Can learning to program be easy and fun with Numipulator?

Hidden
Hidden
Hidden
Hidden, Hidden

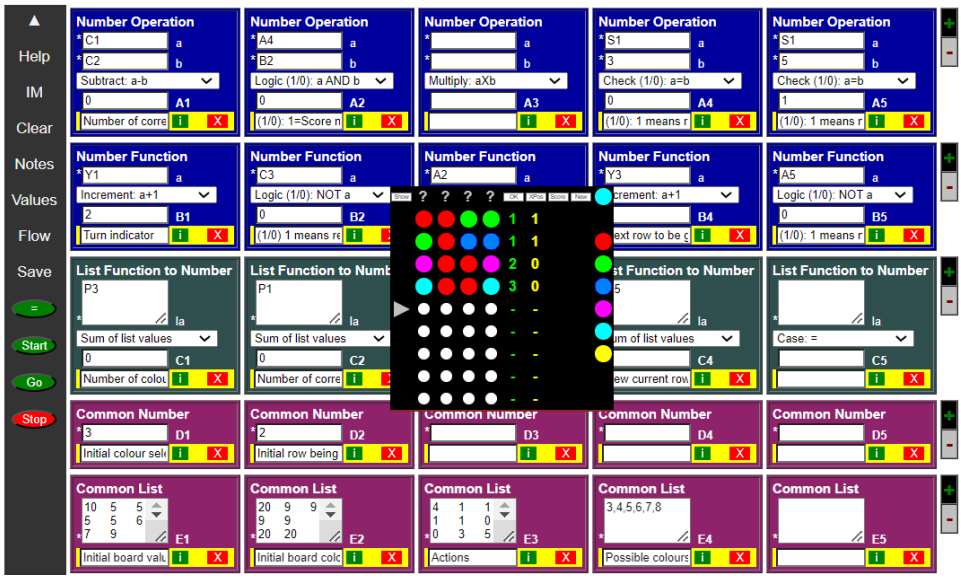

Figure 1: Part of the Numipulator program for the Code4x6 (Mastermind) game, with its graphical output superimposed

ABSTRACT

Research has found that many students find learning to program difficult and boring. This paper explores the features of Numipulator, a novel programming environment, and, through extensive programming in the system and reviewing other research findings, examines the potential use of Numipulator in an educational setting to make learning to program easy and fun. Numipulator has probably the simplest core language syntax in existence and is fully declarative, with no procedural statements or variables. These features, its form-based interface, its graphical formatters that map an input list of numbers directly to the format of the graphical grid cells and its program analysis features appear to make it highly suitable for students learning to program. Its core elements are boxes, each with one or more input fields, and, generally, a drop-down list of functions to be applied to the specified inputs to produce the output. Input and output values may be simple numbers or lists of numbers only. These boxes/functions can be connected, via reference names, to provide the required outputs. Its graphical formatters allow user interaction and, supported by its processing engine with controls (repetitions and delays) specified by evaluated numbers, enable dynamic, interactive graphical apps to be developed. Many of these, including games and puzzles, are built in and can be used as fully-worked programming examples. Students should quickly find that numbers and learning to program can be fun.

CCS CONCEPTS

• **Applied computing** → **Education**; • **Software and its engineering** → **Functional languages**.

KEYWORDS

education, numerical manipulation, declarative programming, functions, lists, games

ACM Reference Format:
Hidden. 2023. Can learning to program be easy and fun with Numipulator?. In *Proceedings of UKICER 2023.* ACM, New York, NY, USA, 7 pages. https://doi.org/XXXXXXX.XXXXXXX

UKICER 2023, September, 2023, Swansea, UK

ACM ISBN ??…$??
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Numipulator is a novel system and programming environment, whose core programming language and syntax is probably the simplest in existence. It is unusual in that its function inputs and outputs are numerical only, each having a value that is either a simple number or a list of numbers. Despite this, it can be used to

develop interactive graphical apps, including letter and word games, through mapping of numbers to graphical cell colours/borders, shapes, text, and emojis. Its simplicity and its program analysis features seem to make it highly suitable for students learning to program, as does the rapid feedback provided to the student when programming.

A key reason for its simplicity is that it is not a general-purpose programming environment, but one particularly aimed at developing interactive graphical apps, such as games and puzzles. It has two different graphical interfaces, both with cellular grids that employ the technique of taking a list of numbers as input and mapping these directly to the formats of the grid cells. For example, the number 3 maps to a cell that is red with a border, 4 to one that is green with a border, and -4 to one that is green but borderless (negative codes always specify no borders).

The simplest of these, the *Animation Zone*, which has an associated 9-key (8 arrows + fire) keypad to provide user-interaction, uses only this list of *cell format codes* to determine its cell displays: just colour and maybe a border. The second one, the *Graphics Formatter*, which allows the user to interact directly with the grid cells by tapping on them, uses a more extensive set of format codes and also has a second list of numbers, the *cell values*, as input, the same length as the cell format codes, which can be used to specify the contents of the cells. The pair of numbers associated with each cell (one from each list) together enable numbers, characters, symbols and emojis to be displayed in cells, with foreground and background colours and borders specified by the cell format codes.

For the Graphics Formatter, simple *output formatting* declarations may also be made, which can be used to map cell values to text/words that replace these numbers in whichever cell they occur. So, if the cell format code for a cell is 3, and its value is -2, the cell, by default, displays the value -2 in a white text field in the middle of a red bordered cell (code 3). If, however, an *output formatting string* is specified *=-2:Score* then the word Score is displayed instead in the text field, wherever the value -2 occurs in the grid. This enables the programmer to design an interface that includes words and symbols, as well as numbers, as seen in Figure 1.

Each of these graphical interfaces has an associated Feedback field for providing information, instructions or scores to the user in text form. This text is derived from a separate evaluated list of numbers, again using output formatting strings to map from numbers to text, if required.

This direct mapping of lists of numbers to graphical cell formats/contents and to Feedback text means that Numipulator requires no output statements (e.g. print or display). In fact, Numipulator has no procedural statements at all as it is a fully declarative environment - one that does not make use of variables.

A further design feature enabling simplicity of syntax is that Numipulator is a form-based environment, consisting mainly of a grid of boxes, as shown in Figure 1. These boxes have one or more input fields and, typically, a drop-down list of functions/operations, from which the user must select one to be applied to the inputs to produce its associated output value (result). Each box also has a reference name, e.g. A1 or L2, that can be used to represent its output value in the inputs of other boxes (similar to usage in a spreadsheet). Use of the reference names in other box inputs means that the boxes can be linked together to produce outputs. These boxes are generally in rows (labelled A-Z) of the same type, with the same input types, output type and set of drop-down functions/operations. The grid initially has 5 boxes in each row named A1...A5, B1...B5, and so on. If users need more of one box type, they may press the + button on the right and a new row will be inserted *underneath* the row, not to the right, with sequential reference names, e.g. A6...A10. NB A programmer cannot write new functions, but does have over 400 built-in functions to choose from.

The basic design of these boxes is modelled on the way most people learn to do basic arithmetic at school, so that users would feel some familiarity with it. A typical school layout is shown below:

```
50
20 -
----
30
```

Each Number Operation box, as shown in Figure 2, is modelled directly on this layout, with two input fields, and then a drop-down list of number operations (including +, -, x, / and % - 28 in all), which essentially combines the line and the operator of the school layout, and the output or result field underneath this. The reference name for the first box shown, A1, is shown next to the result. It can be seen that this reference name is specified in the first input field of the second, A2, so these two are linked. The results are shown in the number fields below the drop-down lists - displayed only after the = or Start button is pressed.

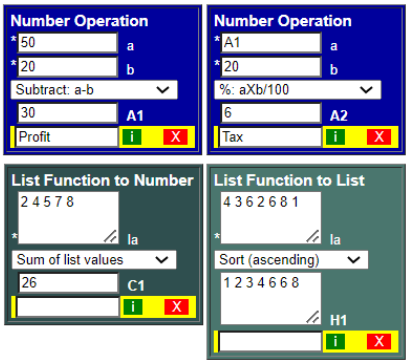

Figure 2: Three box types

Two further types are also shown in Figure 2, this time with lists as inputs. Again, the result fields appear under the drop-down lists of functions. One is the List to Number Function type, with a list as input and a number as output. Available functions (over 30) include: average, maximum value, range, and first item. The other is the List Function to List type, with a list as input and a list as output. In this case, a simple ascending sort has been selected as the function, from a list of over 20, including reverse, cumulative sum, item frequencies and permutation.

Each input field specifies either a number or a list of numbers (the field shapes are distinctive, as seen in Figure 2). A list of numbers is specified by a sequence of tokens, each of which is either a number or a token representing a number or list of numbers, e.g. *pi* or a reference name such as A1 or L2, separated by one or

more separators (including space, comma, tab, semicolon, colon, return/paragraph and *{comment like this}*). The inputs cannot include functions, concatenation symbols or list access indices (e.g. [3]) as might be found in other languages, and no quotes are required. This makes the syntax extremely simple to learn. So, 4 3 2 -7 is a simple list containing four numbers, while if A1 refers to the box that has the number value 20 and L2 refers to the box that has the list value 9 10 11, then the list specification 4 3 2 -7 L2 A1 -8 L2 96 has the list value 4 3 2 -7 9 10 11 20 -8 9 10 11 96.

The Numipulator programming environment facilitates rapid production of graphical results, which can give the student a quick sense of achievement. For example, if box G1 of type List Generation has the function *p1 items of value p2* selected and its input list is specified as 100 3, its output list will contain 100 instances of the number 3. If the cell formats input for the Animation Zone, set up with 10x10 cells, is specified as G1, then, pressing the Start button results in the display of 100 red bordered cells. If the input to G1 is changed to 100 -4, pressing Start again displays 100 green borderless cells. If the function selection of G1 is changed to *p1 random ints 1->p2* and its input list is changed to 100 20, then G1 generates 100 random integers from 1->20, and pressing Start again yields a multicoloured display of bordered cells, as the numbers 1-20 map to different colours. Each of these is a complete, valid program, and the programmer sees the results straightaway.

The development of dynamic, interactive apps requires more than the outputting of one graphical display; it requires a sequence of output displays, changing with or without user interaction. Central to this is Numipulator's *processing engine*. This is triggered whenever the Start button is pressed. By default, it carries out one *processing cycle* that includes one *Function Evaluation*, i.e. the evaluation of all the boxes with inputs specified, and produces any graphical display based on these results. Additionally, a single *repeat-until loop* may be instigated through the specification of a repetitions terminating condition: a simple comparison (e.g. = or >) between two evaluated number inputs. If this condition succeeds (or none is defined) repetitions stop, otherwise another processing cycle, including another Function Evaluation, occurs.

As well as checking for the repeat-until condition, the processing engine also checks two other control values when another cycle is required. The first specifies a delay time before the next processing cycle should begin. This can be used to allow the user of the Graphics Formatter time to interact with the cells of the display grid, or just to slow down or speed up an app. The second is a value that determines whether or not the Graphics Formatter should update its display based on its input lists (useful for hiding certain updates from the user). Both these control values may be specified directly by reference to some other box, so that these controls may change from cycle to cycle. Altogether, these allow the programmer to control the procedural aspects of a program through numerical specification only, with no need to write *for* loops or time-delay mechanisms.

The final element required to develop dynamic apps is the handling of state (e.g. for a game maybe the board positions and the score), without using variables. Numipulator manages state through *memory boxes*, both number and list types. In essence, each memory box has a specification for its main value (in this cycle) and another for its value in any next cycle. At the end of a cycle, the value calculated for the next cycle is taken by the processing engine and used to replace the specification for the main value, so that this will become the value in the next cycle. These memory boxes effectively perform the role of variables in procedural languages when viewed across a series of processing cycles, but they are not variables: they do not change value during a Function Evaluation; the processing engine changes their input specifications only once at the end of every cycle.

In addition to being highly suitable for developing graphical apps, Numipulator (*num*erical man*ipul*ation and calcul*ator*) provides many high-level functions/operations that can be used by students directly in their calculations and numerical analysis in various academic subjects. These functions/operations include statistical functions, various sorts, set operations, find all, counts and many table manipulations and operations. Graphical outputs can also be used to provide data visualization.

Numipulator lends itself to step-by-step introduction in an educational setting, for example: types and use of individual boxes, including input syntax; linking boxes; format codes and static graphical output; repetitions using the Animation Zone; state-handling (e.g. implementing a simple incrementor); state-handling and repetitions with the Animation Zone; use of the Animation Zone to control the movement of an object; use of the Graphics Formatter to produce a simple interactive game, including use of output formatting; and visualization of an algorithm, e.g. a bubble sort.

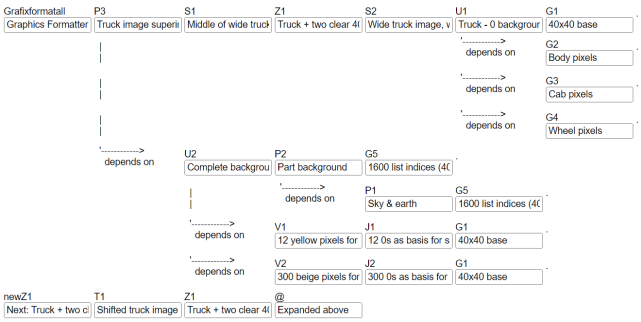

Figure 3: Example of a Flow diagram

Throughout all these steps, Numipulator has certain distinct advantages over most programming environments:

- The values associated with the last cycle for all inputs to boxes and graphical formatters and for all results/outputs can be seen in a bottom-up style by pressing the Values button. This feature can greatly support the student when debugging or dealing with an incomplete program; there is no need to add in print statements to show results so far, as might be the case with other programming environments.
- The data flow can be visualized through the Flow button, which shows the data dependencies of the formatters and boxes in a top-down fashion diagrammatically (Figure 3).
- A set of repetitions may be stopped at any point by pressing the Stop button, so that Values can be inspected.
- Numipulator is a web-based system (developed using Javascript and HTML/CSS only), and is interpreted, not compiled. As

such, the time from programming to receiving feedback is short.

All these can help the programmer quickly to test and debug their programs.

Numipulator has great potential for use by school and college students, both to help them learn programming and achieve their programming objectives and to assist them with the calculations, analyses and manipulations needed for their educational tasks in other academic subjects. Numipulator can be used not only to introduce programming generally to students in a simple and fun way, but also to introduce declarative programming and program specification, as the Numipulator program can be viewed as an executable program specification.

Some researchers consider that the language used for teaching programming should reflect the languages used in industry[8], as students need to be trained in these. However, other research, such as that of Jenkins[1], has concluded that the language used in teaching should be chosen for its pedagogic suitability, rather than for its popularity in industry. As stated by Gomes and Mendes[2], most programming languages were developed for professional use and not to support learning. Bosse and Gerosa[3] and Cheah[4] also refer to the many difficulties with syntax that students encounter with languages used in industry. Learning to program requires many skills, and too often students fail to get over the first hurdle of syntax in the hierarchy of skills[1], so cannot move on to problem solving: analysis, algorithm development, coding and testing.

In his Scientific American article [7], Alan Kay writes of the importance of computer literacy: a contact with computing deep enough to be "fluent and enjoyable", not just using applications such as spreadsheets or word-processors. Synonyms for "enjoyable" include "entertaining" and "fun", and this was one of the objectives of Numipulator: to make it entertaining and fun to use. This it does by providing a means of developing graphical displays, games and puzzles, thereby demonstrating that numbers can be fun and helping users develop programming competency and computer literacy.

2 GRAPHICAL FORMATTERS AND OUTPUT MAPPING

Both the Animation Zone and the Graphics Formatter make use of cell format codes to map directly to the displays of individual grid cells. The Animation Zone maps the integers 1-20 to the colours white, black, red, green, blue, magenta, cyan, yellow, grey ... through to beige for 20. These are Numipulator's *simple colour codes*. Numipulator also has a set of *short RGB codes* that can be used to specify 1000 different colours, rather than just 20. The codes are the integers 1000-1999, and their general form is 1RGB, where the numerals at positions R, G and B specify the intensity of Red, Green and Blue light, from 0 (none) to 9 (full intensity). Some examples are: 1000=Black, 1900=Red, 1990=Yellow, 1999=White, while 1512 is some shade of brown. Long RGB codes can also be used, which enable the user to specify millions of colours.

In all these cases, these positive integers specify that the cell should have a border. In contrast, if the negative forms of these integers are specified, then the cells have no borders.

The Graphics Formatter uses these same codes, but also displays a small white field centrally that displays the value associated with the cell, as described earlier. These values are subject to any output text mapping specified, which can be used to specify that symbols, letters or words be displayed instead of the value. This mapping can also be specified as NOTEXT, in which case no text or text field is displayed.

Code 0 is used with both to specify that a black, borderless cell be displayed, used primarily to create a black background.

The Graphics Formatter can also accept additional codes. The codes in the first set take the form NFGBG, where N is an integer from 1-20, while FG and BG are two-digit numbers from 01 to 20, representing the simple colour codes 1-20, or 00 representing black. These represent the foreground and background colours of the cell. When N is in the range 1-17, some shapes that are useful in games are displayed, as shown in Figure 4. So, code 30504 results in the display of a large circle (3) coloured blue (05) on a green (04) background. With these codes, the associated value is ignored.

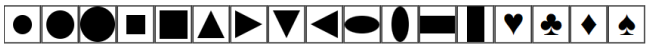

Figure 4: Shapes for codes N=1-17

For N=18, the associated value represents a decimal entity code for HTML characters, symbols and emojis (specifying a Unicode character). This enables many more objects to be displayed than N=1-17. So, if the format code is 180503 and the associated value for the cell is 9733 (the code for a star), a blue (05) star on a red (03) background is displayed. For N=19, the associated value (with any output formatting applied) is displayed with a large font size and no white text field; this is suitable for one or maybe two characters only. If N=20, the value is used to select an image to be displayed from a set of URLs specified line-by-line in a URLs text field; the associated value 2 specifies the 2nd of these.

The codes in the second set are very similar to those in the first set, but have the form NFGCBGC, where FGC and BGC specify the foreground and background colours using the three digits of the short RGB codes.

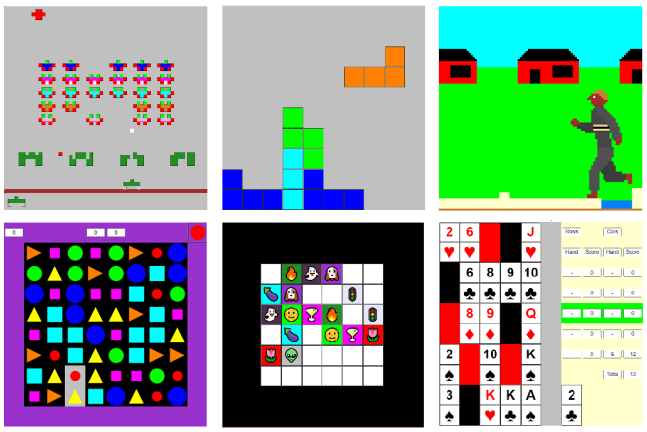

Figure 5: Graphics displays for six games

The images in Figure 5 give an idea of the interfaces that can be achieved with these codes. The top three come from the Animation Zone, while the bottom three come from the Graphics Formatter. Apart from the third one (Fun Runner), only the 20 simple colours are specified. The development of these games testifies to the power of Numipulator, as they are by no means trivial. The first game, like Space Invaders, is a highly dynamic and interactive game, making use of the keypad. The second game is inspired by Tetris. The fourth game is similar to Bejeweled. The fifth is a memory game to find Emoji Pairs. The sixth is a form of Poker Squares (card face values are displayed using N=19 codes).

3 NUMBERS, LISTS AND TABLES

The handling of lists is a core functionality of Numipulator and is easy to carry out. Numipulator has only two main value types: numbers and lists of numbers. Every input and output is specified to be one of these. In Numipulator, the user can enter a list of numbers into the input field of a List Function to Number box, select *Sum of list values* and the output can be calculated (see Figure 2). Three types of box focused on lists have already been discussed: List Generation, List Function to Number and List Function to List. In addition, List & List Operation includes sort by and set operations (Common, Union, Diff & SDiff), while List Operation accepts sets of parameters to return items, percentiles, sublists, counts and shifts and many more.

A distinctive feature of Numipulator is its use of item-by-item list handling to make calculations involving the same operation on lists of numbers easier to specify. For example, the Items Operation box carries out the same operation on corresponding items in two lists to give the resulting list, as shown in the first example of Figure 6. If the second list consists of just one item, it will carry out the same operation using this number on each of the items in the first list, as shown in the second example.

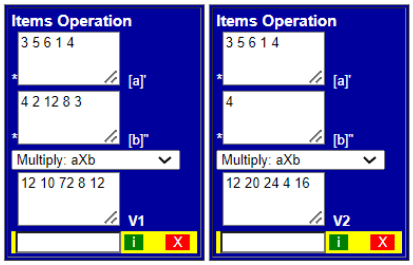

Figure 6: Two examples of Items Operation 1) with list of equal length; 2) with single-item list

Tables are also heavily used in Numipulator, particularly for graphical apps, as the graphical grids are essentially tables. A table is not a core type, but is always specified by two inputs: a values list and the number of columns in the table. Table-specific box types are Table Function (transpose, various sorts, flips, rotations and shifts), Table Operation (including rows, columns, blocks, and cluster analysis), and Table Construction (including merge, line changes, cell changes and block changes: these are particularly valuable in graphical apps, and some are only for graphics, e.g. fill circle). In all cases, if the conceptual result of the function/operation is a table (e.g. a rotated table), the output is a list of the table values (extracted row-by-row).

Numbers and lists are clearly different data types. However, a reference to a list may be specified as a number input, in which case the number value is taken to be the first or only number in the list. The corollary of the semantics of this is that a list output having only one number on it may be used in all cases exactly as if it were a number output. This *loose typing* is also evident in the fact that a list specification may include reference names to both numbers and lists; this helps to keep the syntax simple.

4 PROGRAMMING - OBSERVATIONS AND ISSUES

The observations and issues in this section were identified through extensive programming using Numipulator, mainly while developing games and puzzles.

When specifying inputs, there are, as standard, only 6 reserved words, apart from reference names, that can be used to represent numbers or lists: *pi* and *e* (mathematical constants); *L0*, which represents an empty list (also used in outputs); *now*, a list of numbers concerned with the current time (year through to second); *click*, taking the value 1 or 0 depending on whether a graphics cell has been tapped/clicked or not; and *keypad*, taking values 0-9 depending on the key pressed (if any) on the Animation Zone keypad. The values of the last three are determined once only just before a Function Evaluation takes place, and remain constant throughout that cycle. The programmer can also define their own input *Mappings*, to allow a word, character or symbol to represent a number in inputs. This proved useful when programming word/letter games, as Mappings could be defined as A -1 B -2 C -3 ... Z -26 so that word lists could be input as, for example, O R A N G E : B A N A N A but be treated as list -15 -18 -1 -14 -5 -2 -1 -14 -1 -14 -1 by Numipulator. These Mappings could also be used as output mappings.

Code4x6, see Figure 1, is a logic puzzle, like Mastermind[9], in which the player makes guesses as to what colours are hidden at the top, and requests scores for these guesses from which they can make better guesses, until they solve it. The program for this uses the Graphics Formatter. It does not use repetitions; when the user taps on a cell, this triggers a single processing cycle and Function Evaluation. With the Graphics Formatter it proved useful to define three lists associated with grid cells: the values, the colours/format codes and the actions. The first two together form the basis for the initial appearance of the cell, while the third specifies what action is requested when the user taps on that cell. Tapping on a cell changes the number specifications of two boxes representing the row and column numbers of the cell. These values are used by the programmer to determine the action number requested in any cycle from the actions list. Figure 7 shows the inputs for three boxes, E1, E2 and E3, associated with Code4x6 (also seen partly in Figure 1). The inputs are all 10x10 formatted tables, created in a text editor and pasted into the input fields. Any such formatting is for the benefit of the programmer only, enabling them to relate the numbers to the grid layout more easily by sight. Numipulator, however, treats the tabs and returns as separators, so interprets

them as simple 100-item lists. Three things should be noted about these lists:

- For E1 values, output mapping is required to produce text in the display, e.g. *=5:?* and *=9:Score*.
- The codes of E2 are taken and mapped, using a built-in function, to give the final cell codes, e.g. 5 (a blue bordered square) is mapped to -20500 (a medium blue circle on a black borderless background). Such mapping allows the programmer to change the interface appearance easily if required.
- The action numbers of E3 have meanings for the programmer, e.g. 1 means "place the selected counter (top-right) in the active row of this column", 3 means "display the score for the active row" and 0 indicates "no action".

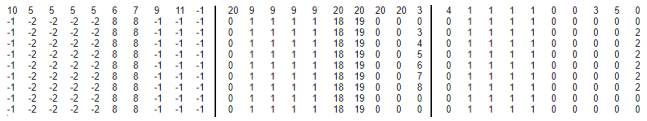

Figure 7: Inputs for E1 (Values), E2 (Colours/format codes) and E3 (Actions) of Code4x6

Although the word "actions" was used above, this can be misleading, as Numipulator has no procedural statements and Function Evaluation can *do* nothing, apart from evaluate and output numbers and lists, some of which can be displayed graphically, and some of which can update the state, through memory box specifications. To understand how Numipulator works, and one of its issues, it can be instructive to consider what happens when the user taps on a cell. The action of tapping triggers a processing cycle and Function Evaluation, which will evaluate *all* boxes, including determining the value of the action number from E3. This evaluation must be able to determine the outputs for *any* cell the user might tap. The programmer would typically make use of a chain of If-Then-Else boxes to determine the display outputs and new state values for the selected action number. These If-Then-Else boxes are not like *if* statements in procedural languages, which determine which alternative procedures to execute; instead they each have four inputs, two specifying values in a comparison condition and two specifying alternative list values, and determine the result from the alternatives according to the result of the condition. With Numipulator, all function inputs must be fully evaluated before the result is evaluated, which means that all alternative values must be calculated in every cycle. Therefore, it must calculate the scores in case action 3 is selected, even when action 0, 1 or any other action is selected. So, unlike with a procedural program, where the scores would be calculated only if that request was made, Numipulator calculates the scores in every cycle, regardless of the action request. Clearly, there is an issue of inefficiency involved in this. Likewise, for the Emoji Pairs puzzle (Figure 5), 18 pairs of emojis are permuted to determine the initial hidden layout, but the function to perform this will be evaluated in every cycle, even though this new permutation will not contribute to the display in any later cycle.

Although the programming is declarative, the procedural aspects of an app are crucial and must still be considered carefully by the programmer. The Bejeweled play-alike game (Figure 5) is fairly complex, so, when programming this, a full game analysis was carried out, which identified 9 different phases, including filling, checking adjacency, swapping, checking for lines of 3 or more, removing jewels and collapsing jewels. As with procedural programming, a flow diagram showing the links between these phases proved invaluable. These phases were numbered, and much of the code was built around the current and next phase numbers, including the specifications of the control values (repetition termination, display and delay), since those associated with each phase were found to differ considerably: some phases require multiple repetitions, others only one; some require the display to be updated after a cycle and a delay for the user to see this or to tap, while others don't.

The greatest difficulty encountered when programming a complex app in Numipulator was programmer confusion between the current and the next state (the main and new specifications of memory boxes). This sometimes led to circularity issues.

5 APPS: BUILT-IN AND STANDALONE

The program for an app being developed, i.e. the complete set of box inputs, selected functions, notes, description, title and settings, can be exported to a formatted text file. This can be easily reloaded or shared and can be used to create a standalone browser app, which should run independently on most browsers, using a separate Numipulator Standalone Maker. Many apps have been created using Numipulator, many of which are built into the system, including utilities (e.g. a Calendar), many games and puzzles, some already mentioned, and programming-support apps (e.g. a Unicode Symbol Finder). Some of these apps, including a simple, dynamic, interactive game (using 24 boxes only) and a demo of controlling the movement of an object, are specifically intended to demonstrate simple, declarative programming with Numipulator.

Research by Tan et al[5] to determine the learning methods that are preferred by university students concluded that the students rated learning by referring to programming examples the highest. The built-in apps provide many of these.

6 CONCLUSION

Despite its novelty and the issues identified, Numipulator could offer many benefits to students learning to program: its simplicity of syntax; its form-based environment allowing selection of functions (no chance of spelling mistakes); its program analysis features, especially its Values feature enabling the programmer to see all values in any evaluation cycle; its built-in programming examples, and, probably most importantly, its enablement of easy, rapid production of graphical results, which can give the student an early sense of achievement, while having fun using the Animation Zone and Graphics Formatter. Jenkins[1] found that the perception that programming is difficult and boring is a barrier to teaching it. In contrast, having fun while learning to code has been shown [6] to improve students' attitudes towards coding and their motivation. Numipulator has not yet been used in an educational environment, so further research into such use is required. Cheah[4] refers to the positive correlation between motivation and achievement: the higher the motivation, the greater the achievement. It would be interesting to determine whether the ease of achieving early results with Numipulator could, in turn, lead to greater motivation of students.

REFERENCES

[1] Tony Jenkins. 2002. On the difficulty of learning to program. *Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences* 2002, 4, 53–58.

[2] Anabela Gomes and A.J. Mendes. 2007. Learning to program - difficulties and solutions. *International Conference on Engineering Education–ICEE.* 2007.

[3] Yorah Bosse and Marco Aurelio Gerosa. 2017. Why is programming so difficult to learn?: Patterns of Difficulties Related to Programming Learning Mid-Stage. *ACM SIGSOFT Software Engineering Notes* 2017, 41(6), 1-6

[4] Chin Soon Cheah. 2020. Factors Contributing to the Difficulties in Teaching and Learning of Computer Programming: A Literature Review *Contemporary Educational Technology* 2020, 12(2), ep272

[5] Phit-Huan Tan, Choo-Yee Ting and Siew-Woei Ling. 2009. Learning Difficulties in Programming Courses: Undergraduates' Perspective and Perception. *2009 International Conference on Computer Technology and Development, Kota Kinabalu, Malaysia* 2009, 42-46.

[6] Gabriella Tisza and Panos Markopoulos. 2020. Understanding the role of fun in learning to code.*International Journal of Child-Computer Interaction* 2020, 48.

[7] Alan Kay. 1984. Computer software. *Scientific American* 251, 3 (Sep. 1984), 52–59.

[8] Andreas Stefik and Susanna Siebert. 2013. An Empirical Investigation into Programming Language Syntax. *ACM Trans. Comput. Educ.* 13, 4, Article 19.

[9] Amy R Strom and Scott Barolo. 2011. Using the Game of Mastermind to Teach, Practice, and Discuss Scientific Reasoning Skills. *PLoS Biol 9(1)* 2011, e1000578.