkan atribut berat dari dataset sebagai bagian dari perbandingan. Hal ini karena kita akan menggunakan dimensi ikan untuk memprediksi beratnya di bab berikutnya, sehingga berat ikan yang dimaksud tidak diketahui.

Uji Unit

Kita memiliki uji unit untuk memastikan kita mendapatkan hasil yang diharapkan dari detektor ikan kita. Kita mulai dengan uji yang memeriksa apakah ikan yang paling dekat dengan ikan sampel adalah ikan yang diharapkan:

```
# 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)
```

Selanjutnya, kita coba mengklasifikasikan sampel ikan:

```
  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--
```

Untuk menjalankan pengujian unit ini, kami memiliki kode menjalankan pengujian standar kami:

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

Perhatikan bahwa kami telah melewati sekitar 40 baris dalam file `test_knn.py` yang berisi pengujian lebih lanjut yang akan muncul nanti di bab ini dan bab berikutnya.

7.4 KLASIFIKASI ANGKA TULISAN TANGAN

Pengenalan karakter optik (OCR) berkaitan dengan penggunaan komputer untuk mengenali karakter dalam gambar teks yang diketik atau ditulis tangan. Misalnya, kantor pos memiliki mesin sortir yang menggunakan OCR untuk secara otomatis membaca alamat yang ditulis pada amplop. Berbagai macam teknik telah berhasil diterapkan untuk melakukan OCR, termasuk KNN. Di bagian ini, kita akan menggunakan KNN untuk mengembangkan pengenal angka tulisan tangan yang akan mencapai akurasi 98 persen pada sampel dalam set pengujian yang signifikan.

Dataset yang akan kita gunakan dikembangkan oleh Cenk Kaynak dan Ethem Alpaydin pada tahun 1998 di Universitas Bogazici di Istanbul, Turki, dan kemudian diserahkan ke UC Irvine Machine Learning Repository di bawah lisensi Creative Commons Attribution 4.0 International. Dataset ini terdiri dari 5.620 bitmap angka tulisan tangan (0–9) yang dibuat oleh 43 orang berbeda. Gambar angka tersebut diperkecil ukurannya menjadi 8×8 piksel, dengan setiap piksel direpresentasikan dalam file CSV sebagai bilangan bulat antara 0 dan 16 yang menunjukkan tingkat skala abu-abunya. Setiap baris dalam CSV terdiri dari 64 bilangan bulat yang mewakili 64 piksel dalam gambar angka tulisan tangan, ditambah bilangan bulat ke-65 yang mewakili angka (0–9) mana yang harus diklasifikasikan dalam gambar tersebut. Gambar 7-3 menunjukkan contoh dari angka-angka tersebut.

Gambar 7-3: Beberapa gambar angka tulisan tangan 8×8 dari dataset OCR

Angka-angka tersebut diperbesar sedikit secara artifisial dalam gambar, tetapi Anda dapat melihat bahwa pengecilan skala menjadi 8×8 mengakibatkan hilangnya beberapa detail. Tingkat detail yang lebih rendah tersebut, dan karenanya dimensi yang lebih rendah dalam

dataset, membuat program kami lebih cepat dieksekusi. Membandingkan 64 piksel antar gambar jelas jauh lebih cepat daripada membandingkan 1.024 piksel antar gambar (awalnya berukuran 32×32 sebelum diperkecil skalanya).

Kelas Angka

Untuk merepresentasikan setiap angka, kita mendefinisikan subkelas lain dari 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))
```

Kita menyimpan data piksel sebagai array NumPy. Ini nyaman karena di bab berikutnya, kita akan menggunakan Pygame untuk bekerja dengan coretan angka tulisan tangan kita sendiri, dan Pygame dapat berinteraksi langsung dengan array NumPy. Karena metode distance() menghitung jarak antar array NumPy, kita menggunakan fungsi bawaan NumPy untuk mengimplementasikan bentuk jarak Euclidean.

Uji Unit

Dataset dibagi menjadi 3.823 gambar angka dalam set pelatihan dan 1.797 gambar angka dalam set pengujian. Kita akan menggunakan set pelatihan sebagai dataset yang digunakan implementasi KNN kita untuk membuat prediksi, dan kita akan menguji berapa banyak angka dalam set pengujian yang dapat diidentifikasi dengan benar terhadapnya. Mari kita definisikan kasus uji lain di `test_knn.py` untuk ini, setelah kasus uji Fish tetapi sebelum baris `if name == " main ":`

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)
```

Tes ini memuat dataset pelatihan (*digits.csv*) ke dalam sebuah instance dari kelas KNN. Kemudian, ia membuka set data uji (*digits_test.csv*) dan mengubah data CSV menjadi daftar titik data, *test_data_points*. Selanjutnya, ia mencoba mengklasifikasikan setiap titik data satu per satu dan mencatat berapa banyak klasifikasi yang berhasil. Terakhir, ia melaporkan persentase tersebut dan gagal jika akurasinya di bawah 97 persen.

Mari kita jalankan semua tes untuk melihat hasilnya. Dengan tes ikan dan OCR yang digabungkan, ini akan memakan waktu cukup lama. Mengklasifikasikan 1.797 gambar digit tersebut membutuhkan waktu sekitar 11 detik di laptop saya:

```
% python3 -m tests.test_knn
Correct Classifications: 1761 out of 1797 or 97.9966611018364%
....
----------------------------------------------------------------------
Ran 4 tests in 10.826s
OK
```

Berdasarkan dokumentasi yang disertakan dengan dataset OCR (lihat bagian bawah *KNN/datasets/digits/readme.txt*), kita tahu bahwa para penulis telah menguji sendiri akurasi dataset tersebut menggunakan KNN dengan berbagai nilai k yang berbeda. Mereka menemukan akurasi tertinggi, 98 persen, dengan k diatur ke 1. Output pengujian pengklasifikasi kita sesuai dengan itu. Itu berarti berhasil!

KODE BERTEMU KEHIDUPAN

Beberapa tahun yang lalu, saya mengajar mata kuliah pengantar tentang kecerdasan buatan kepada sekelompok mahasiswa tingkat akhir. Saya membagi mata kuliah tersebut menjadi dua bagian. Bagian pertama mencakup apa yang kami sebut di awal bab ini sebagai "AI tradisional," termasuk algoritma seperti A* dan MiniMax (keduanya dapat Anda temukan di buku saya sebelumnya, Classic Computer Science Problems in Python) dan konsep seperti sistem pakar. Bagian kedua dikhususkan untuk pembelajaran mesin. Saya menggunakan KNN sebagai contoh pertama algoritma pembelajaran mesin karena kesederhanaannya yang ekstrem. Ini berfungsi sebagai transisi yang bagus ke dunia pembelajaran mesin, itulah sebabnya saya percaya bahwa hal yang sama dapat dilakukan untuk para pembaca buku ini. Sejak saat itu, departemen saya telah menggunakan KNN sebagai topik presentasi demonstrasi pengajaran untuk kandidat yang datang ke kampus untuk wawancara menjadi dosen ilmu komputer baru.

Mereka diberi tahu topik presentasi setidaknya seminggu sebelum mereka datang ke kampus dan memiliki kesempatan untuk mempersiapkan diri. KNN sangat cocok sebagai topik untuk tujuan ini karena, meskipun ada banyak kemungkinan perluasan dan peningkatan pada algoritma inti, algoritma inti itu sendiri seharusnya tidak membutuhkan waktu lama untuk dijelaskan, dan bahkan audiens yang terdiri dari dosen atau mahasiswa tahun pertama yang belum familiar pun seharusnya dapat memahaminya. Ini adalah tolok ukur yang bagus untuk mengetahui apakah seseorang siap menjadi pengajar yang baik.

Sebagai catatan tambahan, Anda akan terkejut betapa banyaknya pemegang gelar PhD dengan latar belakang pembelajaran mesin yang tidak mampu memberikan kuliah pengantar yang baik tentang topik seperti KNN. Perlu diingat bahwa gelar PhD adalah gelar penelitian, bukan gelar pengajaran. Inilah mengapa, ketika Anda memberi nasihat kepada anak Anda tentang ke mana harus kuliah, Anda harus mempertimbangkan perguruan tinggi yang berfokus pada pengajaran. Di universitas riset besar, mahasiswa mungkin diajar oleh dosen riset yang tidak peduli dengan pengajaran, dosen kontrak yang menganggap pengajaran sebagai pekerjaan paruh waktu, atau dalam kasus terburuk, mahasiswa pascasarjana yang sangat tidak berpengalaman.

Memiliki fakultas yang penuh dengan pemegang gelar PhD tidak berarti banyak bagi pengalaman mahasiswa sarjana dalam kursus pengantar ketika para dosen tersebut lebih peduli pada hibah penelitian daripada pengajaran. Sebaliknya, di perguruan tinggi yang berfokus pada pengajaran, Anda memiliki seluruh fakultas penuh waktu (sebagian besar memang bergelar PhD) yang dipekerjakan karena mereka sepenuhnya berdedikasi pada seni pengajaran dan seringkali benar-benar menyukai berada di kelas pengantar. Yang hilang adalah koneksi ke penelitian mutakhir, tetapi bagi seorang mahasiswa S1, koneksi itu

umumnya bukanlah hal yang akan memberikan dampak terbesar pada karier mereka. Namun, anggaplah apa yang saya katakan ini sebagai masukan saja, karena saya telah bekerja di sebuah perguruan tinggi pendidikan selama sembilan tahun terakhir.

Aplikasi Dunia Nyata

KNN telah banyak digunakan di dunia nyata untuk berbagai hal, mulai dari pengenalan karakter optik hingga sistem rekomendasi dan dari klasifikasi teks hingga pemodelan keuangan. Kesederhanaan dan penerapannya yang luas menjadikannya diajarkan secara universal dalam pembelajaran mesin.

Namun, ketika menggunakan KNN dalam praktik, beberapa masalah yang telah disinggung dalam bab ini harus diatasi. Yang pertama adalah menemukan nilai k yang tepat. Ini biasanya dilakukan melalui validasi silang menggunakan dataset uji. Berapa nilai k yang paling sesuai dengan data uji? Menggunakan nilai yang terlalu kecil dapat menyebabkan overfitting, di mana model terlalu dekat dengan satu dataset tertentu. Sementara itu, nilai yang terlalu besar dapat menyebabkan underfitting, di mana model terlalu jauh dari panduan data uji.

Tantangan selanjutnya adalah implikasi kinerja algoritma dasar dengan dataset besar berdimensi tinggi. Seperti yang disebutkan sebelumnya dalam bab ini, dua cara untuk mengatasi masalah ini adalah dengan merancang struktur data yang lebih baik untuk menyimpan dataset atau menggunakan pencarian perkiraan. Salah satu struktur data paling populer untuk mempercepat pencarian tetangga terdekat adalah pohon $k-d$. Namun, ini adalah struktur data yang cukup kompleks dan hanya sepadan dengan kerumitannya jika kinerja sangat penting.

Memilih fungsi jarak yang tepat juga sangat penting. Jarak Euclidean berfungsi untuk banyak aplikasi, tetapi jarak Hamming sesuai untuk dimensi boolean, dan fungsi jarak lainnya telah dipelajari dengan baik dalam literatur penelitian. Fungsi jarak yang tepat bersifat spesifik aplikasi; tidak ada solusi yang cocok untuk semua. Seringkali, Anda juga harus menormalisasi data untuk menghilangkan kemungkinan perbedaan satuan atau besaran yang memengaruhi hasil. Kami tidak menormalisasi data dalam contoh ikan di bab ini, dan data dalam contoh OCR semuanya dalam satuan dan skala yang sama sehingga tidak memerlukan normalisasi.

Meskipun kita melihat bahwa mengimplementasikan KNN dari awal hampir sepele, banyak pustaka pembelajaran mesin Python populer memiliki fungsi KNN bawaan yang sangat dioptimalkan. Misalnya, implementasi scikit-learn banyak digunakan.

LATIHAN SOAL

1. Temukan dataset lain yang Anda minati yang dapat diklasifikasikan secara akurat oleh implementasi KNN kami.
2. Cobalah untuk mempercepat pengujian unit dengan meningkatkan kinerja metode `distance()` kelas Digit sambil mempertahankan akurasi 98 persen dari pengujian tersebut. Anda bebas untuk beralih dari penggunaan array NumPy jika Anda mau. Anda

bahkan dapat beralih dari penggunaan jarak Euclidean murni. Mungkin Anda bahkan tidak perlu membandingkan setiap piksel?

3. Implementasikan ulang pengklasifikasi kami menggunakan pustaka scikit-learn. Bandingkan kinerja pengklasifikasi kami dengan pengklasifikasi KNN yang terintegrasi dalam scikit-learn.

BAB 8
REGRESI DENGAN KNN

Dalam bab ini, kita akan memperluas implementasi KNN kita untuk melakukan regresi. Untuk tujuan kita, regresi berarti memprediksi nilai numerik. Dengan beberapa tambahan kecil pada kode kita dari Bab 7, kita dapat menggunakan kelas KNN yang sama untuk tidak hanya mengklasifikasikan tetapi juga membuat prediksi tentang nilai atribut numerik apa pun dalam dataset kita.

Kita akan menerapkan regresi pada dua contoh KNN dari bab sebelumnya. Pertama, kita akan meninjau kembali dataset ikan dan menggunakan regresi untuk memprediksi berat ikan berdasarkan dimensinya. Kemudian, kita akan menulis program yang memungkinkan pengguna untuk menggambar sebagian dari sebuah angka dan kemudian memprediksi seperti apa sisa gambar tersebut.

Tidak seperti bab-bab lain dalam buku ini, bab ini tidak berdiri sendiri. Bab ini dibangun berdasarkan bab sebelumnya. Pastikan Anda telah menyelesaikan Bab 7 sebelum mempelajari bab ini.

8.1 CARA KERJA REGRESI KNN

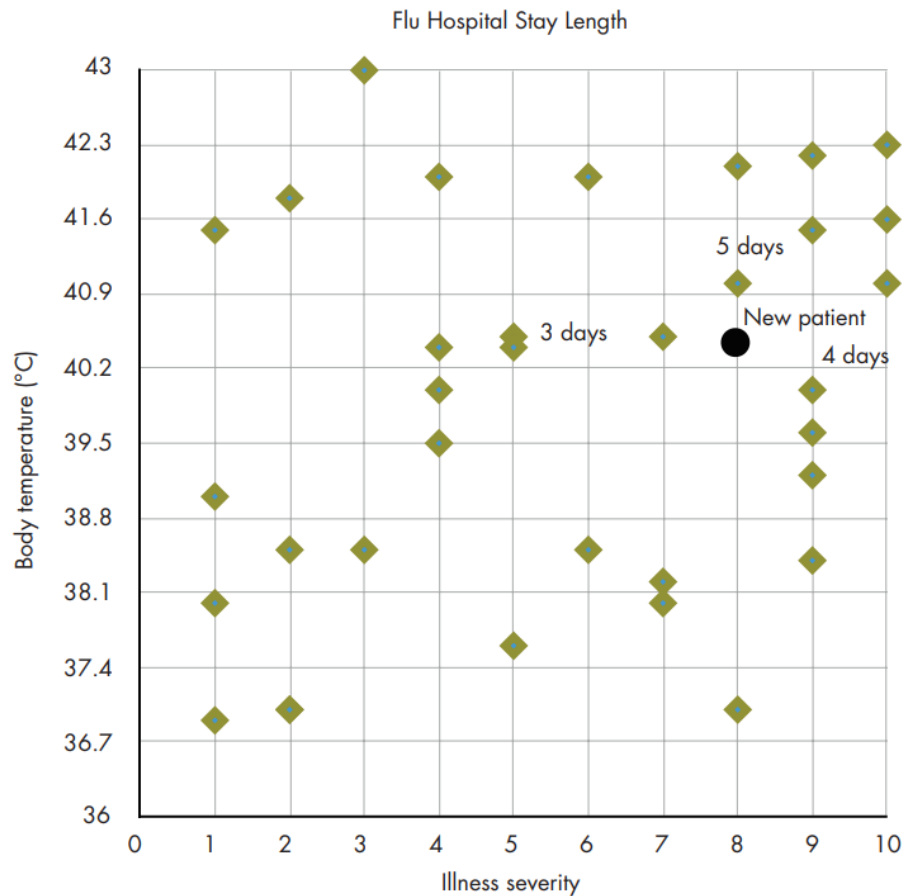
Dalam klasifikasi KNN, kita mencoba memprediksi kelas atau kategori tempat suatu titik data berada, memilih kelas yang sesuai dari serangkaian opsi yang terbatas. Dalam regresi KNN, alih-alih memprediksi kelas, kita mencoba memprediksi nilai atribut. Nilai atribut ini biasanya berupa angka, artinya berpotensi ada rentang nilai tak terbatas yang dapat ditetapkan. Tentu saja, masuk akal jika nilai atribut tersebut hilang jika itu adalah sesuatu yang ingin kita prediksi.

Sebagai contoh, katakanlah kita adalah rumah sakit yang menugaskan pasien ke kamar. Kita mungkin ingin mengetahui berapa hari pasien kemungkinan perlu tinggal. Kita dapat melihat data masa lalu dari pasien dengan diagnosis, gejala, dan tanda vital yang serupa untuk membuat prediksi. Mari kita lihat contoh ini secara visual dengan scatterplot sepanjang dua dimensi (Gambar 8-1).

Misalkan berlian pada Gambar 8-1 mewakili pasien masa lalu yang dirawat di rumah sakit karena flu. Penyakit mereka dinilai oleh dokter pada skala keparahan dari 1 hingga 10, dan suhu tubuh mereka dicatat pada saat masuk. Kita juga memiliki data tentang berapa lama mereka akhirnya tinggal di rumah sakit.

Titik bundar mewakili pasien yang baru saja dirawat. Kita memiliki peringkat keparahan dan suhu tubuh mereka, dan kita ingin memprediksi berapa lama mereka akan tinggal di rumah sakit sehingga kita dapat menempatkan mereka di ruangan yang sesuai. Jika kita menggunakan KNN dengan jarak Euclidean dan menetapkan k menjadi 3, maka kita akan melihat tiga pasien dalam data masa lalu yang paling dekat pada scatterplot dengan titik pasien baru. Gambar tersebut mencatat masa inap mereka masing-masing tiga, empat, dan lima hari. Memperkirakan lama rawat inap pasien baru menggunakan metode ini dapat sesederhana

merata-ratakan tiga tetangga terdekat. Rata-rata tersebut (baik mean maupun median) adalah empat, jadi kita dapat memprediksi bahwa pasien baru akan dirawat di rumah sakit selama empat hari.



Gambar 8-1: Lama Rawat Inap di Rumah Sakit untuk Flu Berdasarkan Suhu Tubuh dan Tingkat Keparahan Penyakit

Secara lebih luas, berikut adalah langkah-langkah untuk melakukan regresi dengan KNN:

1. Pilih k, jumlah tetangga yang akan dibandingkan dengan titik data dengan atribut yang hilang.
2. Temukan k tetangga terdekat dengan titik data tersebut.
3. Rata-ratakan nilai atribut yang sesuai di antara k tetangga terdekat untuk memprediksi nilai atribut yang hilang tersebut untuk titik data yang dimaksud.

Seperti yang Anda lihat, menggunakan KNN untuk regresi sangat mirip dengan menggunakan KNN untuk klasifikasi. Perbedaannya hanya terletak pada langkah terakhir. Kita memiliki beberapa pertanyaan dan jawaban yang sama tentang algoritma ini seperti yang kita miliki di bab sebelumnya: Berapa nilai k yang tepat? Bagaimana cara kita menghitung jarak? Lihat Bab 7 untuk pembahasan pertanyaan-pertanyaan tersebut.

Kita juga memiliki pertanyaan baru: Apa artinya mengambil rata-rata? Biasanya ini adalah mean atau median. Seperti halnya pertanyaan tentang fungsi jarak yang tepat, cara

terbaik untuk mengambil rata-rata dapat bersifat spesifik aplikasi. Umumnya diperlukan beberapa pengetahuan domain untuk membuat penentuan terbaik.

8.2 MENERAPKAN REGRESI DENGAN KNN

Untuk melakukan regresi, kita hanya perlu menambahkan dua metode ke kelas KNN kita dari Bab 7. Satu memprediksi atribut numerik skalar, dan satu memprediksi atribut yang berupa array angka. Kita akan menggunakan yang terakhir untuk contoh tulisan tangan sehingga kita dapat memprediksi piksel. Berikut adalah pembaruannya:

```
# KNN/knn.py
# Predict a numeric property of a data point based on the k-nearest neighbors.
# Find the average of that property from the neighbors and return it.
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))
# Predict a NumPy array property of a data point based on the k-nearest neighbors.
# Find the average of that property from the neighbors and return it.
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))
```

Seperti fungsi `classify()` dari bab sebelumnya, metode ini dimulai dengan menemukan k tetangga terdekat. Kemudian, mereka menghitung dan mengembalikan rata-rata beberapa properti di antara tetangga tersebut. Kita memanfaatkan sifat dinamis Python di sini dengan memungkinkan pemanggil untuk menentukan properti sebagai string dan kemudian menggunakan `getattr()` untuk mengambil properti (atau atribut) yang ditentukan berdasarkan nama.

Satu-satunya perbedaan nyata di sini antara `predict()` dan `predict_array()` adalah bahwa yang terakhir menggunakan fungsi `sum()` NumPy alih-alih fungsi `sum()` bawaan Python. Array NumPy sebenarnya juga akan bekerja dengan fungsi `sum()` bawaan karena mereka mengimplementasikan operator plus, tetapi versi NumPy sedikit lebih cepat. Pada dasarnya hanya membutuhkan dua baris kode lagi (kedua fungsi hampir sama) dan kita sudah dapat membuat prediksi.

Memprediksi Berat Ikan

Mari kita tambahkan uji unit ke FishTestCase untuk memastikan metode `predict()` baru kita berfungsi. Kami ingin menjawab pertanyaan, “Jika kita mengetahui dimensi seekor ikan, dapatkah kita membuat perkiraan yang tepat tentang berapa beratnya?” Jawabannya, tentu saja, adalah ya:

tests/test_knn.py

```
def test_predict(self):
   k: int = 5
   fish_knn = KNN(Fish, self.data_file)
```

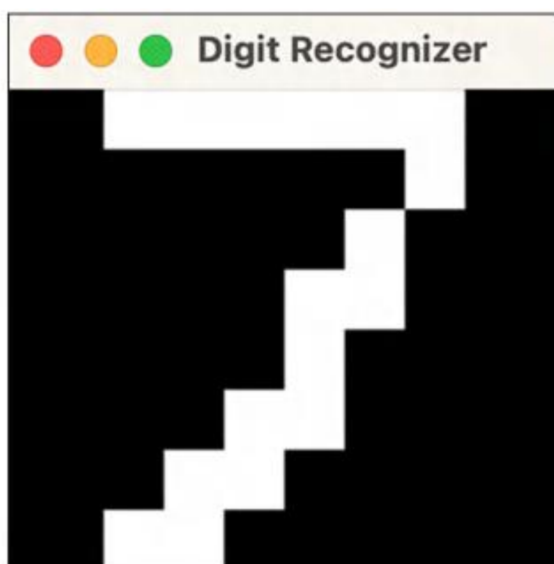
```
    test_fish: Fish = Fish("", 0.0, 20.0, 23.5, 24.0, 10.0, 4.0)
    predict_fish: float = fish_knn.predict(k, test_fish, "weight")
    self.assertEqual(predict_fish, 165.0)
```

Dalam metode ini, kita membuat `test_fish` tanpa berat yang ditentukan (0,0), dan kita membandingkannya dengan lima ikan terdekat berdasarkan dimensi anatominya—`length1`, `length2`, `length3`, `width`, dan `height`. (Ingat dari bab sebelumnya bahwa kita tidak membandingkan ikan berdasarkan berat dalam metode `distance()`.) Kemudian kita memprediksi beratnya dengan merata-ratakan berat kelima tetangga terdekat ini. Jalankan uji unit lagi, dan Anda akan menemukan bahwa berat ikan diprediksi dengan benar. Untuk menguji apakah metode prediksi ini akurat secara lebih umum, kita dapat mencoba menjalankannya di seluruh dataset ikan. Karena dataset mencakup berat yang diketahui dari setiap sampel, kita dapat mengukur seberapa akurat hasil KNN kita dibandingkan dengan berat ikan yang sebenarnya.

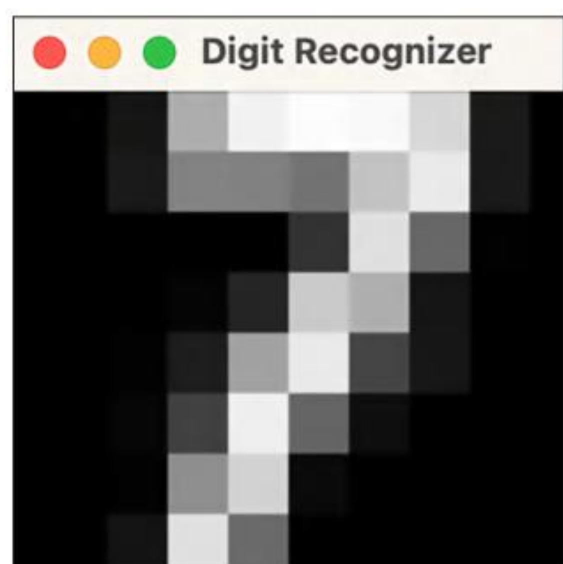
Memprediksi Sisa Angka Tulisan Tangan

Pada bab sebelumnya, kita telah mengklasifikasikan dengan benar 98 persen dari kumpulan gambar angka tulisan tangan terhadap kumpulan pelatihan dari jenis gambar piksel 8×8 yang sama menggunakan KNN. Dalam contoh KNN terakhir ini, kita akan mengklasifikasikan angka 8x8 yang digambar pengguna dan bahkan memprediksi seperti apa piksel gambar lainnya. Alih-alih mengimplementasikannya sebagai serangkaian uji unit lainnya, kita akan membuat program interaktif yang menyenangkan menggunakan Pygame yang memungkinkan pengguna untuk menggambar di jendela yang berisi grid 8x8.

Berikut pratinjau dari apa yang sedang kita bangun. Gambar 8-2 menunjukkan jendela gambar dengan angka 7 bergelombang yang penulis coba gambar. Penulis menekan tombol C dan program tersebut mengklasifikasikannya dengan benar sebagai angka 7 (yang ditampilkan di terminal, tidak ditampilkan dalam gambar).



Gambar 8-2: Menggambar angka 7 dalam program pengenal angka



Gambar 8-3: Piksel angka yang diprediksi berdasarkan piksel tetangga terdekat

Gambar 8-3 menunjukkan jendela setelah menekan tombol P, yang memicu prediksi seperti apa tampilan piksel angka lainnya berdasarkan rata-rata piksel dari sembilan tetangga terdekat.

Dalam program sederhana kita, kita hanya dapat menggambar dengan warna putih. Prediksi menghasilkan angka 7 yang lebih baik karena dapat menggunakan lebih banyak tingkat keabu-abuan. Kita mulai program kita dengan beberapa impor dan konstanta:

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 # pixel to digit scale factor
D_TO_P = 255 / 16 # digit to pixel scale factor
K = 9
WHITE = (255, 255, 255)
```

Konstanta `PIXEL_WIDTH` dan `PIXEL_HEIGHT` adalah ukuran satu gambar.

Konstanta `P_TO_D` mengkonversi antara 255 gradasi abu-abu dalam representasi piksel yang akan kita gunakan dan 16 gradasi abu-abu dalam dataset sumber; `D_TO_P` adalah kebalikannya. Kita menetapkan K, jumlah tetangga yang akan dipertimbangkan KNN, menjadi 9, dan `WHITE` adalah konstanta untuk warna putih dalam format piksel RGB.

Ini adalah program singkat. Kita hanya memiliki satu fungsi `run()` pusat yang menghabiskan waktu yang sama untuk menangani antarmuka pengguna seperti halnya menjalankan KNN.

Berikut adalah awal dari fungsi tersebut:

```
def run():
    # Create a 2D array of pixels to represent the digit
    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)
    # Start up Pygame, create the window
    pygame.init()
    screen = pygame.display.set_mode(size=(PIXEL_WIDTH, PIXEL_HEIGHT),
                                     flags=pygame.SCALED | pygame.RESIZABLE)
    pygame.display.set_caption("Digit Recognizer")
```

Pada beberapa baris pertama ini, kita membuat array piksel, memuat dataset, dan menginisialisasi Pygame. Jendela utama diinisialisasi sebagai jendela berukuran 8x8 piksel yang "diregangkan". Anda dapat mengubah ukuran jendela dan jendela tersebut akan mempertahankan dimensi 8x8-nya. Ini diatur melalui flag `set_mode()` `(flags=pygame.SCALED | pygame.RESIZABLE)`.

Selanjutnya, kita perlu mengatur loop utama:

```
while True:
    pygame.surfarray.blit_array(screen, digit_pixels)
    pygame.display.flip()
```

Karena ini adalah program GUI yang menggunakan Pygame, kita secara efektif memiliki sebuah event loop. Program ini mendengarkan suatu tindakan dari pengguna dan kemudian merespons. Tindakan ini bisa berupa event keyboard atau mouse. Untuk menjaga agar layar tetap sinkron, kita terus-menerus menampilkan `digit_pixels` ke layar di awal loop. Kemudian, kita menangani event keyboard:

```
for event in pygame.event.get():
    if event.type == pygame.KEYDOWN:
        key_name = pygame.key.name(event.key)
        if key_name == "c": # classify the digit
            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}")
```

Kita mulai dengan tombol C yang digunakan untuk klasifikasi. Ini mirip dengan klasifikasi yang kita lakukan di bab sebelumnya. Hasilnya dicetak ke konsol. Satu-satunya bagian yang rumit adalah transformasi piksel yang mewakili gambar ke bentuk yang dapat digunakan oleh pengklasifikasi kita. Pada intinya, kita beralih dari format piksel 255 gray dan beberapa dimensi ke array datar 16 gray. Penanganan keyboard berlanjut:

```
        elif key_name == "e": # erase the digit
            digit_pixels.fill(0)
        elif key_name == "p": # predict what the digit should look like
            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)
```

Tombol E hanya menghapus array piksel. Tombol P adalah bagian prediksi. Pertama, kita kembali mengkonversi array piksel ke bentuk yang dapat digunakan oleh kelas KNN, seperti sebelumnya. Selanjutnya, kita menggunakan metode `predict_array()` untuk mendapatkan

piksel yang diprediksi, yang merupakan rata-rata piksel dari sembilan entri terdekat dalam set pelatihan kita. Kemudian kita mengkonversi hasil tersebut kembali ke bentuk yang dapat ditampilkan di Pygame. Ini tidak hanya melibatkan pembentukan ulang ke array dua dimensi tetapi juga mengubah ke format RGB, di mana tingkat keabu-abuan yang sama diulang di setiap tiga saluran warna. Rantai `reshape()` dan `transpose()` berjalan dari satu ke dua dimensi, dan panggilan `stack()` membuat dimensi ketiga yang semuanya bernilai sama—misalnya, tingkat keabu-abuan 128 menjadi (128, 128, 128) untuk RGB.

Sisa kode tersebut menggambar piksel putih di mana pun pengguna mengklik, keluar saat pengguna menutup jendela, dan memanggil fungsi `run()` saat `main.py` dieksekusi:

```
        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()
```

Hanya dibutuhkan sekitar 50 baris kode untuk membuat pengenal angka tulisan tangan berbasis GUI menggunakan kelas KNN yang sudah ada. Python sangat ringkas! Cobalah: memang tidak sempurna, tetapi dapat mengenali sebagian besar tulisan tangan penulis dengan benar.

KODE BERTEMU KEHIDUPAN

Pada tahun 2016, saya mengerjakan proyek pendidikan sederhana bernama SwiftSimpleNeuralNetwork sebagai persiapan untuk bab tentang membangun jaringan saraf dari awal di Swift untuk buku kedua saya, Classic Computer Science Problems in Swift. Saya mengimplementasikan pengenalan angka tulisan tangan menggunakan kerangka kerja tersebut, dan hasilnya sangat lambat.

Sejujurnya, tidak ada optimasi sama sekali, dan aplikasi tersebut sepenuhnya single threaded dan terikat CPU. Hebatnya, implementasi algoritma KNN yang juga tidak dioptimalkan, tetapi jauh lebih sederhana, dalam bab ini mengunggulinya baik dalam kecepatan maupun akurasi.

Anekdot ini menunjukkan dua pelajaran penting: algoritma yang lebih kompleks tidak selalu merupakan algoritma yang lebih baik untuk aplikasi tertentu, dan penting untuk melakukan riset agar mendapat informasi yang baik tentang algoritma apa yang digunakan untuk

aplikasi apa. Itulah mengapa saya menyertakan bagian "Aplikasi Dunia Nyata" di akhir setiap bab dalam buku ini.

Mengetahui pilihan algoritma Anda penting di banyak bidang. Oleh karena itu, salah satu manfaat membaca buku survei seperti ini adalah buku ini memperkenalkan Anda pada algoritma dan teknik baru yang mungkin belum Anda ketahui sebelumnya. Dengan begitu, ketika Anda menemukan masalah yang dapat diterapkan pada algoritma dan teknik tersebut, Anda akan siap.

Aplikasi di Dunia Nyata

KNN dapat menjadi alat yang hebat untuk memulai ketika mencoba melakukan regresi, karena sangat mudah digunakan. Hanya sedikit penyetelan yang diperlukan, tidak seperti pada jaringan saraf, dan karena pada dasarnya tidak ada pelatihan yang terlibat, aplikasi berbasis KNN sangat mudah untuk dijalankan.

Para peneliti benar-benar telah menggunakan KNN untuk memprediksi lama rawat inap di rumah sakit, seperti yang dijelaskan di awal bab ini. Pei, Lin, dan Chen menemukan bahwa KNN hampir sama akuratnya dengan teknik yang lebih canggih seperti regresi logistik atau random forest untuk memprediksi lama rawat inap pasien COVID-19. Saya dapat menemukan beberapa studi lain di domain yang sama yang menggunakan KNN. Regresi KNN juga telah digunakan untuk aplikasi dalam penambangan teks, pertanian, dan pasar keuangan. Hal ini masuk akal secara intuitif—peristiwa atau titik data dari masa lalu yang paling mirip dengan apa yang sedang terjadi saat ini kemungkinan besar akan paling membantu dalam membuat prediksi.

Karena masalah kinerja, KNN tidak bekerja dengan baik ketika data bising atau dataset terlalu besar dalam hal jumlah titik dan dimensi. Namun, untuk sebagian besar aplikasi, ini adalah titik awal yang wajar untuk dipertimbangkan.

LATIHAN SOAL

1. Buktikan (atau sanggah) bahwa KNN efektif untuk prediksi berat ikan dengan menjalankannya pada seluruh dataset ikan dan membandingkan hasil KNN dengan berat yang diketahui dari dataset. Rata-rata, seberapa akurat prediksi KNN?
2. Ubah program pengenal digit kita untuk menggunakan grid yang lebih besar, misalnya 64×64 alih-alih 8×8. Ini akan memungkinkan pengguna untuk menggambar digit yang lebih halus. Anda perlu menemukan cara untuk menurunkan skala gambar 64×64 secara akurat menjadi 8×8 untuk menggunakannya dengan dataset pelatihan.
3. Gunakan implementasi regresi KNN kita atau dari pustaka seperti scikit-learn untuk mencoba membuat prediksi menggunakan dataset yang Anda minati.

BAB 9
OPERASI BITWISE

Manipulasi bit tingkat rendah sangat penting untuk setengah dari proyek dalam buku ini. Jika Anda tidak memiliki latar belakang dalam operasi bitwise, lampiran ini memberikan gambaran umum, termasuk apa yang dilakukan oleh operasi bitwise yang paling penting, bagaimana menggunakannya di Python, dan beberapa contoh penggunaannya.

9.1 TINJAUAN BINER

Saya berasumsi bahwa sebagian besar pembaca, sebagai programmer tingkat menengah atau mahir, sudah familiar dengan biner. Jika Anda demikian dan hanya ingin penyegaran singkat tentang operasi bitwise, Anda dapat melewati bagian ini. Namun, jika Anda tidak familiar dengan biner, bagian ini akan membantu Anda memulai, meskipun tidak komprehensif.

Semua informasi dalam komputer disimpan sebagai 1 dan 0. Ini nyaman karena jenis perangkat keras yang digunakan untuk membangun komputer dapat secara fisik merepresentasikan 1 dan 0 dengan cukup mudah. Misalnya, jika ada listrik (atau "sinyal"), kita dapat mengatakan bahwa itu mewakili angka 1, sedangkan tidak adanya sinyal listrik mewakili angka 0. Biner juga bermanifestasi secara fisik dalam teknologi CD dan DVD yang sekarang sudah usang.

Pembacanya memiliki laser yang bergerak di atas permukaan cakram. Ketika laser tidak memantul kembali karena cakram memiliki lubang mikroskopis, itu mewakili angka 0. Jika laser memantul kembali karena tidak ada lubang, itu mewakili angka 1. Satu contoh fisik terakhir adalah kode QR. Kehadiran titik hitam dapat berupa angka 1, dan ketidakhadirannya dapat berupa angka 0. Ada banyak manifestasi fisik biner yang mudah digunakan.

Bagaimana semua angka 1 dan 0 tersebut diubah menjadi informasi? Urutan angka 1 dan 0 mewakili angka dalam biner. Dan setelah kita memiliki angka, kita dapat mewakili jenis informasi lainnya. Angka tertentu dapat mewakili huruf tertentu dalam dokumen elektronik. Atau dapat mewakili warna tertentu. Angka lain dapat mewakili di mana di layar untuk menempatkan warna tersebut. Tak lama kemudian kita akan memiliki piksel.

Namun, bagaimana urutan angka 1 dan 0 merepresentasikan angka di luar 1 atau 0? Di situlah sistem bilangan biner, yang juga disebut basis 2, berperan.

Angka-angka umum yang digunakan sehari-hari berada dalam basis 10, yang dikenal sebagai desimal. Itu berarti setiap digit dalam angka tersebut dapat memiliki 10 nilai berbeda $(0-9)$, dan setiap digit itu sendiri merepresentasikan pangkat 10. Misalnya, angka 427 sebenarnya adalah $(4 \times 10^2) + (2 \times 10^1) + (7 \times 10^0)$. Demikian pula, setiap digit dalam bilangan biner dapat memiliki dua nilai berbeda (0 atau 1), dan setiap digit itu sendiri mewakili pangkat 2. Bilangan 427 adalah 110101011 dalam biner, yang merupakan $(1 \times 28) + (1 \times 27) + (0 \times 26) + (1 \times 25) + (0 \times 24) + (1 \times 23) + (0 \times 22) + (1 \times 21) + (1 \times 20)$. Angka 1 adalah pangkat 2 yang "aktif" dan angka 0 adalah pangkat 2 yang "nonaktif".

Untuk menguji pemahaman Anda, cobalah mengkonversi beberapa angka dari desimal ke biner dan sebaliknya. Berapa 73 dalam biner? Berapa 11000 dalam desimal? Cobalah beberapa angka lagi yang Anda pilih. Anda dapat memeriksa pekerjaan Anda dengan Python, di mana bilangan biner direpresentasikan sebagai literal dengan menggunakan awalan 0b, seperti `0b11` untuk bilangan desimal 3:

```
>>> value = 0b11
>>> value
3
```

Sementara itu, fungsi `bin()` Python menerima bilangan bulat dan mengembalikan string yang diformat sebagai nilai biner yang setara:

```
>>> bin(3)
'0b11'
```

Setiap angka biner 1 atau 0 yang tersimpan dikenal dalam komputasi sebagai bit. Dan satu byte standar terdiri dari 8 bit. Semua komputer modern menggunakan byte 8-bit sebagai unit penyimpanan standar mereka. Nilai maksimum yang dapat ditampung oleh satu byte (ketika merepresentasikan bilangan bulat tak bertanda) adalah 255 karena delapan angka 1, atau 11111111 dalam biner, adalah 255 dalam desimal. Itu juga berarti satu byte dapat menampung 256 nilai berbeda yang mungkin (semua nilai dari 0 hingga 255).

Ketika sebuah byte ditulis, bit yang paling tidak signifikan—yaitu, bit yang merepresentasikan pangkat 2 terkecil—biasanya ditulis paling kanan (bit 0, merepresentasikan 20). Bit yang paling signifikan—bit yang merepresentasikan pangkat 2 terbesar—biasanya ditulis paling kiri (bit 7, merepresentasikan 27). Dalam 10000000, misalnya, angka 1 mewakili 27 yang diaktifkan, sehingga byte tersebut memiliki nilai desimal 128. Ini mengasumsikan kita bekerja dengan bilangan bulat tak bertanda—bilangan bulat yang tidak mungkin negatif. Tanda berada di luar cakupan pengantar dasar biner ini.

Beberapa tipe data direpresentasikan menggunakan lebih dari satu byte. Misalnya, pada mikroprosesor 64-bit seperti yang mungkin menggerakkan komputer Anda, bilangan bulat sering disimpan menggunakan 64 bit (8 byte). Nilai maksimum bilangan 64-bit dalam biner adalah:

1111111111111111111111111111111111111111111111111111111111111111

Itu diterjemahkan menjadi 18.446.744.073.709.551.615 dalam desimal.

Untuk beberapa proyek dalam buku ini, kita perlu memanipulasi byte pada tingkat bit. Sisa lampiran ini membahas beberapa operasi umum untuk melakukannya.

9.2 OPERASI BITWISE UMUM

Operasi bitwise memanipulasi nilai pada tingkat digit biner individual. Ini berarti bekerja dengan angka 1 dan 0. Semua mikroprosesor menyertakan instruksi untuk melakukan operasi bitwise, dan Python memiliki operator untuk memanfaatkan instruksi mikroprosesor ini. Tabel kebenaran, yang menunjukkan hasil benar/salah dari fungsi logika berdasarkan kombinasi input yang berbeda, dapat membantu dalam memahami operasi bitwise. Untuk membuatnya lebih praktis dan dapat diterapkan pada penggunaan operasi ini di Python, tabel yang menyertai setiap operasi akan menunjukkan nilai biner alih-alih benar dan salah.

Geser Kiri (<<)

Alih-alih memikirkan data biner kita sebagai angka, untuk sesaat pikirkan saja sebagai kumpulan angka 1 dan 0. Bayangkan angka 0 sebagai ruang kosong dan angka 1 sebagai ruang yang terisi. Bagaimana jika kita ingin menggeser semua angka 1 ke kiri sebanyak satu spasi? Itulah tugas pergeseran kiri, yang di Python diwakili oleh operator <<.

Pergeseran kiri meninggalkan celah pada bit paling tidak signifikan (posisi bit 0). Dengan pergeseran kiri di Python, kita mengisi celah itu dengan angka 0. Karena bilangan bulat Python memiliki panjang sembarang (tidak ada panjang maksimum), kita tidak dapat memindahkan angka 1 dari ujung dengan menggesernya ke kiri. Angka tersebut hanya bertambah satu digit. Misalnya, 1010 yang digeser ke kiri sebanyak 1 menjadi 10100, bukan 0100. Kita juga dapat menggeser ke kiri lebih dari satu tempat, sehingga 1010 yang digeser ke kiri sebanyak tiga menjadi 1010000. Tabel 9.1 menunjukkan hasil dari beberapa pergeseran kiri.

Tabel 9.1: Contoh Pergeseran Kiri

$\mathbf{A}$	$\mathbf{A << 1}$	$\mathbf{A << 3}$
0	0	0
1	10	1000
1010	10100	1010000

Pada baris pertama tabel, di mana angka 0 digeser, Anda mungkin berpikir bahwa itu akan menjadi 00 dan 0000, tetapi pada kenyataannya itu sama dengan 0. Sebaliknya, Anda dapat menganggap pergeseran kiri hanya memindahkan angka 1. Jika tidak ada angka 1, maka pergeseran tersebut pada dasarnya tidak melakukan apa pun.

Operator pergeseran kiri Python didahului oleh hal yang digeser dan diikuti oleh bilangan bulat yang menunjukkan berapa banyak tempat yang akan digeser. Berikut contoh singkat penggunaannya:

```
>>> bin(0b1010 << 3)
'0b1010000'
```

Operator pergeseran biasanya digunakan untuk memindahkan satu atau beberapa bit agar sejajar dengan nilai biner lain dalam kombinasi dengan operator bitwise lain yang akan kita pelajari sebentar lagi.

Pergeseran Kanan (>>)

Pergeseran kanan hampir sama dengan pergeseran kiri, kecuali angka 1 bergerak ke kanan, bukan ke kiri. Jika angka 1 bergerak keluar dari ujung (melewati posisi bit 0), maka angka tersebut "hilang". Tidak ada pembungkus. Misalnya, 1001 yang digeser ke kanan satu posisi menjadi 100, bukan 1100. Kita juga dapat menggeser ke kanan lebih dari satu posisi, sehingga 1001 yang digeser ke kanan tiga posisi menjadi 1. Python memiliki operator >> untuk melakukan pergeseran kanan. Tabel 9.2 menunjukkan beberapa contoh pergeseran kanan.

Tabel 9.2: Contoh Pergeseran Kanan

A	A >> 1	A >> 3
0	0	0
1	0	0
1010	101	1

Pada baris kedua tabel, angka 1 digeser "ke luar ujung" dan tidak ada angka 1 yang tersisa, sehingga hasilnya adalah 0.

ATAU (|)

Dengan operasi OR bitwise antara dua nilai, jika salah satu nilai adalah 1, maka hasilnya akan menjadi 1. Jika tidak ada nilai yang merupakan 1, maka hasilnya akan menjadi 0. Operasi ini dilakukan pada digit biner yang berada di tempat digit yang sama di antara kedua nilai, satu tempat digit pada satu waktu. Bayangkan angka-angka tersebut berjejer, satu di bawah yang lain. Kemudian, operasi OR dilakukan di setiap kolom, satu kolom pada satu waktu, seperti ini:

```
1010
0110
----
1110
```

Cobalah menghitung sendiri baris terakhir di bagian bawah dengan mengikuti aturan pada paragraf sebelumnya. Aturan-aturan tersebut juga dirangkum dalam Tabel 9.3.

Tabel 9.3: Contoh OR

A	B	A | B
0	0	0
0	1	1
1	0	1
1	1	1
1010	0110	1110

Python memiliki operator | untuk melakukan operasi OR bitwise. Berikut contoh singkat penggunaannya pada beberapa bilangan biner di Python:

```
>>> bin(0b1010 | 0b0110)
'0b1110'
```

Salah satu penggunaan umum operasi OR bitwise dalam pemrograman tingkat rendah adalah untuk menggabungkan dua nilai bersama-sama dalam kombinasi dengan operator pergeseran. Misalnya, katakanlah kita memiliki satu nibble (satu nibble adalah 4 bit) yang mewakili setengah dari sebuah byte dan nibble lain yang mewakili setengah lainnya dari byte tersebut. Kita ingin menggabungkannya untuk menghasilkan byte lengkap. Mungkin nibble A adalah 1001 dan nibble B adalah 0110, dan kita ingin nibble A menjadi setengah pertama dan nibble B menjadi setengah kedua dari byte yang dihasilkan. Kode untuk menggabungkannya mungkin terlihat seperti ini:

```
>>> a = 0b1001
>>> b = 0b0110
>>> c = (a << 4) | b
>>> bin(c)
'0b10010110'
```

Dengan `(a << 4) | b` kita menggeser a ke kiri 4 tempat dan kemudian melakukan operasi OR pada nilai-nilainya dengan b. Ingat bahwa ketika kita menggeser ke kiri, angka `0` akan masuk dari kanan, jadi a yang digeser ke kiri sebanyak 4 menjadi `10010000`. Kemudian, jika kita mensejajarkan a dan b dan melakukan operasi OR pada keduanya, kita mendapatkan:

```
10010000
    0110
--------
10010110
```

Byte yang dihasilkan, `10010110`, memiliki huruf a asli di empat digit kiri dan huruf b asli di empat digit kanan. Jika Anda melihat ini untuk pertama kalinya, mungkin tampak sedikit abstrak. Misalnya, Anda mungkin bertanya-tanya mengapa angka-angka tersebut disimpan hanya dalam 4 bit sejak awal. Untuk memberikan salah satu dari beberapa alasan, dalam proyek-proyek buku ini kita melihat beberapa skenario di mana kita ingin menghemat ruang dengan menggabungkan nilai-nilai yang membutuhkan kurang dari 8 bit ke dalam byte yang sama. Misalnya, mikroprosesor 8-bit sering memiliki register flag 1-byte di mana setiap bit individual mewakili flag yang berbeda yang dapat aktif atau nonaktif. Itu jauh lebih ekonomis daripada menggunakan byte terpisah untuk setiap flag. Demikian pula, beberapa format file menyimpan nilai yang membutuhkan kurang dari 1 byte pada byte yang sama untuk menghemat ruang disk.

AND (&)

Operasi bitwise AND mengembalikan 1 jika kedua operand adalah 1; Jika tidak, ia akan mengembalikan nilai 0. Berikut contoh penggunaan operan yang sama seperti yang kita lihat pada operasi bitwise OR:

```
1010
0110
----
0010
```

Hanya bit yang sejajar dengan dua angka 1 yang menghasilkan angka 1. Tabel 9.4 merangkum cara kerja operasi bitwise AND.

Tabel 9.4: Contoh Operasi AND

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1
1010	0110	0010

Python memiliki operator & untuk melakukan operasi bitwise AND. Berikut contoh singkat penggunaannya pada beberapa bilangan biner:

```
>>> bin(0b1010 & 0b0110)
'0b10'
```

Perhatikan bahwa output memotong angka `0` di depan karena tidak diperlukan untuk mewakili angka hasil (2 dalam desimal). Dengan kata lain, `0010` dalam biner adalah 2 dalam desimal sama seperti `10` dalam biner adalah 2 dalam desimal.

Salah satu penggunaan umum operasi AND bitwise dalam pemrograman tingkat rendah adalah untuk memastikan hasil akhir hanya mencakup beberapa bit dari hasil sebelumnya. Misalnya, anggaplah kita hanya peduli dengan 4 bit paling kanan dalam byte `10011110`. Kita dapat melakukan operasi AND byte tersebut dengan 1111 untuk memastikan hanya 4 bit paling kanan yang ada dalam hasil akhir. Kode tersebut mungkin terlihat seperti ini:

```
>>> a = 0b10011110
>>> b = 0b1111
>>> c = a & b
>>> bin(c)
'0b1110'
```

Mari kita lihat operasi ini dengan bit-bit yang sudah tersusun rapi:

```
10011110
    1111
--------
    1110
```

Kita sering menggunakan teknik ini dengan satu bit dalam proyek emulator kita saat bekerja dengan flag. Kita hanya perlu mengetahui flag tunggal tersebut dan apakah nilainya `1` atau `0` (benar atau salah).

XOR (^)

XOR bitwise ("eksklusif atau") mengembalikan nilai 1 jika operandnya berbeda (satu `1` dan satu `0`); jika tidak, ia mengembalikan nilai `0.` Berikut contohnya menggunakan operand yang sama yang telah kita lihat pada OR dan AND bitwise:

```
1010
0110
----
1100
```

Tabel 9.5 merangkum cara kerja XOR.

Tabel 9.5: Contoh XOR

A	B	A & B
0	0	0
0	1	1
1	0	1
1	1	0
1010	0110	1100

Python memiliki operator ^ untuk melakukan operasi XOR bitwise. Berikut contoh singkat penggunaannya pada beberapa bilangan biner di Python:

```
>>> bin(0b1010 ^ 0b0110)
'0b1100'
```

XOR adalah operasi yang sangat ampuh. Operasi ini mendasari skema enkripsi yang tidak dapat dipecahkan yang dikenal sebagai one-time pad. Anda juga dapat membalik bit dengan melakukan XOR dengan 1: jika Anda melakukan XOR antara 1 dengan 1, maka akan menjadi 0, tetapi jika Anda melakukan XOR antara 0 dengan 1, maka akan menjadi 1. Setiap bit yang Anda XOR dengan 1 akan menjadi kebalikan dari keadaan sebelumnya. Inilah cara kerja menggambar di layar dalam proyek CHIP-8 di Bab 5.

Komplemen (~)

Komplemen adalah operasi bitwise yang paling sederhana: ia menukar semua 1 dengan 0 dan semua 0 dengan 1, seperti yang ditunjukkan pada Tabel 9.6.

Tabel 9.6: Contoh Komplemen

A	**~ A**
1	0
0	1
1010	0101
1010	0101

Python memiliki operator ~ untuk mengambil komplemen biner. Kita tidak banyak menggunakan komplemen dalam buku ini, kecuali di satu tempat kecil pada emulator NES di Bab 6.

DAFTAR PUSTAKA

Abdurazakov, M. M., Aziyev, R. A., & Muhidinov, M. G. (2017). The principles of constructing a methodical system for teaching computer science in general educational school. *Revista ESPACIOS*, *38*(40).

Banâtre, J. P., Fradet, P., & Radenac, Y. (2005). Principles of chemical programming. *Electronic Notes in Theoretical Computer Science*, *124*(1), 133-147.

Bernonville, S., Kolski, C., Leroy, N., & Beuscart-Zéphir, M. C. (2010). Integrating the SE and HCI models in the human factors engineering cycle for re-engineering Computerized Physician Order Entry systems for medications: Basic principles illustrated by a case study. *International journal of medical informatics*, *79*(4), e35-e42.

Bézivin, J. (2004). In search of a basic principle for model driven engineering. *Novatica Journal*, *5*(2), 21-24.

Bolch, G., Greiner, S., De Meer, H., & Trivedi, K. S. (2006). *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. John Wiley & Sons.

Bondarev, A., & Efanov, V. (2019, December). The Principles of Forming of the Mathematical Model of Nanoelectronic Components of Quantum Computer Systems with Memresistance Branches. In *21st International Workshop on Computer Science and Information Technologies (CSIT 2019)* (pp. 17-22). Atlantis Press.

Chae, E., Choi, J., & Kim, J. (2024). An elementary review on basic principles and developments of qubits for quantum computing. *Nano Convergence*, *11*(1), 11.

Cortina, T. J. (2007). An introduction to computer science for non-majors using principles of computation. *Acm sigcse bulletin*, *39*(1), 218-222.

De Oliveira, R. R. L., Albuquerque, D. A. C., Cruz, T. G. S., Yamaji, F. M., & Leite, F. L. (2012). Measurement of the nanoscale roughness by atomic force microscopy: basic principles and applications. *Atomic force microscopy-imaging, measuring and manipulating surfaces at the atomic scale*, *3*, 147-174.

Denning, P. J. (2004, March). Great principles in computing curricula. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education* (pp. 336-341).

Dumas II, J. D. (2018). *Computer architecture: Fundamentals and principles of computer design*. CRC press.

Ejaz, A., Ali, S. A., Ejaz, M. Y., & Siddiqui, F. A. (2019). Graphic user interface design principles for designing augmented reality applications. *International Journal of Advanced Computer Science and Applications*, *10*(2).

El-Abbassy, A., Muawad, R., & Gaber, A. (2010). Evaluating agile principles in CS Education. *International Journal of Computer Science and Network Security*, *10*(10), 19-28.

Fincher, S., Petre, M., & Clark, M. (Eds.). (2001). *Computer science project work: principles and pragmatics*. Springer Science & Business Media.

Freeman, J., Magerko, B., & Verdin, R. (2015, February). Computer science principles with EarSketch. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (pp. 710-710).

Hamlen, K., Sridhar, N., Bievenue, L., Jackson, D. K., & Lalwani, A. (2018, February). Effects of teacher training in a computer science principles curriculum on teacher and student skills, confidence, and beliefs. In *Proceedings of the 49th ACM technical symposium on computer science education* (pp. 741-746).

Hare, K. (2024). *Computer Science Principles: The Foundational Concepts of Computer Science-For AP® Computer Science Principles*. Yellow Dart Publishing.

Havard, D. D., & Howard, K. (2019). All Advanced Placement (AP) computer science is not created equal: A comparison of AP Computer Science A and Computer Science Principles. *Journal of Computer Science Integration*, *2*(1), 2.

Hergert, W., Ernst, A., & Däne, M. (Eds.). (2004). *Computational materials science: from basic principles to material properties* (Vol. 642). Springer Science & Business Media.

Householder, A. S. (2006). *Principles of numerical analysis*. Courier Corporation.

Hu, F. (2013). *Telehealthcare Computing and Engineering: Principles and Design*. CRC Press.

Kiili, K., De Freitas, S., Arnab, S., & Lainema, T. (2012). The design principles for flow experience in educational games. *Procedia Computer Science*, *15*, 78-91.

Ladner, R. E., Stefik, A., Naumann, J., & Peach, E. (2020, March). Computer science principles for teachers of deaf students. In *2020 Research on Equity and Sustained Participation in Engineering, Computing, and Technology (RESPECT)* (Vol. 1, pp. 1-4). IEEE.

Mannhold, R., Kubinyi, H., Timmerman, H., Folkers, G., Höltje, H. D., & Vacca, J. (2001). Methods and principles in medicinal chemistry. *Molecular Modeling. Basic Principles and Applications*, *5*, 23-36.

Newell, A., & Simon, H. A. (2007). Computer science as empirical inquiry: Symbols and search. In *ACM Turing award lectures* (p. 1975).

Nisan, N., & Schocken, S. (2021). *The elements of computing systems: building a modern computer from first principles*. MIT press.

Pozinkevych, R. (2021). Logical principles in ternary mathematics. *Asian Journal of Research in Computer Science*, *7*(3), 49-54.

Saleem, S., Popov, O., & Bagilli, I. (2014). Extended abstract digital forensics model with preservation and protection as umbrella principles. *Procedia Computer Science*, *35*, 812-821.

Saltzer, J. H., & Kaashoek, M. F. (2009). *Principles of computer system design: an introduction*. Morgan Kaufmann.

Sanghera, P. (2011). *Quantum physics for scientists and technologists: Fundamental principles and applications for biologists, chemists, computer scientists, and nanotechnologists*. John Wiley & Sons.

Sax, L. J., Newhouse, K. N., Goode, J., Nakajima, T. M., Skorodinsky, M., & Sendowski, M. (2022). Can computing be diversified on "principles" alone? Exploring the role of AP Computer Science courses in students' major and career intentions. *ACM Transactions on Computing Education (TOCE)*, *22*(2), 1-26.

Snyder, L., Barnes, T., Garcia, D., Paul, J., & Simon, B. (2012). The first five computer science principles pilots: summary and comparisons. *ACM Inroads*, *3*(2), 54-57.

Tarakanov, A. O., Skormin, V. A., & Sokolova, S. P. (2003). *Immunocomputing: principles and applications*. Springer Science & Business Media.

Taylor, B., & Azadegan, S. (2006, September). Threading secure coding principles and risk analysis into the undergraduate computer science and information systems curriculum. In *Proceedings of the 3rd annual conference on Information security curriculum development* (pp. 24-29).

Tkachenko, R., & Izonin, I. (2018, January). Model and principles for the implementation of neural-like structures based on geometric data transformations. In *International Conference on Computer Science, Engineering and Education Applications* (pp. 578-587). Cham: Springer International Publishing.

Veletsianos, G., Beth, B., Lin, C., & Russell, G. (2016). Design principles for thriving in our digital world: A high school computer science course. *Journal of Educational Computing Research*, *54*(4), 443-461.

PRINSIP DASAR ILMU KOMPUTER :

Interpreter, Seni Komputasi, Algoritma, Emulator, Pembelajaran Mesin (ML in Python)

Dr. Ir. Agus Wibowo, M.Kom, M.Si, MM

BIO DATA PENULIS

Penulis memiliki berbagai disiplin ilmu yang diperoleh dari Universitas Diponegoro (UNDIP) Semarang. dan dari Universitas Kristen Satya Wacana (UKSW) Salatiga. Disiplin ilmu itu antara lain teknik elektro, komputer, manajemen dan ilmu sosiologi. Penulis memiliki pengalaman kerja pada industri elektronik dan sertifikasi keahlian dalam bidang Jaringan Internet, Telekomunikasi, Artificial Intelligence, Internet Of Things (IoT), Augmented Reality (AR), Technopreneurship, Internet Marketing dan bidang pengolahan dan analisa data (komputer statistik).

Penulis adalah pendiri dari Universitas Sains dan Teknologi Komputer (Universitas STEKOM) dan juga seorang dosen yang memiliki Jabatan Fungsional Akademik Lektor Kepala (Associate Professor) yang telah menghasilkan puluhan Buku Ajar ber ISBN, HAKI dari beberapa karya cipta dan Hak Paten pada produk IPTEK. Sejak tahun 2023 penulis tercatat sebagai Dosen luar biasa di Fakultas Ekonomi & Bisnis (FEB) Universitas Diponegoro Semarang. Penulis juga terlibat dalam berbagai organisasi profesi dan industri yang terkait dengan dunia usaha dan industri, khususnya dalam pengembangan sumber daya manusia yang unggul untuk memenuhi kebutuhan dunia kerja secara nyata.

PENERBIT :
YAYASAN PRIMA AGUS TEKNIK
Jl. Majapahit No. 605 Semarang
Telp. (024) 6723456. Fax. 024-6710144
Email : penerbit_ypat@stekom.ac.id

ISBN 978-634-7227-85-0 (PDF)
9 786347 227850