Universality in Two Dimensions

Nachum Dershowitz
School of Computer Science, Tel Aviv University
Ramat Aviv 69978, Israel
nachum.dershowitz@cs.tau.ac.il

Gilles Dowek
INRIA, 23 avenue d'Italie, CS 81321
75214 Paris CEDEX 13, France
gilles.dowek@inria.fr

לארנון, חברנו ועמיתנו, עוקר וטוחן הרים—לקראת מחצית הדרך הבאה עליך לטובה!

Abstract

Turing, in his immortal 1936 paper, observed that "[human] computing is normally done by writing... symbols on [two-dimensional] paper", but noted that use of a second dimension "is always avoidable" and that "the two-dimensional character of paper is no essential of computation". We propose to promote two-dimensional models of computation and exploit the naturalness of two-dimensional representations of data. In particular, programs for a two-dimensional Turing machine can be recorded most naturally on its own two-dimensional input-output grid in such a transparent fashion that schoolchildren would have no difficulty comprehending their behavior. This two-dimensional rendering allows, furthermore, for a most perspicacious rendering of Turing's universal machine.

1 Introduction

> *Computing is normally done by writing certain symbols on paper.*
> *We may suppose this paper is divided into squares like a child's arithmetic book.*
> *In elementary arithmetic the two-dimensional character of the paper is sometimes used.*
> *But such a use is always avoidable, and I think that it will be agreed that*
> *the two-dimensional character of paper is no essential of computation.*
> *I assume then that the computation is carried out on one-dimensional paper,*
> *i.e. on a tape divided into squares.*
>
> —Alan M. Turing (1936, p. 249)

1.1 Our 2D World

Though we humans live in a three-dimensional world evolving in time, written communication is virtually always two-dimensional, from its hoary beginnings with impressions on clay envelopes [38], through multi-directional hieroglyphics, down to modern e-book readers. Though prose may be read in one dimension, as one long string, the second dimension of the page (or stele) is often quite relevant, not just as a convenience for breaking lines into easily scannable segments. The written or printed page may include diagrams, tables, charts, and illustrations; art and poetry are inherently two-dimensional; arithmetic and geometry are usually performed on the plane; set relations and propositional logic have intuitive two-dimensional renderings (with Euler diagrams, Venn diagrams, and truth tables); mathematical notation (matrices, integrals, etc.), physical notation (e.g. Feynman diagrams), and chemical notation (as far back as John Dalton's molecular diagrams), all make use of both dimensions; even formal proofs are better viewed as trees of formulas than as sequences.

There are, of course, data objects that are inherently multi-dimensional and which require encodings to place on a two-dimensional grid. Movies have a third dimension, time. Architectural plans and organic molecules are also three-dimensional. As we humans traditionally communicate two-dimensionally, we employ more-or-less standard two-dimensional encodings of such multi-dimensional entities: movies as sequences of images;

perspectives and views for rendering plans; and stereographic notations for molecules. The overall importance of the visual dimension for communication and thought has been powerfully argued by Rudolph Arnheim [5].

1.2 2D Computer Science

Today, two-dimensional input-output is de rigueur, with ubiquitous video displays and increasingly prevalent touch screens (used already in 1964 in the innovative Plato environment for interactive education [42]). Many programs—spreadsheets in particular—present data in two-dimensional tabular or graphic form, with which a person interacts. The ease of use of two-dimensional spreadsheets is commonly considered one of the catalysts of the PC revolution [20].

Additionally, there are numerous programming paradigms in which programs are expressed two-dimensionally. In the beginning, there were graphs of finite automata (attributed to Shannon and Weaver [40]) and flowcharts of imperative programs (used by Turing [46]; attributed to Goldstine and von Neumann [19]). Then there were Petri nets [36] and Statecharts [23], followed by UML [16] and various hierarchical, graphical specification paradigms. An old example of a column-aware programming language is IBM's Report Program Generator (RPG) [2]. A number of languages use indentation to indicate structure, notably Python [22], following in the footsteps of ABC and its predecessors (beginning with B0 [18]). Some early two-dimensional efforts were surveyed in [49].

Two-dimensional cellular automata operate in a two-dimensional world and have been studied extensively, starting with Conway's Game of Life (see [17]), and—more recently—with Langton's "ants" [28], dubbed "turmites" (see [15, 44]); see also [31]. These were preceded by studies of finite-state automata on grids [41, 8]. Most recently, cellular automata with evolving topologies have been investigated [6]. The rules for cellular automata are often themselves described in a two-dimensional pattern-based programming language.

There is also a plethora of "visual" programming languages, generic and special-purpose (see [3]), and conferences and journals devoted just to this subject. Still, we see room for further advances. For example, new languages ought to be designed with two-dimensional active interfaces to a constraint-based graphical system—as proposed in [14]. It would seem that the success of two-dimensional visual languages would depend on how well intuition matches the medium and its manipulation.

We describe in the sections that follow what is the simplest fully-powerful (sequential) grid-based language, namely, two-dimensional Turing machines.

1.3 What's to Come

As quoted in the epigraph, Turing [45] already asserted the equipotency of one-dimensional and two-dimensional machines. Hartmanis and Stearns discussed the relative complexity of computing on multi-dimensional and multi-tape Turing machines back in 1965 [24]. And two-dimensional Turing machines remain a common exercise in courses on the theory of computation ([9, Problem 8.2], for example). We will observe the relative naturalness of working with two-dimensional data and the additional advantage of also setting the program down two-dimensionally. One major benefit of two-dimensional languages operating on planar data is that programs and data look similar, so a universal Turing machine can be quite easy to code.

We begin, in the next section, by introducing the language and its two-dimensional computation state. Then to convince the reader of the convenience and pleasantness of this language, we present, in Section 3, various examples of programs written therein. A very straightforward universal machine (self-interpreter) is the subject of Section 4. Lastly, we discuss some of the potential benefits of two-dimensional programming languages, basing ourselves on our experiences with this Turing-machine instance.

2 A 2D Language for 2D Data

In his 1936 paper [45], Turing first introduced the idea of algorithms transforming symbols written on a two-dimensional page divided into squares like a child's arithmetic book. Then Turing explained that it suffices to replace the standard two-dimensional page with a one-dimensional tape.

We stick to Turing's original conception and consider the state of a computation (besides the location in the program) to be a potentially infinite grid of squares, where all but a finite number of squares are blank and where the *cursor* (focus of attention) is always on one particular square. This leads to the extension of the standard

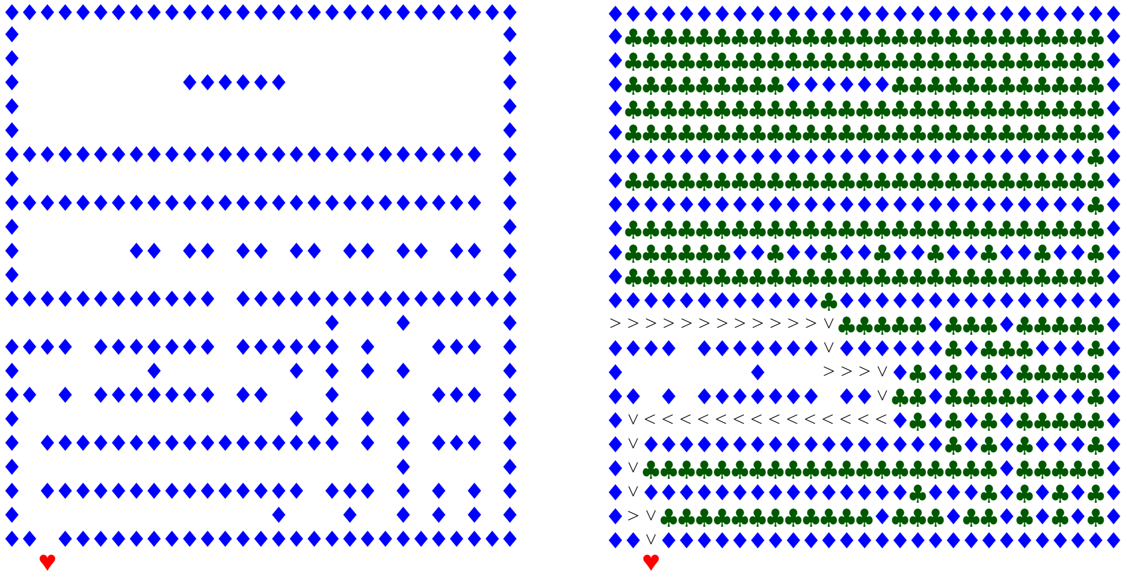


Figure 1: A maze laid out on a grid and its solution. See Example 3.1.3.

language of Turing machines with two additional primitive instructions for moving the cursor on the page one square up or one square down, besides the usual instructions for moving left and right.

As it turns out, and as we illustrate here, two-dimensional Turing machines are remarkably easy to program and reason about. Rather than using quintuples for programs, as in Turing's original work (wherein every step involves reading a symbol, writing on the tape, and moving the head), we use flowcharts à la Uspensky [47] (see [12]), which are akin to Wang's Turing-machine instructions [48], which, in turn, are a linearization of Post's quadruples [33], which separated the motion commands from the reading and writing, any one of which can be performed at any step. To quote one set of course notes [26]: "Written as a list of 5-tuples, the instruction table δ of a TM [Turing machine] M can be hard to understand. We will often find it easier to represent M as a graph or flowchart. The nodes of the flowchart are the states of M."

2.1 2D Data

In general, algorithms are state-transition systems, in which the next-state relation is described in terms of atomic operations on states. The algorithm applies these operations to explore the state of the computation and transform it step-by-step. The data being manipulated by an algorithm are part and parcel of the computation state. The state of a graph-traversal algorithm, for example, would include a representation of the graph in question.

Depending on the model of computation, the state space for the data takes on different forms. For random-access machines [11], the state is conventionally described as a memory map; for Turing machines, the complete state (called an "instantaneous description") is the contents of the machine's tape, the position of its read/write head, and the current "internal state"; most generally—as in abstract state machines [21] (see [13])—states are logical structures. In our case, the state comprises a two-dimensional grid of unbounded size (situated in the lower-right quadrant), and the position of the (single) focus point (head) on that grid.

The representation of the underlying data structures within a computation state can be quite obscure. A one-dimensional linked-list description of an unlabeled graph, referring to arbitrarily ordered vertices, for example, is a far cry from the typical drawing on the office whiteboard. By using a two-dimensional grid, one can, in many natural cases, minimize the need for unintuitive encodings of data. For instance, a labyrinth (and paths within) can be directly drawn as data, as depicted in Figure 1.

2.2 2D Programs

As in ordinary one-dimensional Turing machines, commands for two dimensions come in three flavors: move, draw, and test. Movement of the cursor on the plane is always in single steps in any of the four cardinal directions: ↑ to move one square up (north); ↓ to move down (south); → for right (east); and ← for left (west). Since we will work only in the lower-right quadrant of the plane, attempts to move off that region upwards or leftwards result in no movement at all. (Other conventions, full plane, half plane, or another quadrant, are equally tenable.)

Additionally, one can draw any available symbol in the current cell, the grid square pointed to by the cursor. Instead of considering a finite number of symbols, we will, in our programs, consider a finite number of colors and shapes and paint the current square in any of the available icons (we will be using ♥, ♦, ♣, ♠, and □, predominantly[1]). To draw icon X, we employ the command ✚X. For instance, ✚♥ will draw a (red) heart in the square under the cursor. In some cases (in particular, for the universal machine described later, in Section 4), we also write and test for symbols of the language itself (↑, ✚, etc.)

As an example, the sequence

✚♦→✚♦→✚♦→✚♦↓✚♦↓✚♦↓✚♦←✚♦←✚♦←✚♦↑✚♦↑✚♦↑

(i.e. [draw (blue) diamond; go right]^3; [draw (blue) diamond; go down]^3; [draw (blue]) diamond; go left]^3; [draw (blue) diamond; go up]^3) draws the following picture:

```
♦♦♦♦
♦  ♦
♦  ♦
♦♦♦♦
```

A program can also examine the current cell and continue to one instruction or another according to the outcome of the examination. Thus, a command ●X checks if the current square is painted X, in which case execution proceeds one way. If the square is not X, then execution proceeds a different way. The following program tests the initial cell: if it is a (red) heart, it draws a square made of (red-) hearts instead.

```
●♥→✚♥→✚♥→✚♥↓✚♥↓✚♥↓✚♥←✚♥←✚♥←✚♥↑✚♥↑✚♥↑
✚♦→✚♦→✚♦→✚♦↓✚♦↓✚♦↓✚♦←✚♦←✚♦←✚♦↑✚♦↑✚♦↑
```

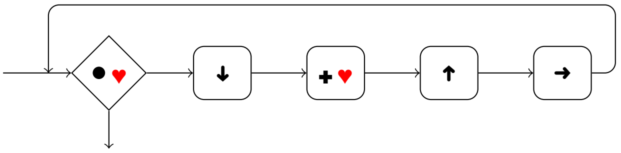
In this layout, when a test (●♥) succeeds, execution continues to the right (with →✚♥, etc.); when it fails, it continues downward (with ✚♦→). Finally, a blank command, as at the end of each line of this drawing program, causes the program to stop in its place.

More generally, a straight-line (loop-free) program bears a tree-like shape, where each node is either labeled with a move instruction, a draw instruction, or a test instruction. Each move or draw node is connected to one child and each test node is connected to two for each possible outcome, forming a "decision tree". In the tabular layout we just used, ordinary instructions are connected to the next command to their right, while tests also connect to the command lying below, whence execution continues in the event the test fails.

Such two-dimensional representations of decision trees are often used in documentation, usually with arrows connecting commands with those that follow. The US Internal Revenue Service (IRS) provides numerous examples; see Figure 2. Using a two-dimensional representation avoids the use of parentheses that are needed when trees are represented in a one-dimensional language, so as to disambiguate and differentiate between

```
if b1 then (if b2 then x else y)
if b1 then (if b2 then x) else y
```

In many cases there are shared components, suggesting an acyclic graph (dag) structure, as in the IRS instructions in Figure 3. More generally, repetitive tasks require a graph structure. Again the IRS has been most obliging; see Figure 4. For these cases, one needs to indicate where in the program to continue in the various cases. One typically uses arrows for that purpose. For example, an algorithm for copying an arbitrary contiguous sequence of (red) hearts from one line to the next can be expressed as the following flowchart, incorporating a loop that terminates after the last (red) heart:



There are various equivalent ways of representing such a flowchart. One is to give a name to each wire and to call them *states*:

[1] The online version of this article colors the icons.

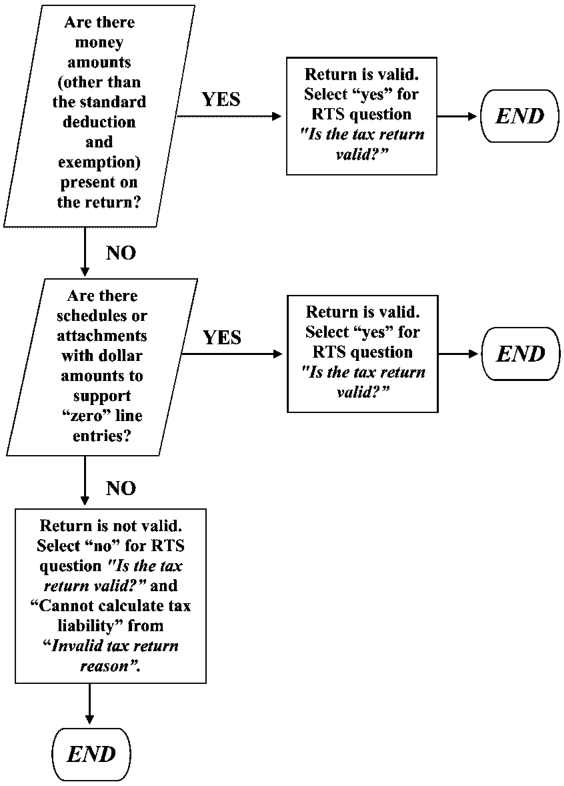

Are there money amounts (other than the standard deduction and exemption) present on the return?
YES
Return is valid. Select "yes" for RTS question "Is the tax return valid?"
END
NO
Are there schedules or attachments with dollar amounts to support "zero" line entries?
YES
Return is valid. Select "yes" for RTS question "Is the tax return valid?"
END
NO
Return is not valid. Select "no" for RTS question "Is the tax return valid?" and "Cannot calculate tax liability" from "Invalid tax return reason".
END

Figure 2: IRS decision-tree flowchart.

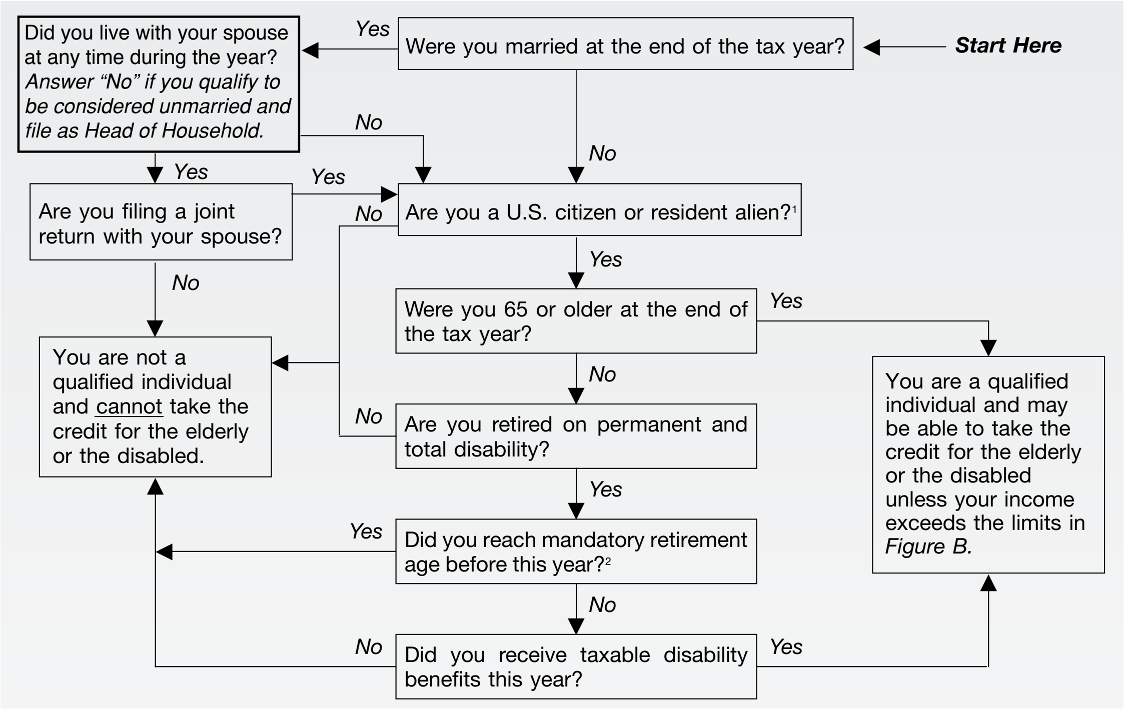

Start Here
Were you married at the end of the tax year?
Yes
Did you live with your spouse at any time during the year? Answer "No" if you qualify to be considered unmarried and file as Head of Household.
No
Yes
Are you filing a joint return with your spouse?
Yes
No
No
Are you a U.S. citizen or resident alien?1
No
Yes
Were you 65 or older at the end of the tax year?
Yes
No
Are you retired on permanent and total disability?
No
Yes
Did you reach mandatory retirement age before this year?2
Yes
No
Did you receive taxable disability benefits this year?
No
Yes
You are not a qualified individual and cannot take the credit for the elderly or the disabled.
You are a qualified individual and may be able to take the credit for the elderly or the disabled unless your income exceeds the limits in Figure B.

Figure 3: IRS acyclic flowchart.

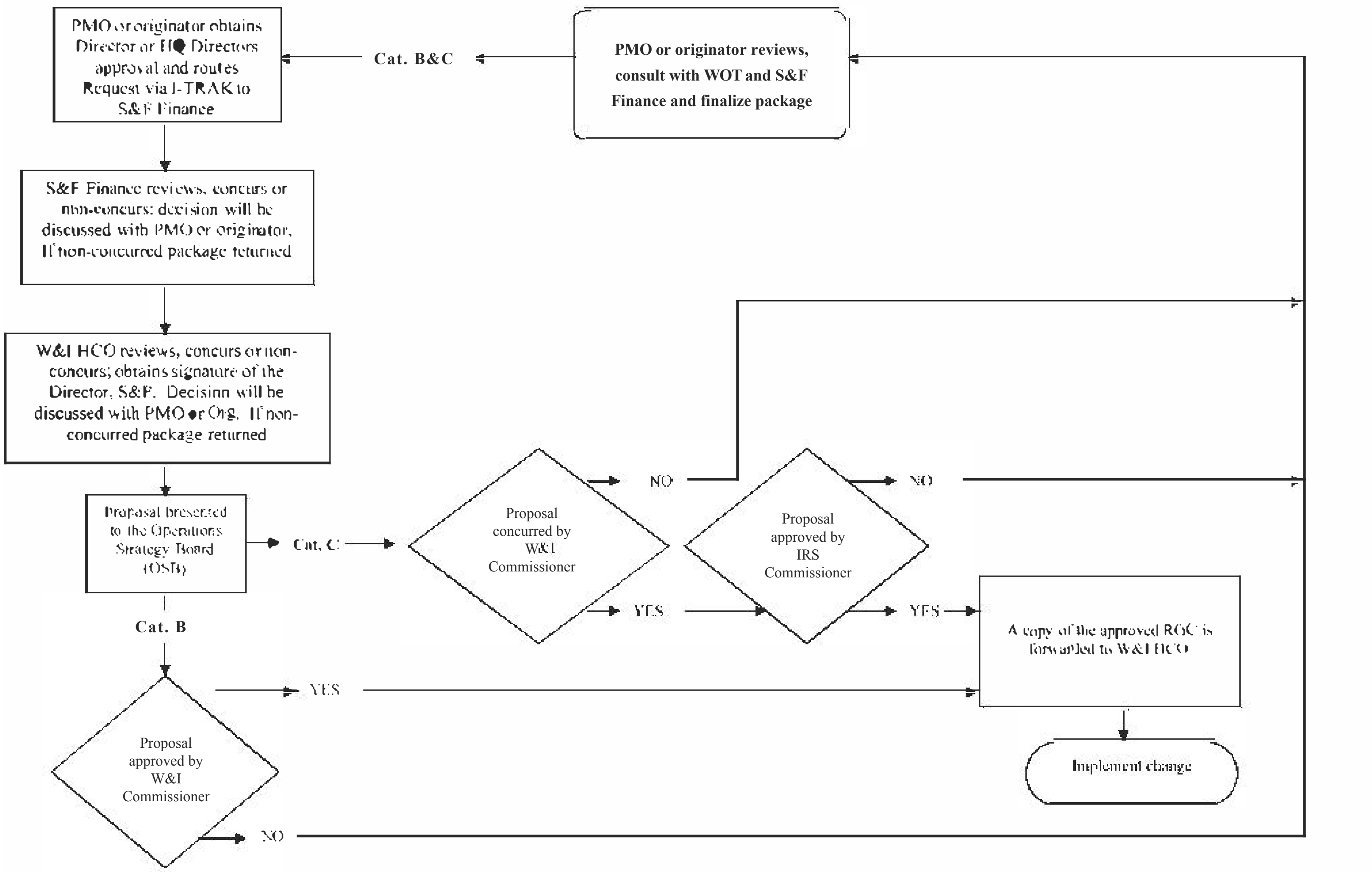
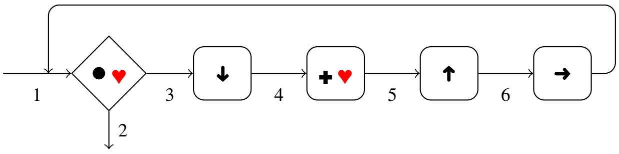

PMO or originator obtains Director or HQ Directors approval and routes Request via J-TRAK to S&F Finance
Cat. B&C
PMO or originator reviews, consult with WOT and S&F Finance and finalize package
S&F Finance reviews, concurs or non-concurs; decision will be discussed with PMO or originator. If non-concurred package returned
W&I HCO reviews, concurs or non-concurs; obtains signature of the Director, S&F. Decision will be discussed with PMO or Org. If non-concurred package returned
Proposal presented to the Operations Strategy Board (OSB)
Cat. C
Proposal concurred by W&I Commissioner
NO
YES
Proposal approved by IRS Commissioner
NO
YES
A copy of the approved ROC is forwarded to W&I HCO
Implement change
Cat. B
Proposal approved by W&I Commissioner
YES
NO

Figure 4: IRS graph flowchart.



Then, each node is a transition between states, leading to the transition table:

In state	do	and go to state
1	look under cursor:	3 if it is;
	is it a (red) heart?	2 if it is not
2	(halt)	
3	move down	4
4	draw (red) heart	5
5	move up	6
6	move right	1

Another option is to write the program down in a language with gotos (jumps),

```
1: **if** (red) heart **then go to** 3
2: **halt**
3: **move** down
4: **draw** (red) heart
5: **move** up
6: **move** right
7: **go to** 1
```

the only difference being that, if not stated otherwise, the next command to be executed is the one below, on the next line.

In the next section, we will lay programs out on a grid. That will require various connectors, in lieu of arrows.

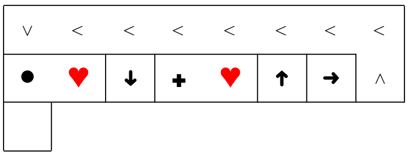
2.3 2D Programs as 2D Data

Considering that the state of computation is a two-dimensional grid, flowcharts can just as well be represented on the same sort of grid layout as for the data (though not on the input/output grid itself). In Section 4, we will see how this convention allows any program to be given as input to a universal machine without any need to encode it, in which case the *input* program and *its* input are both placed on the universal program's data grid.

This tabular format for programs requires some sort of notation for arrows to connect nonadjacent commands. The following is our copying example in this format:

∨	<	<	<	<	<	<	<
●	♥	↓	✚	♥	↑	→	∧

Arrows are realized by a series of links, ∧, ∨, <, and >. The test and draw commands are written across two squares, one for the type of instruction (● or ✚) and the second for the symbol that is to be read or written (♥ in this case). In the absence of a link, the instruction to the right is performed next. For a conditional, if the test fails, the command below is taken. In this example, that is a blank, halt instruction. Think of the above layout as five commands (not counting halt) and one non-adjacent back-arrow, as shown here:



In the examples that follow, we will usually omit cell boundaries, turning the above program into this:

```
∨ < < < < < < <
● ♥ ↓ ✚ ♥ ↑ → ∧
```

Since the grid is only two-dimensional, we need to make provision for crossing over other links when required. It suffices to use just ≪ and ≫ indicating a double step to the left or right, respectively. For example, the following program (for exclusive-or of two binary numbers) shares code segments and uses a jump-right link over an up-arrow link (≫ ∧ ↓):

```
∨ < < < < < < < < < < < <
● ♥ ↓ ● ♥ > ↓ ✚ ♦ ↑ ↑ → ∧
∨   ∨ > ≫ ∧ ↓ ✚ ♥ ∧
● ♦ ↓ ● ♦ ∧ ∧
      > > > ∧
```

2.4 Completeness

It should be clear that our language is Turing-complete and can describe anything that is computable in one or two dimensions. If an ordinary one-dimensional Turing machine, when in state q, for example,

- just moves left, not writing anything or changing state—when it sees a 1, but
- writes a 1, moves right, and then enters state r—when it sees a 0,

then in our language there would be a program segment for q looking something like this:

	∨	<	<	<	<	<	
q ▷	●	1	✚	1	←	∧	
	●	0	✚	1	→	>	▷r

with entry point q and exit to r as indicated. The first line corresponds to the quintuple $\langle q, 1, q, 1, L\rangle$; the second, to $\langle q, 0, r, 1, R\rangle$.

The extra dimension does not, of course, increase the computational ability of Turing machines: the same numeric and string functions are computable regardless of the number of dimensions.

3 Examples

A few examples should demonstrate the ease with which many a program can be written.

3.1 Motion Examples

We begin with three simple examples that use the planar layout to advantage.

3.1.1 Example: Bounce, Bounce

Our first example is an endless display of a ball bouncing off the walls of a room and knocking out black spades, the "bricks", as in the classic Breakout computer game (a.k.a. Little Brick Out) [50]. Sample input data is shown in Figure 5. The ball can move in any of four directions: up left (↖), up right (↗), down left (↙), or down right (↘). When it hits a (green) club (clover/trefoil), it changes its vertical direction; when it hits a (red) heart, it changes horizontal direction; when it hits a cross (as in the corners), it changes both directions, and turns on its heels; when it hits a black spade, it destroys it and changes its vertical direction. The program consists of four parts (each dealing with one direction of motion) with "wires" connecting them:

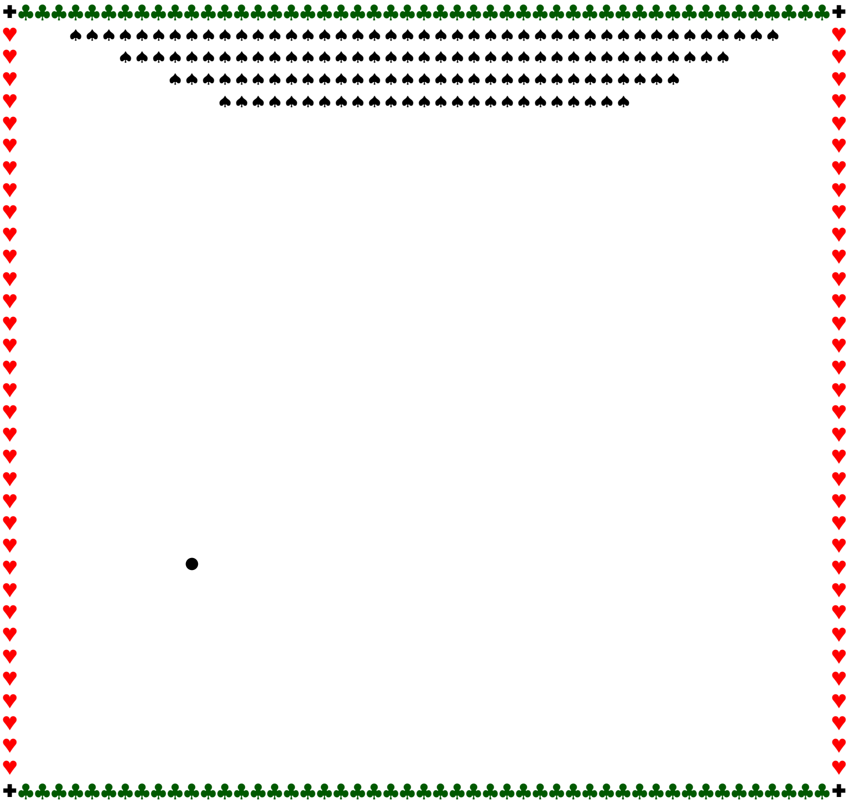
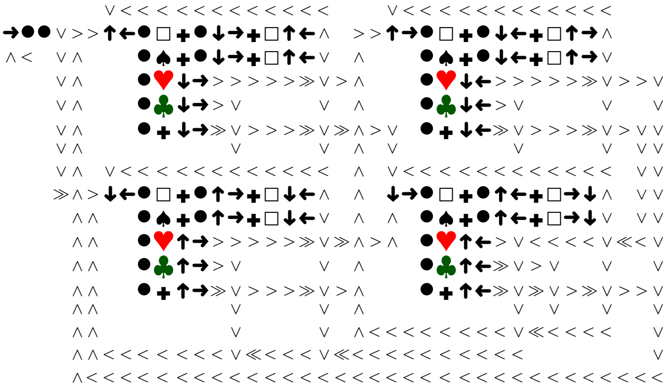


Figure 5: Layout for Breakout ball game.



For clarity, white squares are drawn when they indicate testing for white or painting white (erasing).

All our example programs start from the first command on the second line. In the above example, that is a loop, going right until the first circle is encountered.

3.1.2 Example: Follow the Road

In our next example, the cursor simply follows the path on a map placed on the data grid. The path is indicated by means of the four one-step links, ∧, ∨, <, and >, and the two double-step links, ≪, ≫. Here is the program (note the six tests, one above the other, for the six link symbols) and some sample input (starting at the lower left and cycling about):

```
∨<<<
●>→∧                    ∨<<<<
●<←∧                  ∨<∨≪<<∧
●∧↑∧                  >≫∨>≫∧∧
●∨↓∧<       >>>>>>>>>>>≫∨>>∧
●≫→→∧       ∧<<<<<<<<<<<<
●≪←←∧
```

Were the path to end in a blank (white) square, execution would halt at that point, with all the tests for links failing.

3.1.3 Example: Maze

The next program searches for the way out of a maze, indicated by a (red) heart:

```
∨<<<<<<
✚>→●♥ ∧
   ●□>∧<<<<
   ←✚<←●♥ ∧
       ●□>∧<<<<
       →✚∧↑●♥ ∧
           ●□>∧<<<<
           ↓✚∨↓●♥ ∧
               ●□>∧<<<
               ↑✚♣←●>∧<<
                   →→●<∧<<
                     ←↓●∧∧<<
                       ↑↑●∨∧
```

This realization of Ariadne’s thread (Hansel and Gretel’s “bread-crumb” method) uses the four one-step links to trace the path connecting the entrance with the current location. They replace both the stack and the set of “grey” (currently visited) cells in the standard depth-first search algorithm. Already-visited cells are painted with (green) clubs; unvisited cells are left blank. See Figure 1.

3.2 Arithmetic Examples

We write unary numbers as one (blue) diamond followed by as many (red) hearts as the value of the number (zero or more). For example, 9 and 3 are recorded as follows:

```
♦♥♥♥♥♥♥♥♥♥
♦♥♥♥
```

3.2.1 Example: Addition

The following program for adding two numbers takes two such inputs (as above) and erases the hearts on the second line one by one, each time adding a heart to the end of the first line:

```
∨<<<<<<<<<<<<<<<<<<<<<
↓→●□←●♥✚□←●♦↑→●□✚♥←●♦∧
 ∧<      ∧<  ∧<   ∧<
```

The result (i.e. 12) looks like this:

```
♦♥♥♥♥♥♥♥♥♥♥♥♥
♦
```

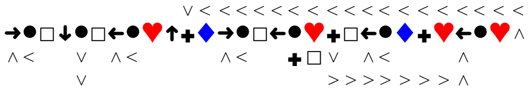
Note the two occurrences of a programming cliché for “go left until (blue) diamond”:

```
←●♦
∧<
```

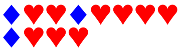
and similar ones for “right until white”.

3.2.2 Example: Subtraction

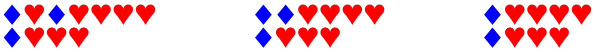
Subtraction could be performed in a similar way. But, instead, we use the following method, which preserves the subtrahend:



It first paints a (blue) diamond on the upper line just above the last element of the lower line:



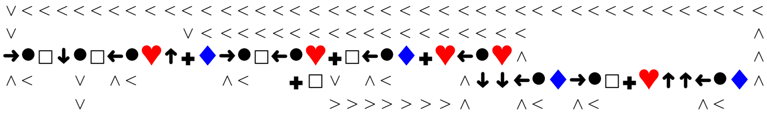
Then it moves the (blue) diamond left, erasing a (red) heart at the end of the first line, step by step,



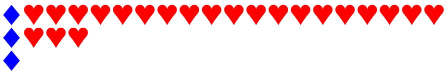
until it is on the first column, at which time the upper line contains the difference. This method is facilitated by the proximity of the two rows.

3.2.3 Example: Division

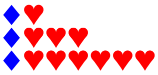
The quotient of two numbers may be found by repeated subtraction:



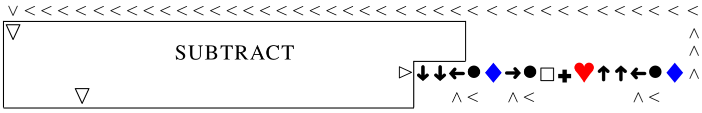
With input



the output gives the quotient ($19 \div 3 = 6$) and remainder (1) on the third and first lines, respectively:



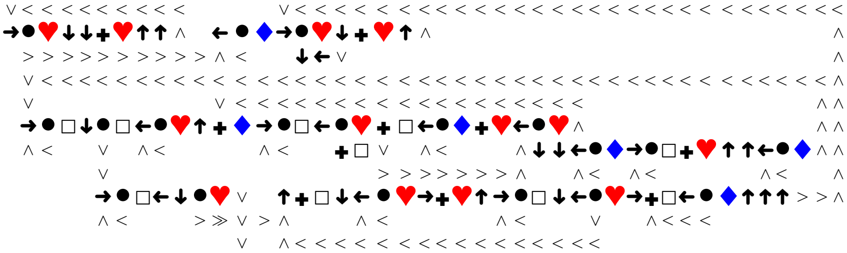
This program could better be expressed modularly, as follows:



where the module SUBTRACT is that defined above (Example 3.2.2). The module entrance and its two exits—the lower exit for when the difference would be negative—are marked by triangles.

3.2.4 Example: Square Root

Our next example is the ancient Babylonian (or Heron's) method for calculating (the floor of) the square root of a positive integer z, starting with the (rough) over-estimate z and repeatedly replacing an estimate y with the (rounded down) average of $\lfloor z/y \rfloor$ and y, until the average is greater than the current estimate:



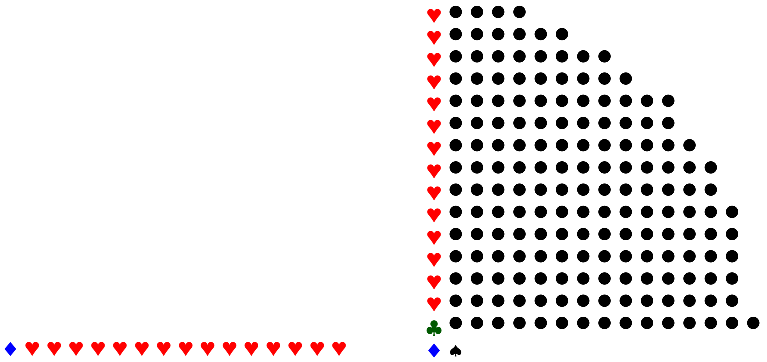
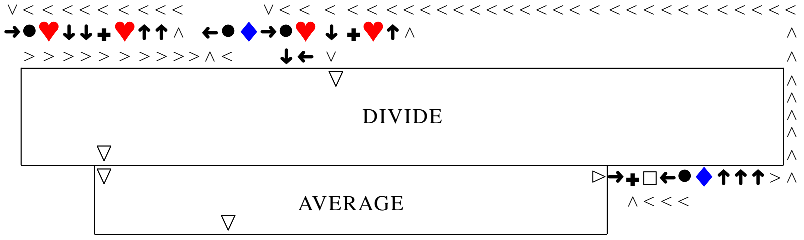


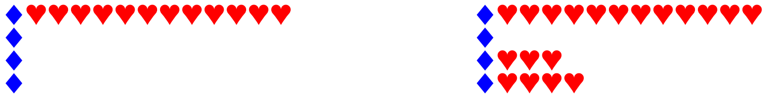
Figure 6: Input and output of Example 3.2.5.

Shown modularly, this is simply



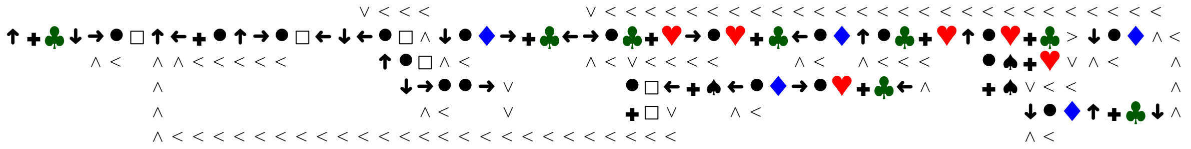
where division is as above (Example 3.2.3), and AVERAGE is as in the full program shown above. The rest consists of various copying and erasing operations. The averaging module has two exits: if the first number is not larger than the second, the bottom exit is taken and the computation of the square root comes to a halt.

With this program, the input on the left (12) results in the outcome (3) on the right, on the third line of the output grid:



3.2.5 Example: Quarter Pie

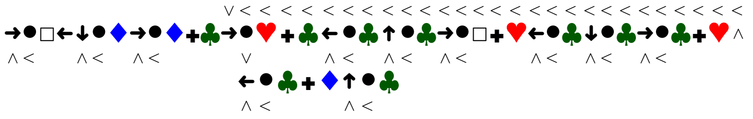
Given a unary number, as before, the following little program draws a (pixelated) quarter of a circle, as depicted in Figure 6 for input 15:



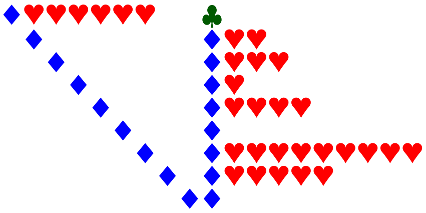
We leave it to the intrepid reader to decipher.

3.2.6 Example: Addressing

Given a number n in unary, the following program accesses the number stored on the n-th line and copies it to a "register" (line 0).



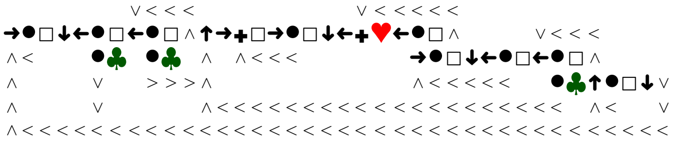
For this to work as advertised, the data has to be organized with a diagonal addressing scheme, as follows:



The number 9 is copied from the 6th location in the array to the line with the (green) club.

3.2.7 Example: Sorting

This is a version of bubble sort, rearranging a list of unary numbers in non-descending order, with a (green) club marking the end of the list:

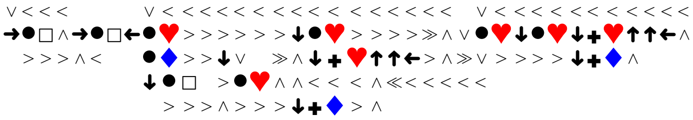


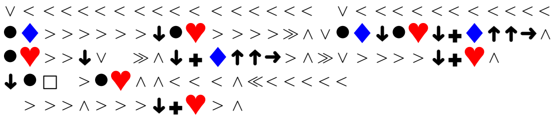
3.3 Binary Arithmetic Examples

Next, we present a few examples of binary arithmetic (we already saw exclusive-or). To indicate a digit 0, (blue) diamonds are used; (red) hearts signify 1; blank squares demarcate numbers.

3.3.1 Example: Binary Addition and Subtraction

Binary addition and subtraction require consideration of the carry, but are not difficult:

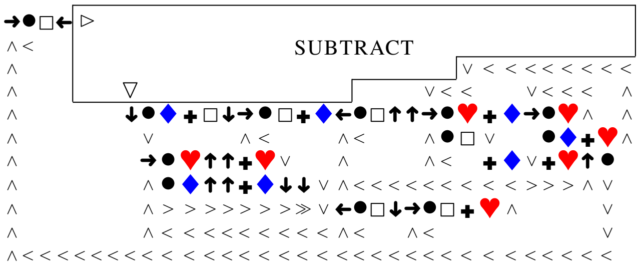




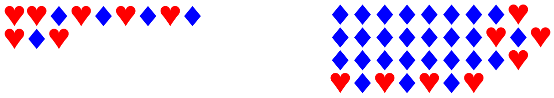
The upper program, for addition, skips over white space to the left of the first addend, while the lower one, for subtraction, presumes that the first number abuts the left margin.

3.3.2 Example: Binary Division

The program

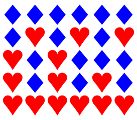


is “hardware” division, and uses the above binary subtraction module to determine each digit of the quotient after successive right shifts of the divisor. Running it on the input shown on the left (426 in binary), results in the output on the right (85 on the last line, with a remainder of 1 on the first):

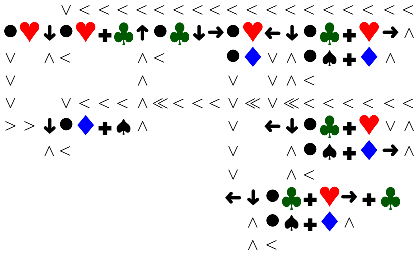


3.3.3 Example: Binary Addressing

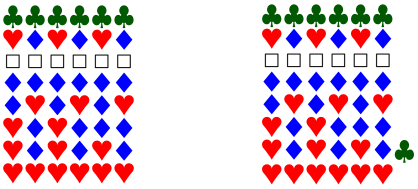
To look up a binary number (e.g. 42) in an ordered list of binary numbers like



(containing: 0, 21, 40, 42, 63), we can search digit by digit:



Here, we use a (green) club to “highlight” the current (red) heart and a (black) spade to highlight a (blue) diamond. Running this program on the input on the left, looking for 42, yields the row marked with a (green) club on the right:



The input is delimited by a row of (green) clubs on top and a row of (red) hearts below, with a row of white squares separating the number (42) being sought from the list of numbers being searched. (The program is not designed to work for numbers not in the list.) This program can be combined with instructions to copy the remainder of the marked row, as we did in the unary case (Example 3.2.6).

4 A Universal Machine

Turing [45] also invented the notion of *universal machine*, a program that can interpret *any* program and run it on any given input. In Turing’s case, this is all done on a one-dimensional tape: the transition function of the machine to be simulated is written on the tape in a standard form as a sequence of quintuples; this is followed by the desired input to that program, with some marker separating them. In models of computation not having strings, such as the recursive functions, programs are encoded in some abstruse fashion. In the usual programming languages, which have facilities for inputing strings, the program to be interpreted can be given in its natural textual form. Of course, modern languages come chock full of programming features, making the writing of a universal program (for C, for example) quite complicated (hundreds of thousands of lines [1]), unless the language has a built-in self-interpreter (as, for example, in Scheme), in which case, one simply calls the interpreter, but need not reprogram the semantics of instructions.

A two-dimensional Turing machine enjoys the best of all worlds. Programs and data require virtually no encoding, and the interpreter is half-a-page long and of transparent simplicity. See Figure 7. To keep track of the program counter (current position in the program) and the cursor (current position in the data), the input to the interpreter, both program and data, have extra lines inserted between their lines, so that the interpreter can mark those two positions in those channels, without disturbing the input itself. The layout is shown in Figure 8. (One could arrange for the input to be given in pristine form and for the interpreter to insert blank lines before continuing as above.)

For example, if the current command in the program being interpreted is draw-diamond (line 18 in the universal machine: ●✚→● ♦), then the interpreter executes the following instructions:

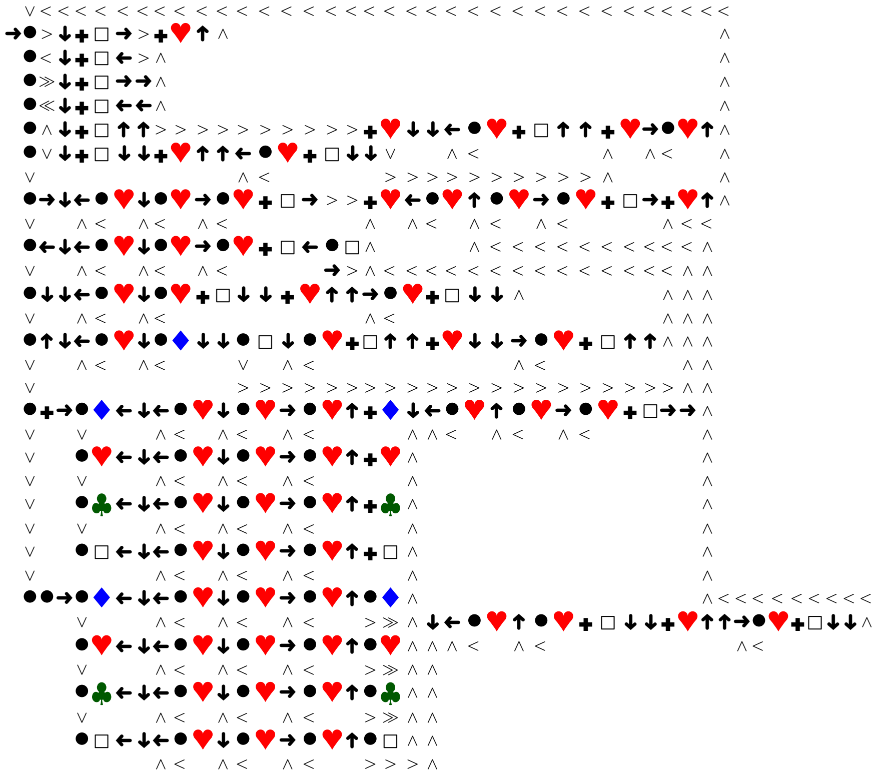
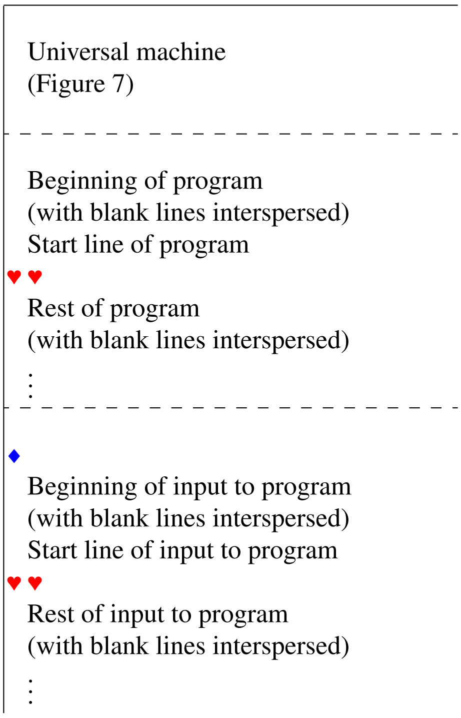
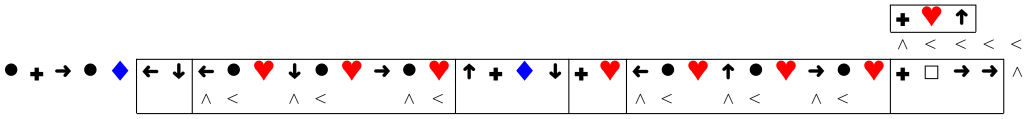


Figure 7: A two-dimensional universal machine.



Universal machine
(Figure 7)
Beginning of program
(with blank lines interspersed)
Start line of program
♥ ♥
Rest of program
(with blank lines interspersed)
⋮
♦
Beginning of input to program
(with blank lines interspersed)
Start line of input to program
♥ ♥
Rest of input to program
(with blank lines interspersed)
⋮

Figure 8: Initial layout for universal machine.



It first goes left and down (← ↓) back to the access channel. Then it moves to the input-data area by going left-until-heart to a marker that has been left in the left margin, followed by down-until-heart to get to the current line in data, and finally right-until-heart to get to the marked position of the focus within the data. Once in position, the draw-diamond command can be executed (↑ ✚ ♦ ↓) by going up from the channel to the data, drawing, and then back down again. This is followed by a retracing of steps back to the program (left until margin, up to program, right to program pointer). The last thing to do is to move the program-pointer two cells to the right (erasing the heart and redrawing it two cells over to the right) and reposition the universal machine's cursor on the next instruction above the new (red) heart (✚ □ → → ✚ ♥ ↑).

This interpreter handles only four icons—(blue) diamond, (red) heart, (green) club, and (white) square, but it is a trivial matter to add two lines for each additional drawing instruction or test. A full set of input/output symbols would include all desired icons, plus the symbols for the six commands (←, →, ↑, ↓, ✚, ●), plus the six symbols for links (<, >, ∧, ∨, ≪, ≫). This would enable the universal machine to interpret itself.

It is the marriage of two-dimensional programming with two-dimensional data that allows for this very simple, transparent approach to universality. While a Turing-machine interpreter executes the universal program (our prototype is written in OCaml [29]), one can watch exactly how each cell changes. In this way, computation proceeds visibly, on the precise level of Turing's analysis of human clerical computation.[2]

5 Discussion

Ever since the invention of painting and writing in the hoary past, humankind has been using two dimensions for interpersonal communication and for the recording of data. The invention of codices in antiquity added a convenient third dimension as well, in the form of a sequence of two-dimensional pages. But the mechanical age reverted some forms of communication to just one dimension, notably the telegraph and ticker tape. Unfortunately, to a large degree, computer programming inherited this distorted one-dimensional view. All the same, computer scientists certainly continue—informally, at least—to use two-dimensional representations of data on paper and whiteboards, as well as two-dimensional depictions of programs, most conspicuously for finite automata. And they commonly use various types of diagrams and charts for the specification of requirements.

Nowadays, the second dimension is becoming ever more important. Today's touch-screen interfaces, with their finger gestures, are inherently two dimensional, and often easier than communication via keyboard. We contend that the time is ripe for programming itself to make greater and better use of the second dimension and of modern interfaces. Such use can take many different forms. One can simply use flowcharts and diagrams to program Turing machines directly—as described in this paper, and similarly for some higher-level languages. The modular structure of programs or systems could be given a graphic formulation—using, say, Venn-like diagrams, where each function is a point and each module is represented as a set, allowing one to drag and drop functions and modules. A visual language of pipes and fittings could be used for Unix-like shell scripts.

Turing machines were conceived so as to employ only the most primitive basic operations, with the intent of analyzing the fundamental nature of computation. Though standard one-dimensional machines suit this purpose, two-dimensional operations are no less fundamental. Unlike one-dimensional Turing machines, two-dimensional machines are quite practical for small-scale tasks. We have seen in the preceding sections how easy it is to code basic algorithms for unary or binary arithmetic, for sorting, for graph traversal, and even for a universal machine. Having two-dimensional data makes algorithms for arithmetic quite natural, much more so than running back and forth on a narrow tape. It goes without saying that algorithms for mazes or planar graphs benefit from the second dimension. Indeed, we have found programming in this way both relatively hassle-free and error-free, and quite enjoyable to boot.

Having two dimensions for programs, as advocated here, makes their control structures more transparent than with a linear language and improves readability. Despite the bad name unstructured flowcharts have earned [30] (see [39], for example, for a two-dimensional flowchart-based environment designed for teaching novices the basics of programming), they do not have the same disadvantages in a variable-free language operating on a single global state—the grid. We have seen the use of modules in some of our examples. Packaging clichés and modules

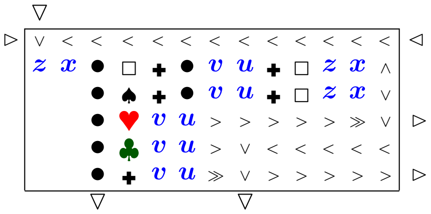
[2] A video of this universal machine in action, interpreting a sorting program, may be viewed at `http://nachum.org/Universal.html`.

in small named units would make for more perspicuous programs. Modular Turing-machine flow diagrams were utilized extensively by Hermes [25].

We have found two-dimensional Turing machines to be an ideal springboard for an investigation of the coupling of two-dimensional programs and two-dimensional data. True, programming-language layout and data-domain layout are orthogonal issues; one could be two-dimensional, while the other need not be. But—for the universal machine—the combination is particularly apt, as we saw in the previous section.

Various extensions of the two-dimensional paradigm come to mind. To add a third dimension to the data, one would only need to add two motions ("in" and "out"); it is similarly a trivial matter to adopt other topologies. Adding a third dimension to programs, too, would obviate the need to accommodate crossing wires (with ≪ and ≫). It is also a trivial matter to add nondeterminism (as suggested by Turing [45] for implementing proof search) to our programming language. A command "?" with two exits would do the trick. The desirability and difficulty of including nondeterminism in a secondary-school computer-science curriculum are discussed in [4].

Compared to higher-level programming languages, it is true that not having even finite-domain variables often leads to duplication in programs. The Breakout bouncing-ball program (Example 3.1.1) lacks a variable for the velocity vector. Macros—that is, modules with cell variables—could help reduce the resultant duplication in programming. For the bouncing ball, one could employ a macro with the following shape:



	▽													
▷	∨	<	<	<	<	<	<	<	<	<	<	<	<	◁
	z	x	●	□	✚	●	v	u	✚	□	z	x	∧	
			●	♠	✚	●	v	u	✚	□	z	x	∨	
			●	♥	v	u	>	>	>	>	>	≫	∨	▷
			●	♣	v	u	>	∨	<	<	<	<	<	
			●	✚	v	u	≫	∨	>	>	>	>	>	▷
			▽					▽						

The parameter x takes as its value either ← or → and z takes ↑ or ↓ to indicate the direction of motion; parameters u and v should be their opposites. These commands are substituted in the macro for the parameters to yield a specific module. Entry points and exit points of the four modules need to be connected as before. The abstraction made possible by such macros would go a long way to making the Turing paradigm more practical.

An even better way to overcome the duplication problem would be to have more than one read/write head, each with its own cursor, a possibility also entertained by Turing. With a second cursor, one could use one grid square to code the direction of the velocity vector in the bouncing-ball example, obviating the need to repeat everything fourfold. Multiple heads provide a modicum of local storage for controlling behavior, while, with only one cursor, that drop of data needs to be shuttled around to keep it near the head for easy access. Also, with many heads, it is much easier to perform actions that span unbounded distances—like copying a line an arbitrary (computed or inputted) distance from the original. Whereas local actions—copying a fixed distance—are quite easy with only one head, long-distance operations require much shuffling about with only one head. Incorporating multiple heads in our framework is easy: just add a command "!X", say, to shift focus to the head colored X. A universal machine, given one more head than the program it interprets, would have an easy time, easily keeping everything it needs to know at its "fingertips"; if, on the other hand, the universal machine had only as many heads as the program, it would need to add tracks in the program and data, as we did for the single-headed machine.

In the long run, these consideration might suggest the implementation of a random-access machine (RAM) model on top of a Turing machine model, as hinted by the addressing programs (Examples 3.2.6 and 3.3.3). RAM machines are, of course, more powerful (time-wise) than Turing machines. Turing machine-steps, on the other hand, were invented to mirror atomic human operations. Their absolute and relative simplicity is why we launched our voyage with two-dimensional Turing machines.

Last, but not least, our two-dimensional language is ideal for introducing youngsters and novices to the exciting world of algorithms. The pedagogic advantages of two-dimensional languages have long been recognized, beginning with Seymour Papert's Logo turtles [32][3] and more recently with Kara [34, 35][4] and Scratch [37], for instance. The value of visually interpreted code for beginning programmers is considered in [27]. For experimental evidence of the advantage of direct manipulation of objects made possible by two dimensions, see [7]. Similarly, our two-dimensional language provides a "hands on" learning experience. In fact, the Bloomfield Science Museum in Jerusalem has included such two-dimensional Turing machines as part of its innovative 2013–2015 computer-science exhibition [51]. A prototype JavaScript implementation [10] is available online.[5]

[3] See `https://logothings.wikispaces.com`.

[4] See `http://www.swisseduc.ch/compscience/karatojava`.

[5] See `http://inriamecsci.github.io/#!/grains/Turing-2D`.

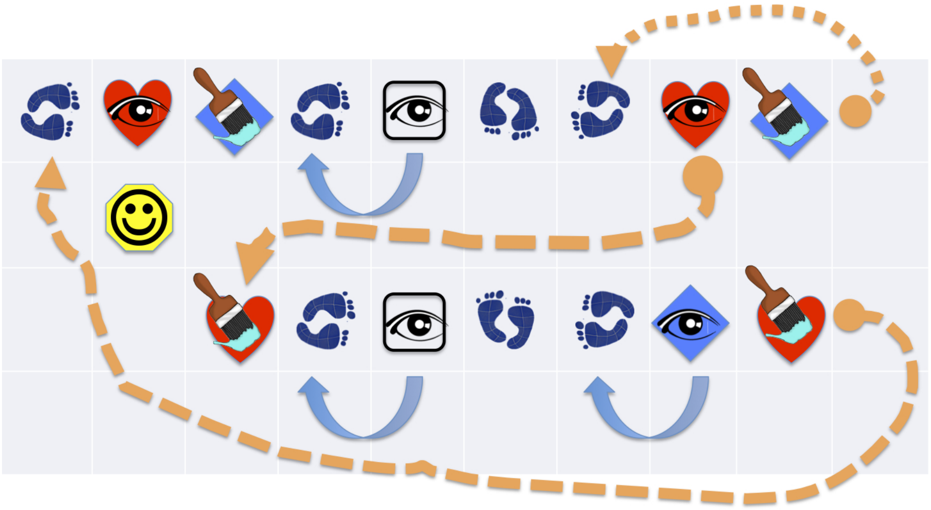


Figure 9: A whimsical program to convert from unary to binary.

The two-dimensional Turing machine forges a wonderful framework within which one can explicate the workings of a RAM machine. Recently, a two-dimensional Turing-machine interpreter of the Intel 8086 assembly language has been constructed. The assembly-language program, (random-access) memory, and registers are all laid out on the plane; the Turing machine decomposes and interprets each instruction, accesses the indicated memory locations, copies (binary) numbers from registers to memory and vice-versa, and performs arithmetic operations as instructed, all in a most transparent fashion. See [43].[6]

As our 2D programming language contains no alphabetical symbols or numerals (only ✚, ●, arrows, and [colored] shapes), there is actually no need for one to know the alphabet to become proficient in reading and writing programs. A whimsical rendition (inspired by Snakes and Ladders) of a program to convert a unary number into binary is portrayed in Figure 9.

Acknowledgements

We thank Idan Dershowitz, Erwin Engeler, Eran London, Jürg Nievergelt, Mariano Schain, and Amiram Yehudai for reading drafts of this work and for their comments.

References

[1] European Organization for Nuclear Research (CERN), 2013, “What is CINT?”, Geneva, Switzerland. Available at http://root.cern.ch/drupal/content/cint (viewed Apr. 12, 2013).

[2] International Business Machines (IBM), June 1994, *RPG/400 User's Guide*, IBM, Armonk, New York. Available at http://publib.boulder.ibm.com/infocenter/iseries/v5r3/topic/books/c0918160.pdf (viewed Apr. 12, 2013).

[3] Wikipedia, 2013, “Visual programming language”, http://en.wikipedia.org/wiki/Visual_programming_language (viewed July 4, 2013).

[4] Michal Armoni and Judith Gal-Ezer, 2006, “Introducing non-determinism”, *Journal of Computers in Mathematics and Science Teaching* 25(4), pp. 325–359.

[5] Rudolf Arnheim, 1969, *Visual Thinking*, University of California Press, Berkeley, CA.

[6] See http://www.2dturingmachine.com.

[6] Pablo Arrighi and Gilles Dowek, July 2012, “Causal graph dynamics”, *Proceedings of the 39th International Colloquium on Automata, Languages, and Programming (ICALP 2012)*, Warwick, UK, Lecture Notes in Computer Science, vol. 7392, Part II, doi:10.1007/978-3-642-31585-5_9, pp. 54–66. Available at `http://arxiv.org/pdf/1202.1098v3` (viewed Apr. 12, 2013).

[7] Alan F. Blackwell, 1997, “Correction: A picture is worth 84.1 words”, *Proceedings of the First ESP Student Workshop*, Washington, DC, pp. 15–22. Available at `http://www.cl.cam.ac.uk/~afb21/publications/Student-ESP.html` (viewed Apr. 12, 2013).

[8] Manuel Blum and Carl Hewitt, Oct. 1967, “Automata on a two-dimensional tape”, *IEEE Conference Record of the 8th Annual Symposium on Switching and Automata Theory (SWAT)*, Austin, TX, pp. 155–160.

[9] George S. Boolos, John P. Burgess, and Richard C. Jeffrey, March 2002, *Computability and Logic*, Cambridge University Press, Cambridge, UK.

[10] Arthur Brongniart, Gilles Dowek, and Nachum Dershowitz, 2013, “Simulation d’une machine de Turing 2D”, online at `http://inriamecsci.github.io/#!/grains/Turing-2D` (viewed May 2, 2012).

[11] Stephen A. Cook and Robert A. Reckhow, 1973, “Time-bounded random access machines”, *Journal of Computer Systems Science*, 7: 354–375, doi:10.1145/800152.804898. Available at `http://www.cs.utoronto.ca/~sacook/homepage/rams.pdf` (viewed Apr. 12, 2013).

[12] Liesbeth De Mol, Sep. 2006, “Post’s machine”, unpublished report, Center for Logic and Philosophy of Science, Universiteit Gent, Belgium. Available at `http://logica.ugent.be/liesbeth/postsmachine.pdf` (viewed Apr. 12, 2013).

[13] Nachum Dershowitz, July 2011, The Generic Model of Computation, *Proceedings of the Seventh International Workshop on Developments in Computational Models (DCM 2011)*, E. Kashefi, J. Krivine, and F. van Raamsdonk, eds., Zurich, Switzerland, *Electronic Proceedings Theoretical Computer Science (EPTCS)*, vol. 88, pp. 59–71. Available at `http://nachum.org/papers/Generic.pdf` (viewed Apr. 16, 2013).

[14] Nachum Dershowitz and Claude Kirchner, 2008, “SPREADSPACES: Mathematically-intelligent graphical spreadsheets”, in P. Degano, R. Nicola, and J. Meseguer, eds., *Concurrency, Graphs and Models: Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*, Lecture Notes in Computer Science, vol. 5065, Springer-Verlag, Berlin, pp. 194–208. Available at `http://nachum.org/papers/SpreadSpaces.pdf` (viewed Apr. 12, 2013).

[15] Alexander K. Dewdney, Sept. 1989, “Two-dimensional Turing machines and tur-mites make tracks on a plan”, Computer Recreations, *Scientific American* 261, pp. 180–183.

[16] Martin Fowler, 2003, *UML Distilled: A Brief Guide to the Standard Object Modeling Language* (3rd ed.), Addison-Wesley, Reading, MA.

[17] Martin Gardner, 1970, “The fantastic combinations of John Conway’s new solitaire game ‘life’”, Mathematical Games, *Scientific American* 223, pp. 120–123.

[18] Leo Geurts and Lambert Meertens, 1975, “Designing a beginners’ programming language”, in S. A. Schuman, ed., *New Directions in Algorithmic Languages* , pp. 1–18, IRIA, Rocquencourt, France. Available at `http://www.kestrel.edu/home/people/meertens/publications/papers/Designing_a_beginners_programming_language.pdf` (viewed Apr. 19, 2013).

[19] Herman H. Goldstine and John von Neumann, Apr. 1947, “Planning and coding of problems for an electronic computing instrument: Report on the mathematical and logical aspects of an electronic computing instrument, part II, volume 1–3”, Institute for Advanced Study, Princeton, NJ. In A. Taub, ed., Collected Works of J. von Neumann, New York, Pergamon, vol. 5, 1965, pp. 80–151. Available at `http://library.ias.edu/files/pdfs/ecp/planningcodingof0103inst.pdf` (viewed Apr. 15 2013).

[20] Burton Grad, July–Sept. 2007, “The creation and the demise of VisiCalc”, *IEEE Annals of the History of Computing* 29(3), pp. 20–31.

[21] Yuri Gurevich, 1995, “Evolving algebras 1993: Lipari guide”, in Egon Börger, ed., *Specification and Validation Methods*, Oxford University Press, Oxford, pp. 9–36. Available at `http://research.microsoft.com/~gurevich/opera/103.pdf` (viewed Apr. 12, 2013).

[22] Naomi Hamilton, Aug. 2008, “The A–Z of programming languages: Python”, *Computerworld.* Available at `http://www.computerworld.com.au/article/255835/a-z_programming_languages_python` (viewed Apr. 20, 2013).

[23] David Harel, June 2007, “Statecharts in the making: A personal account”, *Proceedings of the 3rd ACM SIGPLAN History of Programming Languages Conference (HOPL III)*. Available at `http://www.wisdom.weizmann.ac.il/~harel/papers/Statecharts.History.pdf` (viewed Apr. 12, 2013).

[24] Juris Hartmanis and Richard E. Stearns, May 1965, “On the computational complexity of algorithms”, *Transactions of the American Mathematical Society* 117, pp. 285–306.

[25] Hans Hermes, 1961, *Aufzählbarkeit, Entscheidbarkeit, Berechenbarkeit: Einführung in die Theorie der rekursiven Funktionen*, Grundlehren der mathematischen Wissenschaften 109, Springer, Berlin. Also English 2nd. rev. ed., 1969, *Enumerability, Decidability, Computability: An Introduction to the Theory of Recursive Functions*, G. T. Hermann and O. Plassmann, trans., Grundlehren der mathematischen Wissenschaften 127, Springer, New York.

[26] Ian Hodkinson, Dec. 2003, “Computability, algorithms, and complexity: Course 240”, course notes. Available at `http://www.doc.ic.ac.uk/~imh/teaching/Turing_machines/240.pdf` (viewed Apr. 12, 2013).

[27] Christopher D. Hundhausen and Jonathan L. Brown, Feb. 2007, “What you see is what you code: A ‘live’ algorithm development and visualization environment for novice learners, *Journal of Visual Languages and Computing* 18(1), doi:10.1016/j.jvlc.2006.03.002, pp. 22–47. Available at `http://www.cs.duke.edu/~rodger/jflappapers/Hundhausen2007.pdf` (viewed Apr. 12, 2013).

[28] Chris G. Langton, 1986, “Studying artificial life with cellular automata”, *Physica D: Nonlinear Phenomena* 22(1–3), pp. 120–149.

[29] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon, July 2012, *The OCaml System Release 4.00: Documentation and Users Manual*, Inria, Le Chesnay, France. Available at `http://caml.inria.fr/distrib/ocaml-4.00/ocaml-4.00-refman.pdf` (viewed July 4, 2013).

[30] Lindsay Marshall and James Webber, 2000, “Gotos considered harmful and other programmers’ taboos”, *Proceedings of the 12th Workshop of the Psychology of Programming Interest Group*, pp. 171–180. Available at `http://www.ppig.org/papers/12th-marshall.pdf` (viewed Apr. 12, 2013).

[31] Norman H. Packard and Stephen Wolfram, 1985, “Two-dimensional cellular automata”, *J. of Statistical Physics* 38(5–6), pp. 901–946.

[32] Seymour Papert, 1980, *Mindstorms; Children, Computers, and Powerful Ideas*, Basic Books, New York.

[33] Emil L. Post, 1936, Finite combinatory processes – Formulation 1, *J. of Symbolic Logic* 1(3), pp. 103–105. Reprinted in Martin Davis, 2004, *The Undecidable: Basic Papers on Undecidable Propositions, Unsolvable Problems and Computable Functions*, Dover Publications, New York, pp. 289–291.

[34] Raimond Reichert, 2003, *Theory of Computation as a Vehicle for Teaching Fundamental Concepts of Computer Science*, Ph.D. thesis, Eidgenössische Technische Hochschule (ETH), Diss. No. 15035, Zurich, Switzerland.

[35] Raimond Reichert, Jürg Nievergelt, and Werner Hartmann, 2005, *Programmieren mit Kara: Ein spielerischer Zugang zur Informatik* (2nd ed.), Springer-Verlag, Berlin.

[36] Wolfgang Reisig, 1992, *A Primer in Petri Net Design*, Springer-Verlag, Berlin.

[37] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai, Nov. 2009, "Scratch: Programming for all", *Communications of the ACM* 52(11) pp. 60–67, doi:10.1145/1592761.1592779.

[38] Denise Schmandt-Besserat, July 1980, "The envelopes that bear the first writing", *Technology and Culture* 21(3), pp. 357–385.

[39] Andrew Scott, Mike Watkins, and Duncan McPhee, Jan. 2007, "A step back from coding – An online environment and pedagogy for novice programmers", *Proceedings of the 11th Java in the Internet Curriculum Conference*, The Higher Education Academy, London Metropolitan University, UK, pp. 35–41. Available at `http://www.ics.heacademy.ac.uk/events/jicc11/scott.pdf` (viewed Apr. 12, 2013).

[40] Claude E. Shannon and Warren Weaver, 1998, *The Mathematical Theory of Communication*, University of Illinois Press, Urbana, IL.

[41] John C. Shepherdson, Apr. 1959, "The reduction of two-way automata to one-way automata", *J. IBM Journal of Research and Development* 3(2), pp. 198–200.

[42] Stanley G. Smith and Bruce Arne Sherwood, Apr. 1976, "Educational uses of the PLATO computer system", *Science* 192(4237), pp. 344–352.

[43] Amitay Stern and David Weinberg, 2013, "2D Turing Machine". Available at `http://www.2dturingmachine.com` (viewed 4 July 2013).

[44] Ian Stewart, 1994, "The ultimate in anty-particles", *Scientific American* 271, pp. 104–107.

[45] Alan M. Turing, Dec. 1936, "On computable numbers, with an application to the Entscheidungsproblem", *Proceedings of the London Mathematical Society* 42, pp. 230–265. Corrections in vol. 43, 1937, pp. 544–546. Reprinted in Martin Davis, 2004, *The Undecidable: Basic Papers on Undecidable Propositions, Unsolvable Problems and Computable Functions*, Dover Publications, New York, pp. 115–154. Available at `http://www.turingarchive.org/browse.php/B/12` (viewed Apr. 12, 2013).

[46] Alan M. Turing, 1949, "Checking a large routine", in *Report of a Conference on High Speed Automatic Calculating Machines*, University Mathematical Laboratory, Cambridge, UK, pp. 67–69. Available at `http://www.turingarchive.org/browse.php/B/8` (viewed Apr. 16, 2013).

[47] Vladimir A. Uspensky, 1983, *Post's Machine*, Mir Publishers (Little Mathematics Library), Moscow. Originally published 1979.

[48] Hao Wang, 1957, "A variant to Turing's theory of computing machines", *J. of the ACM* 4, pp. 63–92.

[49] Mark B. Wells, Oct. 1972, "A review of two-dimensional programming languages", *Proceedings of the Symposium on Two-Dimensional Man-Machine Communication*, *ACM SIGPLAN Notices* 7(10), pp. 1–10 (plus references).

[50] Gregg Williams and Rob Moore, Dec. 1984, "The Apple story, Part 1: Early history", *Byte* 9(13), pp. A68-A69. Available at `http://apple2history.org/museum/articles/byte8412` (viewed Apr. 12, 2013).

[51] Nathan Zeldes, Mar. 2013, "The curator's take on the wonderful CAPTCHA exhibition". Available at `http://www.nathanzeldes.com/wp-content/uploads/2012/11/CAPTCHA-Curator-Take.pdf` (viewed Apr. 12, 2013).