Eric Roberts
CSCI 121

Handout #50
November 14, 2018

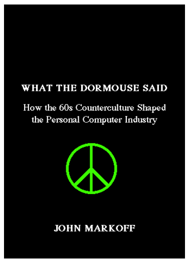
Data-Driven Programs

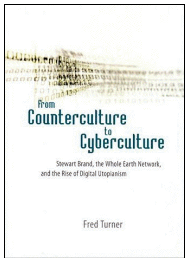
Data-Driven Programs

Eric Roberts
CS 106AJ
November 17, 2017

Computing and the Counterculture

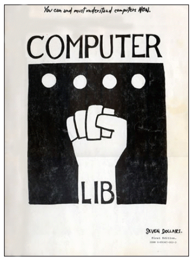
Two recent books argue that the personal computing revolution owes as much to the counterculture of the 1960s as it does to the technological strength and entrepreneurial spirit of Silicon Valley.

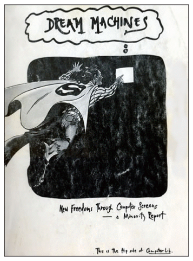

WHAT THE DORMOUSE SAID
How the 60s Counterculture Shaped the Personal Computer Industry
JOHN MARKOFF


From Counterculture to Cyberculture
Stewart Brand, the Whole Earth Network, and the Rise of Digital Utopianism
Fred Turner

Ted Nelson's Cyberspace Dreams

The countercultural vision comes across particularly clearly in the two-sided book *Computer Lib/Dream Machines* which was written by cyberspace visionary Ted Nelson in 1974.





Data-Driven Programs

- In most programming languages, data structures are easier to manipulate than code. As a result, it is often useful to design applications so that as much of their behavior as possible is represented as data rather than in the form of methods. Programs that work this way are said to be ***data driven***.
- In a data-driven system, the actual program (which is usually called a ***driver***) is typically very small. Such driver programs operate in two phases:
  1. Read data from a file into a suitable internal data structure.
  2. Use the data structure to control the flow of the program.
- To illustrate the idea of a data-driven system, most of this lecture focuses on writing a "teaching machine" of the sort that Ted Nelson discusses (mostly critically) in *Dream Machines*.

The Course Data File

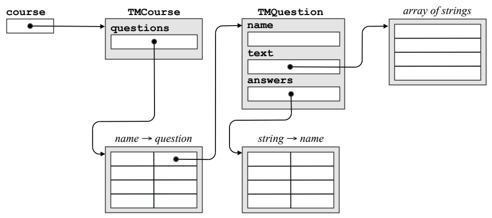
In the teaching machine application, the course designer—who is an expert in the domain of instruction and not necessarily a programmer—creates a data file that serves as the driver. The general format of the file is shown on the left, and a specific example of a question and its answers appears on the right.

identifying name for the first question
text of the first question
`-----`
response$_1$ **:** *name of next question*
response$_2$ **:** *name of next question*
response$_3$ **:** *name of next question*
.
.
.
. . . *other question/answer entries* . . .

```
RemQ1
What is the value of 17 % 4?
   a. 0
   b. 1
   c. 3
   d. 4
-----
a: RemQ2
0: RemQ2
b: PrecQ1
1: PrecQ1
c: RemQ2
3: RemQ2
d: RemQ2
4: RemQ2
```

Choosing an Internal Representation

The first step in building the teaching machine is to design a set of classes that can represent the data and relationships in the file. All of the relevant data should be accessible from a single structure that contains all relevant information in a nested series of classes.



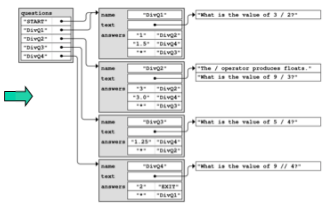
Converting External to Internal Form

```
DivQ1
What is the value of 3 / 2?
-----
1: DivQ2
1.5: DivQ4
*: DivQ3

DivQ2
The / operator produces floats.
What is the value of 9 / 3?
-----
3: DivQ2
3.0: DivQ4
*: DivQ3

DivQ3
What is the value of 5 / 4?
-----
1.25: DivQ4
*: DivQ2

DivQ4
What is the value of 9 // 4?
-----
2: EXIT
*: DivQ1
```



The **TeachingMachine** Program

```
# File: TeachingMachine.py

from TMCourse import TMCourse

def TeachingMachine():
    course = readCourseFile()
    course.run()

def readCourseFile():
    while True:
        try:
            filename = input("Enter course name: ")
            with open(filename + ".txt") as f:
                return TMCourse.readCourse(f)
        except IOError:
            print("Please enter a valid course name.")

# Startup code

if __name__ == "__main__":
    TeachingMachine()
```

The **TMCourse** Class

```
# File: TMCourse.py

from TMQuestion import TMQuestion

class TMCourse:

    def __init__(self, questions):
        self._questions = questions

    def run(self):
        current = "START"
        while current != "EXIT":
            question = self._questions[current]
            for line in question.getText():
                print(line)
            answer = input("> ").strip().upper()
            next = question.lookupAnswer(answer)
            if next is None:
                print("I don't understand that response.")
            else:
                current = next
```

The **TMCourse** Class

```
# Implementation notes
# --------------------
# To make sure that the course starts at the first
# question, this method always includes an entry
# labeled "START" in the question table.

    @staticmethod
    def readCourse(f):
        questions = { }
        while True:
            question = TMQuestion.readQuestion(f)
            if question is None: break
            if len(questions) == 0:
                questions["START"] = question
            name = question.getName()
            questions[name] = question
        return TMCourse(questions)
```

The **TMQuestion** Class

```
# File: TMQuestion.py

MARKER = "-----"

class TMQuestion:
    def __init__(self, name, text, answers):
        self._name = name
        self._text = text
        self._answers = answers

    def getName(self):
        return self._name

    def getText(self):
        return self._text

    def lookupAnswer(self, response):
        next = self._answers.get(response, None)
        if next is None:
            next = self._answers.get("*", None)
        return next
```

The **TMQuestion** Class

```
    @staticmethod
    def readQuestion(f):
        name = f.readline().rstrip()
        if name == "":
            return None
        text = [ ]
        while True:
            line = f.readline().rstrip()
            if line == MARKER: break
            text.append(line)
        answers = { }
        while True:
            line = f.readline().rstrip()
            if line == "": break
            colon = line.find(":")
            if colon == -1:
                raise ValueError("Missing colon in " + line)
            response = line[:colon].strip().upper()
            next = line[colon + 1:].strip()
            answers[response] = next
        return TMQuestion(name, text, answers)
```