25 REM PORTS TO OTHER PLATFORMS

APPLESOFT BASIC AND TANDY COLOR BASIC
PERL AND JAVASCRIPT: MODERN ONE-LINERS
PATH: MAZE AS PERVERSE PROGRAM
WHAT PORTING REVEALS

Adapting a program from one hardware system to another is "porting," a term derived from the Classical Latin portāre—to carry or bear, not unlike the carrying across (trans + lātus) of translation. A port is borne from one platform to another, and the bearer is the programmer, who must gather up the details of the original and find places for them amid the particulars of the destination, attempting to identify and preserve the program's essential properties. The translator faces these same sorts of problems when encountering a text, and such problems are particularly acute when the text is a poem. Where does the poetry of the poem lie? In its rhythm? Its rhyme? Its diction? Its constraints? Its meanings? Which of these must be carried over from one language to another in order to produce the most faithful translation?

In *Nineteen Ways of Looking at Wang Wei*, a study of the act and art of translation, Eliot Weinberger (1987) reads nineteen versions of a four-line, 1,200-year-old poem by the Chinese master Wang Wei, attentive to the way translators have reinterpreted the poem over the centuries, even as they attempted to be faithful to the original. With a single word, a translator may create a perspective unseen in Wei's original, radically shift the mood of the poem, or transform it into complete tripe. Many times these changes come about as the translator tries to improve the original in some way. Yet translation, Weinberger writes, "is dependent on the dissolution of the translator's ego: an absolute humility toward the text" (17).

The programmer who ports faces similar challenges. What must be preserved when a program is carried across to a new platform: The program's interface? Its usability? Its gameplay? Its aesthetic design? The underlying algorithm? The effects of the constraints of the original? And should the programmer try to improve the original? The ethos of adaptation will vary from project to project and programmer to programmer; what a programmer chooses to prioritize will help to determine the qualities of the final port and its relationship to the original program.

In this remark, a number of ports—translations—are presented. These are ports from Commodore 64 BASIC to other platforms and languages, developed specifically for this book. Other ports can be found elsewhere in this book. By striving to design accurate adaptations, and to capture qualities of the original code as well as the output, nuances of the original that might otherwise be overlooked can be revealed. Just as the variations of `10 PRINT` in the previous remark illustrate the consequences of choosing


```
10 PRINT CHR$(205.5+RND(1)); : GOTO 10
```

one particular set of parameters among the many that were possible on the Commodore 64, ports of `10 PRINT` can highlight the constraints and affordances of individual platforms. The ports provide a tightly focused comparison of the Commodore 64 to other systems, emphasizing the unique suitability of the Commodore 64 for this particular program.

APPLESOFT BASIC AND TANDY COLOR BASIC

Applesoft BASIC is one of two standard BASIC implementations for the Apple II; Applesoft is the one that supports floating point math and seems very similar to Commodore 64 BASIC. The Apple II family of computers was of the same era and uses the same processor as did the Commodore 64, the MOS 6502. Applesoft BASIC, like Commodore 64 BASIC, was written by Microsoft and based on its 6502 BASIC, a version (as discussed in the chapter on BASIC) that derives from Microsoft's Altair BASIC. The Apple II computers and the Commodore 64 were really quite alike, almost as if they were siblings separated by corporate circumstance.

This makes the Apple II a good starting point for a series of `10 PRINT` ports. The same BASIC statements and keywords can be used in a version for this computer, and the same sort of scrolling will push the maze continually up the screen.

On the Apple II, however, the slash and backslash characters must serve as the maze walls, since the PETSCII diagonal-line characters are not available. The codes for those Apple II characters are not adjacent; they have the ASCII values 47 and 92. This means that a more elaborate expression for the selection of a character must be used. The first step is selecting the value 0 or 1. This first selection is accomplished in `INT(RND(1)*2)`, which in the inner expression produces a floating point number that is at least 0 and less than 2, such as 0.492332 or 1.987772; then, using INT, this value is truncated to either 0 or 1. The next step is to multiply that value by 45 and add 47 so that either 47 or 92 results. This is a reasonably simple way to make this selection, but, as with certain Commodore 64 BASIC variants, the code that is needed is more elaborate and less pleasing than in the canonical `10 PRINT`:

```
10 PRINT CHR$(47+(INT(RND(1)*2)*45)); : GOTO 10
```

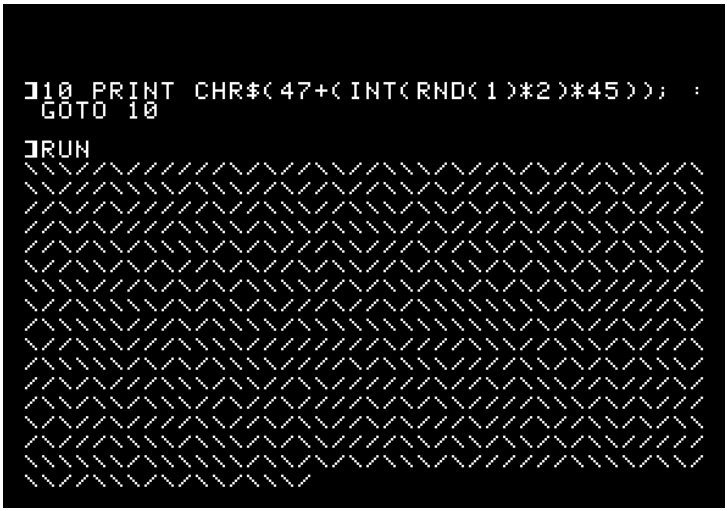


Figure 25.1

Screen capture from the Apple II port of `10 PRINT`.

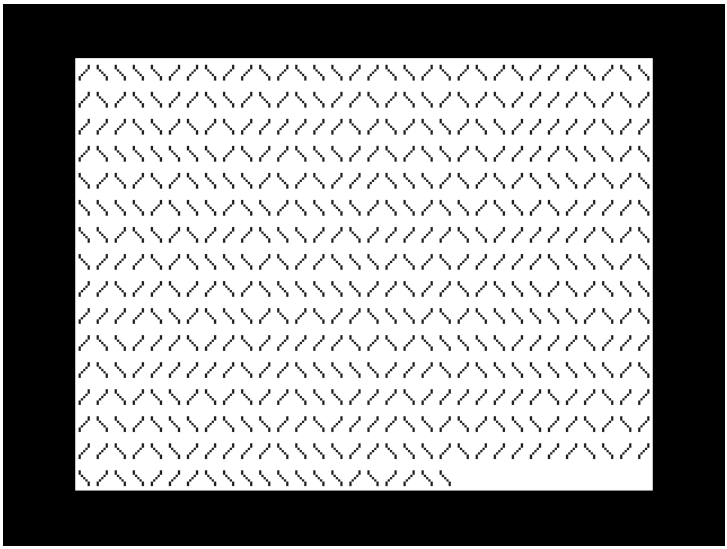


Figure 25.2

Screen capture from the TRS-80 Color Computer port of `10 PRINT`.

```
(54)    10 PRINT CHR$(205.5+RND(1)); : GOTO 10
```

The output of the program is less satisfying, too (figure 25.1). Although the "/" and "\" characters on Apple II computers are exactly diagonal, they do not span the entire square that bounds a character. This means that the "walls" do not meet either horizontally or vertically. Each Apple II character is five pixels wide and seven pixels tall, so the perfect diagonals of the slash and backslash have a pixel of empty space at the top and another at the bottom. In any case, Apple II characters cannot be drawn directly against one another, as all characters on the system are printed with a one-pixel-wide space on either side of them and a one-pixel space below.

This space between characters is even more evident in the port of `10 PRINT` to another competitor of the Commodore 64 in the 1980s—the TRS-80 Color Computer (or "CoCo"), sold through Radio Shack. If the Apple II was the Commodore 64's sibling, raised by another corporation, then the Color Computer, with the Motorola 6809 and a different version of Microsoft BASIC, was the eccentric cousin. Just as with Applesoft BASIC, the Color BASIC port of `10 PRINT` requires the use of ASCII characters 47 and 92; one significant change, however, must be made to the program:

```
10 PRINT CHR$(47+INT(RND(0)*2)*45);:GOTO 10
```

Note the change from `RND(1)` to `RND(0)`. This revision is due to the Color Computer's implementation of `RND`, which diverges quite a bit from that in other BASICs. In a move to make the `RND` command more intuitive, the TRS-80 chooses a random number between 1 and the argument, X. So `RND(6)` chooses a random number between 1 and 6. `RND(1)` in Color BASIC will only ever choose the number 1, making for a decidedly nonrandom pattern. `RND(0)`, however, selects a floating point number between 0 and 1, which, multiplied by 2, can serve as the numerical basis for the random pattern. The execution of the program reveals, though, that randomness is not the only essential element of `10 PRINT` (figure 25.2). Even when compared to the Apple II, the TRS-80's text display is poorly suited for the transformation of typographical symbols into graphical patterns. The Color Computer's slash and backslash characters each occupy a 5 × 7 region on a larger grid of 8 × 12, leaving so much space between the characters that they can never resolve themselves into the suggestion of a connected pattern, much less a maze.

While the Apple II and Color Computer had many interesting BASIC programs written for them and shares features with the Commodore 64, the way these computers handle text display means that neither can host a one-line BASIC version of `10 PRINT` that is as satisfying as the Commodore version.

PERL AND JAVASCRIPT: MODERN ONE-LINERS

Perl and JavaScript programs were devised that are parts of `10 PRINT` and output the ASCII slash and backslash characters. The JavaScript program is chiefly interesting because it presents a graphical, or typographical, problem that is even worse than the ones seen on the Apple II and the Tandy Color Computer. The default font on a Web page, viewed in a graphical user interface browser, is proportional—different letterforms have different widths. While slash and backslash are the same width, differences in kerning mean that the pair "/\" is wider than either "//" or "\\". So the two symbols do not line up in a grid, and the result is even less like a maze.

A first version of the Perl one-liner follows; it's shown in figure 25.3:

```
while (print int(rand(2)) ? "/" : "\\") {}
```

The "\" character (the backslash) is used in combination with another character in Perl to print special characters such as the newline, which is indicated as "\n". (The same is true in JavaScript.) Because of this, it is necessary to use "\\" to print a single backslash character. This Perl port uses the `while` construct to create an infinite loop. The condition of this loop prints either "/" or "\" at random. The `print` statement, which should always succeed, will return a value of 1, corresponding to true—so the loop will always continue. The body of the `while` loop is empty; nothing else except printing a character needs to be done, and that is already accomplished within the condition. The resulting output is similar to that of the Apple II program: random slashes are produced that line up in a grid but don't meet horizontally or vertically.

There are a few ways to tweak this code to make it more like `10 PRINT` in form and to have it produce output that is more like `10 PRINT`'s. First, the somewhat obscure but more `GOTO`-like `redo` statement can be

```
(56)    10 PRINT CHR$(205.5+RND(1)); : GOTO 10
```

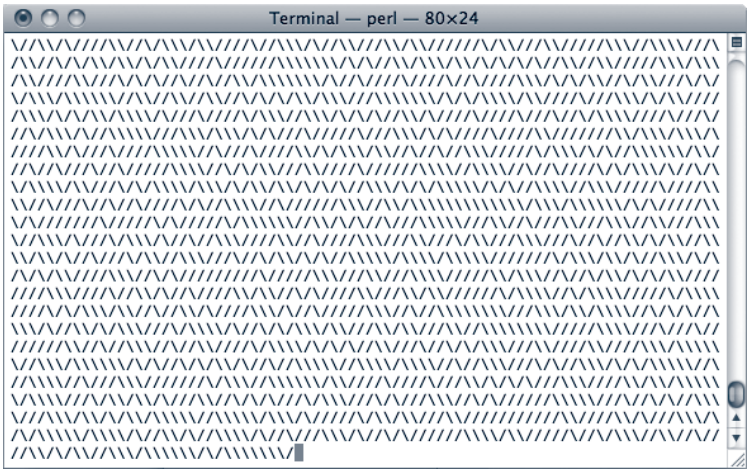


Figure 25.3

Screen capture of the ASCII Perl port of `10 PRINT`, which uses the slash and backslash to approximate the diagonal lines.

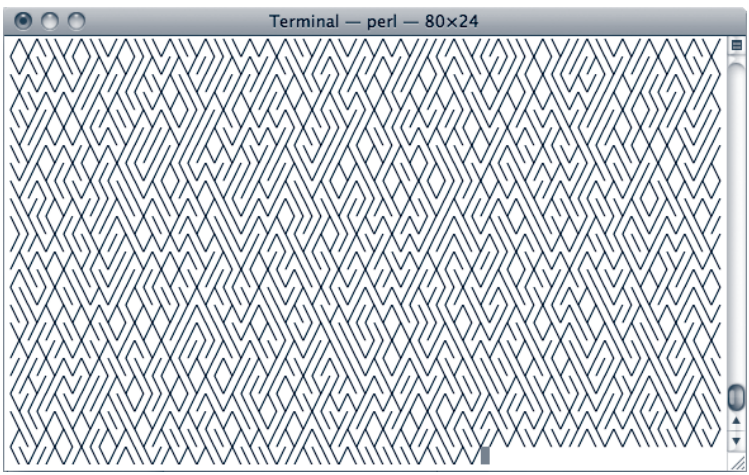


Figure 25.4

Screen capture of the Unicode Perl port of `10 PRINT`, which uses characters 9585 and 9586 to better approximate the PETSCII characters.

used, causing the program to loop back to the beginning of its code block, which is enclosed in curly braces, "{" and "}". Second, the Unicode characters 9585 and 9586 can be used to build the maze. These characters are the two diagonal lines, similar to the PETSCII characters on the Commodore 64, and like those characters they are also adjacent. This means that a trick similar to `205.5+RND(1)` can be used to randomly select between them—in this case, `9585.5+rand`. That expression is used as an argument to Perl's `chr` function, just as the original BASIC program wraps it in the `CHR$` function. Finally, to avoid the production of error messages, a statement needs to be included that tells Perl it can output characters in Unicode. That statement could go outside or inside the loop; the program just runs slightly slower, which is probably desirable, if it is placed inside and executed each time:

```
{binmode STDOUT,"utf8";print chr(9585.5+rand);redo}
```

While the original `10 PRINT` produces a maze with gaps or thin connections at each grid point, this maze (see figure 25.4) has what look like overlaps at each of these junctures. Nevertheless, the use of Unicode's similar characters does a great deal to enhance the appearance of the output.

PATH: MAZE AS PERVERSE PROGRAM

While computer users may think of programming languages as relatively straightforward instruments used to produce increasingly complex or efficient tools and experiences, `10 PRINT` begins to show that code itself can have aesthetic features.

Some programmers choose to reject—at least for a while—the values of clarity and efficiency in programming in favor of other values. While some of the techniques such programmers use rely on the exploitation of conventions in existing, "normal" programming languages, others involve the invention of entirely new languages with their own aesthetic properties. These "weird languages" (sometimes also called "esoteric languages") test the limits of programming language design and comment on programming languages themselves. One them is the unusual-looking language called PATH.

```
(58)   10 PRINT CHR$(205.5+RND(1)); : GOTO 10
```

The sort of weird languages Michael Mateas and Nick Montfort (2005) dub "minimalist" comment on the space of computation itself. As they put it, "Minimalist languages strive to achieve universality while providing the smallest number of language constructs possible. Such languages also often strive for syntactic minimalism, making the textual representation of programs minimal as well." The archetypical minimalist language is Brainfuck, which provides seven commands, each corresponding to a single character of punctuation.

Another style of weird language eschews the usual organization into lines of code and uses a two-dimensional space to hold a program's instructions. One such language is Piet, whose source code resembles abstract paintings (like those by its namesake, Piet Mondrian). Another is Befunge, which uses typographical symbols including "<," "v," and "^" to direct program flow.

PATH is a weird language that borrows from the conventions of Brainfuck and Befunge, offering a syntactically constrained language whose control flow takes place in a two-dimensional space. PATH has a natural connection to `10 PRINT` because the language uses the slash and backslash characters to control program flow. These symbols are reflectors in PATH. As the program counter travels around in 2D space, it bounces off the reflectors in the intuitive way.

In addition to "/" and "\," PATH uses "v," "^," "<," and ">" to move the flow conditionally, down, up, left, and right, if the current memory location is nonzero. Memory locations are arrayed on an infinite tape Turing style, and the program can increment and decrement the current memory focus.

Given PATH's strong typographical similarity to the output of `10 PRINT`, it is possible to implement a port of `10 PRINT` in PATH—a program that generates labyrinths by endlessly walking a labyrinth (figure 25.5).

When the program is run, the result is similar to figure 25.3. Confusing? The point of such a program, and such a programming language, is to confuse and amuse, of course. Without understanding the details of how this program works, one can still appreciate an intriguing property it has. The output of `10 PRINT` in PATH is itself a PATH program. This new program doesn't do anything very interesting; it simply moves the program counter around without producing any output. Still, it demonstrates a general idea: that programs are texts, and there is nothing to keep people from

Figure 25.5

The PATH port of `10 PRINT`, an actual computer program written in an intentionally perverse programming language.

```
(60)    10 PRINT CHR$(205.5+RND(1)); : GOTO 10
```

writing programs (such as the much less perverse compilers and interpreters that are in continual use) that accept programs as input and produce programs as output.

WHAT PORTING REVEALS

Porting a program is always an act of translation and adaptation. As such, porting reveals what in a program is particular to its source context, suggests many potential approaches to what is essential about the program, and explores how that essence may be portable to a specific target context. Each port is unique, whether to a related platform, to a modern scripting language, or even to a weird, minimalist language. Each involves different constraints, and once realized each offers different insights. Sometimes these insights are into the platform itself, such as when different implementations of randomness require a change in how a value is used or a calculation is done. At other times, the new insights may be into the syntax of a particular language, which may afford more or less elegant ways of expressing the same process. Other insights may point to the permeable boundaries between a program and its platform environment, as when the graphic qualities of a particular character are vital to a particular visual effect. Porting to radically different languages can also challenge deeper paradigmatic assumptions about a program's form and function, including how and why output is produced and whether it (in turn) becomes input of some kind. Taken together, the combined insights of many ports may produce a new, different understanding of the original source. Inhabiting the native ecosystem of its platform, articulated in the mother tongue of its language, ports clarify the original source by showing the many ways it might have been other than what it is. Notably, many of these insights are not available through token-by-token analysis of code. They require closely considered reading, writing, *and* execution of code.

Other ports of `10 PRINT` are discussed in detail later in this book. Three of these, discussed in the remark Variations in Processing, are versions of the program that elaborate on the original and are written in the system Processing. Two others ports are in assembly language, written at the lower level of machine instructions and requiring things to be implemented that are taken for granted in other ports. The first of these, also

discussed in a remark, is for a system without character graphics or, indeed, without typographical characters at all: the Atari VCS. Finally, the last chapter introduces and explicates a Commodore 64 assembly version of `10 PRINT` to show some of the differences between BASIC and assembly programming and to reveal more about the nature of the Commodore 64. These explorations all interrogate the canonical `10 PRINT` program, asking what it means to try to write the same program differently or to try to make a program on another platform the same.

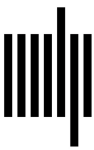
```
(62)    10 PRINT CHR$(205.5+RND(1)); : GOTO 10
```

10 PRINT CHR$(205.5+RND(1)); : GOTO 10

By: Nick Montfort, Patsy Baudoin, John Bell, Ian Bogost, Jeremy Douglass, Mark C. Marino, Michael Mateas, Casey Reas, Mark Sample, Noah Vawter

Citation:

10 PRINT CHR$(205.5+RND(1)); : GOTO 10
By: Nick Montfort, Patsy Baudoin, John Bell, Ian Bogost, Jeremy Douglass, Mark C. Marino, Michael Mateas, Casey Reas, Mark Sample, Noah Vawter
DOI: 10.7551/mitpress/9040.001.0001
ISBN (electronic): 9780262305501
Publisher: The MIT Press
Published: 2014



The MIT Press

This book was designed and typeset by Casey Reas using Avenir by Adrian Frutiger, C64 by Style, and TheSansMono by LucasFonts. Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

10 PRINT CHR$(205.5+RND(1)); : GOTO 10 / Nick Montfort . . . [et al.].
p. cm.—(Software studies)
Includes bibliographical references and index.
ISBN 978-0-262-01846-3 (hardcover : alk. paper)
1. BASIC (Computer program language)—History. I. Montfort, Nick.
QA76.73.B3A14 2013
005.26'2—dc23

2012015872

10 9 8 7 6 5 4 3 2