Element Reference Programming and Fun with Numipulator

David Wolstenholme

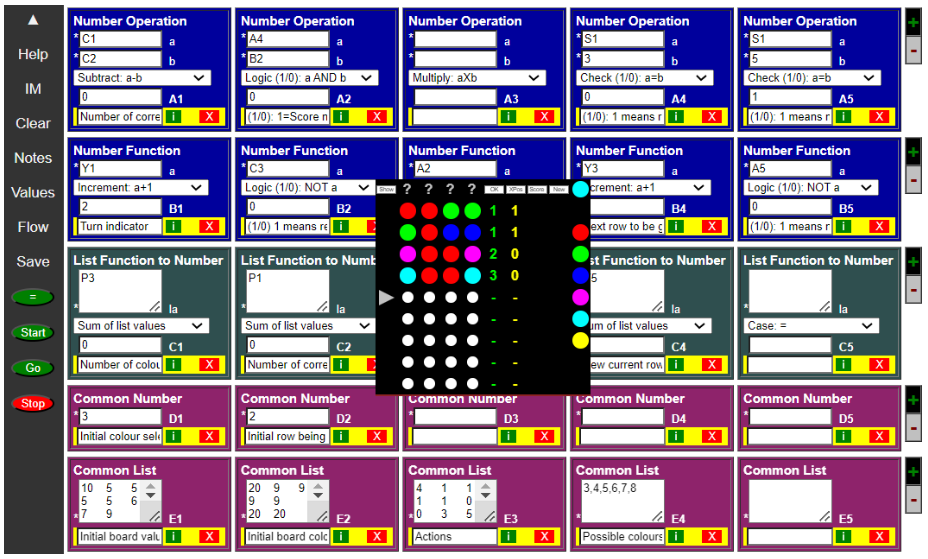
TopAccolades Ltd `wolstede@yahoo.com`
www.numipulator.com

Abstract. Numipulator is a fully declarative programming environment with a very simple syntax. Its core elements are boxes, each of which has one or more input fields, and, generally, a drop-down list of built-in functions to be applied to the inputs to give the output. Inputs and outputs may be either simple numbers or lists of numbers only. Each box has a reference name that can be used in inputs of other boxes to represent its value, thus allowing boxes/functions to be linked up to provide the required outputs through a Function Evaluation cycle. It has two graphical formatters that can be used to provide graphical outputs (grids of cells), derived from one or two lists of numbers, principally through the use of format codes specifying shapes/symbols, colours and borders. It has a simple processing engine that allows repeated Function Evaluations to be specified. The control of repetition termination, the delay till the start of the next cycle and the display or not of graphics after any cycle can be controlled by Function Evaluation results, thus giving dynamic declarative control. Using these, plus Memory boxes to manage state, and number<->text mapping specifications, sophisticated graphical apps, including ones similar to well-known interactive games, can be developed, showing that declarative programming with functions and numbers can be fun.

Keywords: Numbers · Lists · Graphics · Games · Reference Names.

1 Introduction

Numipulator is a novel, fully declarative programming environment and language, intended for students, hobbyists and those prototyping, that has possibly the simplest syntax of any high-level programming language. At its heart is a set of boxes, each of which specifies a numerical manipulation or calculation (hence the system name) to be carried out. These are mostly laid out in rows of the same type, labelled from A to Z. Some of these are shown in Figure 1. Each box can be considered as a form specifying a function (a few are non-deterministic, such as random number) to be evaluated by the Function Evaluator. Input fields specify the inputs to the function. Each input field specifies either a simple number or a list of numbers (the field shapes are distinctive, as seen in Figure 1). The result of each will also be either a simple number or a list of numbers, normally displayed in an output field in the box. Each box has a reference name,



Number Operation — a: C1; b: C2; Subtract: a-b; A1: 0; Number of corre

Number Operation — a: A4; b: B2; Logic (1/0): a AND b; A2: 0; (1/0): 1=Score n

Number Operation — Multiply: aXb; A3

Number Operation — a: S1; b: 3; Check (1/0): a=b; A4: 0; (1/0): 1 means r

Number Operation — a: S1; b: 5; Check (1/0): a=b; A5: 1; (1/0): 1 means r

Number Function — a: Y1; Increment: a+1; B1: 2; Turn indicator

Number Function — a: C3; Logic (1/0): NOT a; B2: 0; (1/0) 1 means re

Number Function — a: A2

Number Function — a: Y3; crement: a+1; B4; ext row to be g

Number Function — a: A5; Logic (1/0): NOT a; B5: 0; (1/0): 1 means r

List Function to Number — la: P3; Sum of list values; C1: 0; Number of colou

List Function to Numb — la: P1; Sum of list values; C2: 0; Number of corre

st Function to Number — la: 5; um of list values; C4; ew current row

List Function to Number — Case: =; C5

Common Number — D1: 3; Initial colour sel

Common Number — D2: 2; Initial row being

Common Number — D3

Common Number — D4

Common Number — D5

Common List — E1: 10 5 5 / 5 5 6 / 7 9; Initial board valu

Common List — E2: 20 9 9 / 9 9 / 20 20; Initial board colo

Common List — E3: 4 1 1 / 1 1 0 / 0 3 5; Actions

Common List — E4: 3,4,5,6,7,8; Possible colours

Common List — E5

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

e.g. A1 or C2. These reference names can be used to link the boxes, and hence the functions, together by specifying that the result of one be the input, or part of the input, to another, in a way that is similar to that used in spreadsheets.

There are different types of box, with one or more inputs, and generally a drop-down list of function names/descriptions, one of which is selected by the programmer to be applied to the specified inputs to give the resulting value. The types include: simple Number Operations and Number Functions, as shown in Rows A and B in Figure 1; various list functions and operations, such as those shown in Row C, that take a list of numbers and produce a number as output, e.g. sum of list; list analysis (Findall, Count and Range); If-Then-Else expressions; and table-handling functions and operations (where a table is defined by two inputs - a list of values and the number of columns in the table), which also include graphics-specific functions.

Given that its core functions can accept only numerical input values and have numerical results, Numipulator is not a general purpose programming environment. However, apart from being eminently suitable for numerical applications, it has a number of features that make it particularly suitable for programming interactive games and puzzles.

It has two graphical interfaces, both with a grid of square graphical cells: the Animation Zone (up to 150x150 cells) and the Graphics Formatter (10x10 default). The Animation Zone has an associated 9-key keypad (8 arrows + fire) allowing the user to interact with graphical objects on the display. Tapping one of these sets an internal value called *keypad* to the value of the key pressed (1-9). In contrast, the second one, the Graphics Formatter, has no keypad; instead, the user interacts directly with the cells by tapping on them. This changes two input values, representing Row and Column number, and also sets an internal value called *click* to 1. The programmer must specify initial (*init*) values for Row and Column if used.

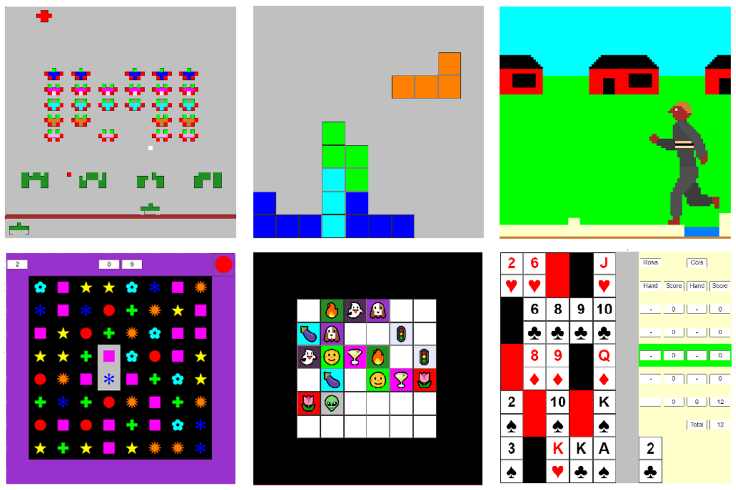


Fig. 2. Graphics displays for six games

The displays generated in the simplest one, the Animation Zone, are determined from a single input list, the *cell format codes*, generally specified by the reference name of one of the boxes that has a list result. Each number on the list corresponds to one of the cells on the grid and is taken to be an integer format code that specifies the colour of the cell and whether or not the cell should have a border. The simplest codes are the codes 1-20, each of which specifies that the cell should have a border and should take a specific background colour, e.g. 1 means white, 2 black, 3 red, 4 green, 5 blue, 6 magenta and so on to 20 specifying beige. Code 3 associated with a given cell therefore results in that cell having a red background and a border. Negative equivalents of these codes have the same colour but no border, while code 0 specifies a black borderless cell. So, a list beginning -3 0 -5 4 will result in a display having three borderless cells in the top-left corner, red, black and blue, followed by a bordered green cell. Most games and puzzles included in Numipulator for the Animation Zone use only the simple codes -20 to 20. The top three images of Figure 2 show image captures for three games that use the Animation Zone: Alien Invasion (like Space Invaders), Falling Shapes (like Tetris) and Fun Runner. Additionally, Numipulator has a set of format codes used to specify 1000 different colours, rather than just 20. The codes are the integers 1000-1999. The form of the code is 1RGB, where the RGB numerals specify the intensity of the three colours red, green and blue, from 0 (no intensity) to 9 (full intensity). So, code 1990 specifies yellow (full intensity red and green, but no blue). Other codes based on RGB intensities specify millions of different colours. In all cases, negative numbers are borderless.

The Graphics Formatter uses two input lists to produce its graphics. One is again a set of cell format codes, while the other is a list of values, the same length as the format codes. Given the same format code as for the Animation Zone, the cell will again display the specified background colour, but the corresponding number on the value list is displayed in a text field in the middle of the cell. However, additional codes are also available for the Graphics Formatter that enable many thousands of shapes and Unicode symbols, characters and emojis to be displayed, where the codes include specification of the foreground and background colours to be used, based on the Animation Zone colours (e.g. 0 - 20). So, code 860305 specifies a star (86) that is red (03) on a blue (05) background with a border. Some of these additional codes make use of the associated value, but others, such as the example just given, ignore it entirely.

The programmer may also employ *output mapping*, a different type of simple declarative programming. Output mapping is specified through declarations that specify text to be output instead of the associated value, e.g.
-1 A
-2 B ...
-26 Z
<-26 No way!
n NOTEXT
The first line means that -1 should be displayed as 'A'. The penultimate line means that any number below -26 should be displayed as 'No way!'. The last line specifies that any other number should be mapped to the special word NOTEXT, which means that no text field is displayed at all. Such mappings enable letters, text buttons or labels to be displayed. The bottom three images of Figure 2 show images from Graphics Formatter games and puzzles: Triple Jewels (like Bejeweled), Memory Pairs and Poker Squares (with text labels).

Being fully declarative, Numipulator needs no procedural statements to display the graphical output, just the specification of input number lists. Both graphical formatters also have a Feedback field associated with them to provide messages, scores, instructions, etc. to the user. Each Feedback has its own input specification and output mappings to allow suitable messages to be presented.

Apart from the lack of procedural statements to be written, a major contributor to the simplicity of Numipulator for the programmer is the syntax when specifying a list input. A list input field can contain numbers, e.g. 2 or -3.6, separated by one or more number separators: space, comma, tab, semicolon, colon, paragraph/return or comment in a pair of braces {like these}. A list specification may also be or include a token that represents a list or a number, including a reference name, e.g. A1. Except for reference names, there are only 6 built-in tokens for numbers and lists, including *pi* and *L0*, which specifies an empty list (a list of length 0). Apart from these tokens, numbers, reference names and separators, nothing else appears as standard in a list input specification: no parentheses, square brackets, quotes, or function names. Furthermore, the use of punctuation symbols and returns as number separators is optional for the programmer (spaces only could be used), and need be included for the programmer's benefit only, where their use helps the programmer in some way, e.g. for grouping sets of parameters or for formatting the list as a table. Regardless of the formatting and the separators used, the system treats the input as a simple list in all cases. So, the list 1 2 3 D1 5 6 could be entered exactly like this or as, for example:

 1, 2 : 3
 D1 {the initial number of items}; 5 : 6

Further tokens *may* be used in input specifications, at the programmer's discretion, to represent individual numbers, through the use of *input mapping.* This is another form of very simple declarative programming. The programmer enters token/number pairs in the *Mappings* field, e.g. A -1 B -2 C -3 ... Z -26, so that the letter A may be used in any input instead of -1, etc. A benefit of this from the game/puzzle programmer's perspective is that letters making words can be entered into list inputs, e.g. A P P L E : M A N G O : G R A P E which, as a word list, is, of course, much more meaningful for the programmer than -1 -16 -16 -12 -5 -13 -1 -14 -7 -15 -7 -18 -1 -14 -5.

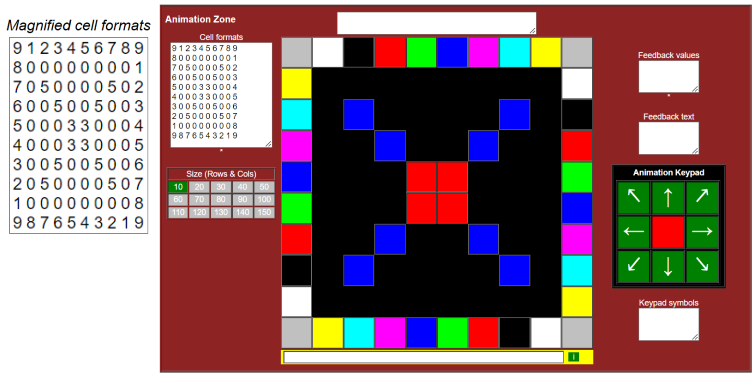
Numipulator includes an A-Z button below the Mappings field. Pressing this inserts the above mappings into the field. To simplify the programmer's effort further, the output mapping declaration associated with the Graphics Formatter may have just the single word *Mappings* entered into it, which essentially reverses the input mappings. So, despite the fact that the core Function Evaluator handles only numbers and number lists, word/letter games and puzzles, e.g. Word Ladder or Wordle-like puzzles, may be implemented with extensive word lists using these mapping declarations.

Numipulator also includes an Expression function that enables arithmetic and logical expressions to be evaluated, where number codes represent operators, e.g. 1=add, 2=subtract, 3=multiply. With this, the expression 10+11x20 can be evaluated by the list 10 1 11 3 20. Input mapping can be applied to these operator codes (a button can be used to add these) so that this can be written as 10 + 11 x 20. This mapping enables mixed arithmetic and logical expressions such as A1 > 5 + A2 AND A3 < 25 OR A4 = A5 to be entered and evaluated (1 or 0 instead of True or False).

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*, which incorporates a single *repeat-until loop.* 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 then terminates and produces any graphical display based on these results. Additionally, a repetitions *termination condition* - a simple comparison (e.g. = or >) between two numbers - may be specified. If this condition succeeds, repetitions stop, otherwise another processing cycle, including another Function Evaluation and output display, occurs, and so on.

As well as checking for the termination condition, the processing engine also checks two other *control values* when another cycle is required. *Control1* 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 to change the speed of an app. *Control2* 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). These two control values and the two numbers in the termination condition may all be specified directly by a number or by reference to other boxes. This means that they may change from cycle to cycle. This *dynamic declarative control* is important for the programmer, as the procedural aspects of an app can be crucial. In a game with several phases, for example, 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 program must therefore include functions that produce different control values (repetition termination, delay and display) for the different phases.

The final element required to develop dynamic apps is the handling of state - 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 the 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. Although these Memory boxes are used to store state, they are not variables: their values remain constant during a single Function Evaluation; the processing engine changes their main input specifications once only after an evaluation cycle. In summary, state is managed by changes to input specifications, i.e. by changes to the program, carried out automatically by the processing engine.

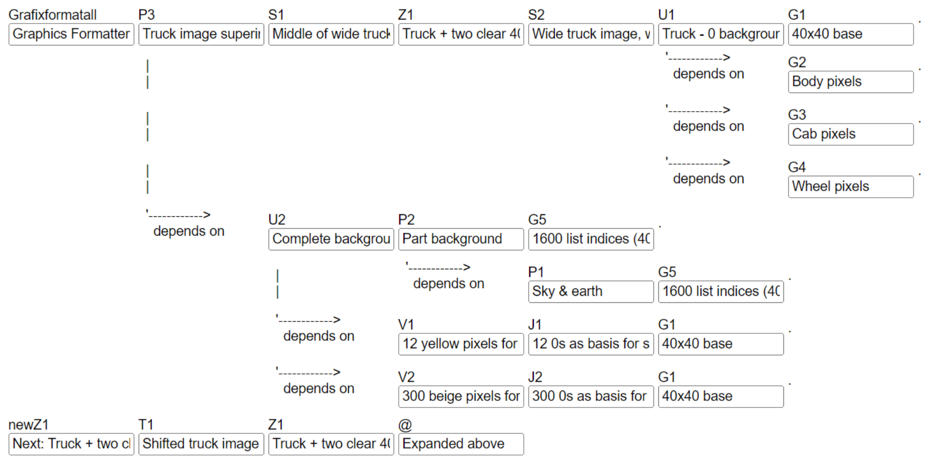

Magnified cell formats

```
9 1 2 3 4 5 6 7 8 9
8 0 0 0 0 0 0 0 0 1
7 0 5 0 0 0 0 5 0 2
6 0 0 5 0 0 5 0 0 3
5 0 0 0 3 3 0 0 0 4
4 0 0 0 3 3 0 0 0 5
3 0 0 5 0 0 5 0 0 6
2 0 5 0 0 0 0 5 0 7
1 0 0 0 0 0 0 0 0 8
9 8 7 6 5 4 3 2 1 9
```

Fig. 3. Simple graphical layout with direct entry of cell formats (codes magnified)

The Numipulator programming environment facilitates rapid production of graphical results. For a student learning to program with Numipulator, this can give a quick sense of achievement. As an example of rapid graphical results, the programmer could enter the codes directly into the Cell formats field of the Animation Zone. The graphical output resulting from pressing the = button is displayed immediately; this is shown in Figure 3. This clearly demonstrates the advantages of being able to format an input list, in this case as a 10x10 table, as the association of code with cell can be easily seen.

For the programmer, Numipulator has certain distinct advantages over many programming environments when testing or debugging. 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. A set of repetitions may be stopped at any point by pressing the Stop button to allow this. 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 4). It is a web-based system (Javascript and HTML/CSS only), and is interpreted, not compiled, so the time from programming to receiving feedback is short.



```
Grafixformatall      P3                   S1                   Z1                   S2                   U1                   G1
Graphics Formatter   Truck image superi   Middle of wide truck Truck + two clear 4( Wide truck image, v  Truck - 0 backgrour  40x40 base
                                                                                                        '----------->        G2
                                                                                                          depends on         Body pixels
                                                                                                        '----------->        G3
                                                                                                          depends on         Cab pixels
                                                                                                        '----------->        G4
                                                                                                          depends on         Wheel pixels
                     '----------->        U2                   P2                   G5
                       depends on         Complete backgrou    Part background      1600 list indices (4(
                                                               '----------->        P1                   G5
                                                                 depends on         Sky & earth          1600 list indices (4(
                                          '----------->        V1                   J1                   G1
                                            depends on         12 yellow pixels for 12 0s as basis for s 40x40 base
                                          '----------->        V2                   J2                   G1
                                            depends on         300 beige pixels for 300 0s as basis for  40x40 base
newZ1                T1                   Z1                   @
Next: Truck + two cl Shifted truck image  Truck + two clear 4( Expanded above
```

Fig. 4. Example of a Flow diagram

One downside with Numipulator is that it can be difficult for the programmer to see the details of the list inputs in the small fields, as these could contain many numbers and comments. However, there is a *Text* button that displays the full program text in a more easily readable top-to-bottom layout.

These features and its syntax mean that Numipulator has great potential for use by students, not only as an introduction to declarative programming, but also for learning programming generally in a simple and fun way. 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. Bosse and Gerosa[2] and Cheah[3] refer to the many difficulties with syntax that students encounter with languages used in industry; this should not be the case with Numipulator. Numipulator programming will appear as a course for the University of the Third Age (U3A) in Kingston in 2024.

Tisza and Markopoulos[4] emphasize the role of fun in learning to program, while in his Scientific American article[5], Alan Kay writes of the importance of computer literacy: not just using applications such as spreadsheets or word-processors, but a contact with computing deep enough to be "fluent and enjoyable". One of the objectives of Numipulator development was to make it enjoyable and fun to use for hobbyists and students by providing means of creating graphical displays, games and puzzles, so demonstrating that numbers can be fun and helping users develop programming competency and computer literacy.

2 Element Reference Programming and Relationship to Spreadsheets

Spreadsheets influenced the design of Numipulator. A spreadsheet such as Excel is widely used for handling lists, yet a list is not a core data type. A user wishing to sum up a list of numbers must enter them into separate cells, and then enter a formula in another cell to determine their sum, e.g. =SUM(A1:A3). An initial goal for the design of Numipulator was to make the handling of lists a core functionality and easier to carry out than in spreadsheets by making a number and a list of numbers the core data types of Numipulator. This allows the list and the sum to be encapsulated in one box.

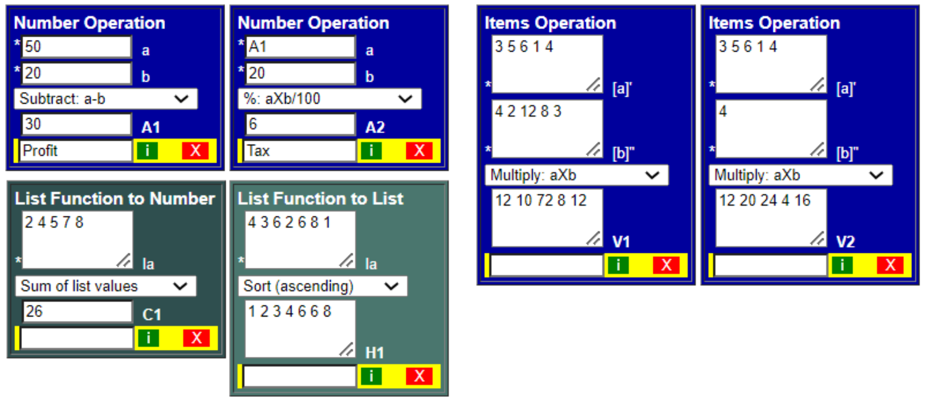
Spreadsheets are very widely used and, in their purest form, are declarative in nature, making them probably the most widely used form of declarative programming. Their use of reference names to refer to the value of another cell is easy to understand, and the naming convention helps users locate the cell being referred to. Numipulator also follows this approach, but these reference names are used only to help locate an individual box; there is no concept of a range (such as A1:A3) as found in spreadsheets.

Numipulator and spreadsheets have much in common. Both provide the user with an extensive set of functions (over 450 in Numipulator) that can be used to determine the value of their core evaluation elements (cells in spreadsheets, boxes in Numipulator), and their values can be specified as inputs to other elements through reference names. Both also employ bottom-up evaluation of their elements. Now, Burnett et al [6] describe the spreadsheet paradigm as a first-order subset of the functional programming paradigm. However, functional programming is generally characterized by its treatment of functions as first-class citizens and by its use of recursion, amongst other features. These do not apply to a spreadsheet like Excel or to Numipulator, so both are probably better described as programming *with* functions or as *element reference programming*, due to their use of reference names to refer to evaluation elements.

3 Boxes: Structure, Layout and Types

Most of the boxes of Numipulator are laid out in rows of 5 boxes (see Figure 1). The boxes have reference name labels, e.g. A1, A2, A3, A4 and A5, for row A. Users can easily insert a new row *underneath* the current row, with sequential reference names, e.g. A6...A10.

The basic design of these boxes is modelled on the way most people learn to do basic arithmetic at school, with inputs, an operator, and then a line with the result underneath. Each Number Operation box, as shown in Figure 5, is modelled 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, 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. Two further types are also shown in Figure 5 below the Number



Number Operation
50 a
20 b
Subtract: a-b
30 A1
Profit

Number Operation
A1 a
20 b
%: aXb/100
6 A2
Tax

Items Operation
3 5 6 1 4 [a]'
4 2 12 8 3 [b]''
Multiply: aXb
12 10 72 8 12 V1

Items Operation
3 5 6 1 4 [a]'
4 [b]''
Multiply: aXb
12 20 24 4 16 V2

List Function to Number
2 4 5 7 8 la
Sum of list values
26 C1

List Function to List
4 3 6 2 6 8 1 la
Sort (ascending)
1 2 3 4 6 6 8 H1

Fig. 5. Examples of boxes

Operation boxes, this time with lists as inputs. Again, the result fields appear under the drop-down lists of functions. One is List Function to Number (Row C), with a number as output, which has over 30 functions available including: sum, average, maximum value, expression, and head. The other is List Function to List (Row H), with a list as output, where an ascending sort has been selected from over 20 available functions, including reverse, cumulative sum, frequencies, permutation and tail.

Each box also has a note field, an information (i) button that shows its dependencies and dependents and the values of inputs and outputs, and a clear (X) button. The reference name is shown next to the result. Tapping on the box type (Number Operation) opens up descriptions and examples of use of the functions associated with it.

Other box types cover list generation (e.g. series, random numbers, dates and number repetitions), list operations (e.g. item at given position and sublist sum), list/list operations (e.g. union and diff), findall (e.g. all numbers > 3 on the list or their positions), count (e.g. number of items < 3), If-Then-Else expressions (various types including the one already mentioned), and table functions and operations, including shifts, rotations, flips, permutations, cluster analysis, row, column, block or cell values, lookups, searches (e.g. rows where column $2 > 0$), and table changes, such as deleting rows, adding rows and changing values of the table - particularly useful for graphics.

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 (identical to those for Number Operation: add, subtract, etc.) on corresponding items in two lists to give the resulting list, as shown in the third box at the top of Figure 5. 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 last box at the top of Figure 5.

This item-by-item approach is used elsewhere in Numipulator, including with some If-Then-Else expressions, so that comparisons of the first two input lists are done an item at a time, and the output list is constructed as appropriate from the alternative items on the last two input lists.

4 Numbers and Lists: Syntax and Semantics

The syntax of most declarative or other programming languages is not easy to understand or remember for learners: there are so many parentheses, square brackets, arrows, commas and colons. For the boxes handled by Numipulator's Function Evaluator, the syntax can be kept simple due to their use of input fields and their drop-down lists of selectable functions. The language does not need commas to separate arguments or parentheses to group the arguments together: the fields and the boxes perform these roles.

A list input may include numbers, reference names that refer to either numbers or lists and tokens of either type: single number or list of numbers. So, if D1 and D2 are references to numbers with values 1 and 2 respectively and E1 and E2 are references to lists with values 3 4 and 5 6 respectively, then the list input specification D1 D2 has the list value 1 2, the list specification E1 E2 has the list value 3 4 5 6, while the list specification E1 E2 D2 has the list value 3 4 5 6 2, with no explicit concatenation functions required. Such mixing of number and list reference names in an input specification might seem unusal, but this is due to the semantics of a list in Numipulator: a list is an empty list (L0), a number, or a number followed by a list. This means that a list is 0 or more numbers, where these numbers follow each other. In the specification of a list, each list, reference name or token can be considered to be separated from the next by an implicit 'followed by' operator. Expressed informally, the value of a list with reference names in the specification is derived by placing the value of the reference in the list where it is found in the specification. Given this, if K4 is a list reference whose value is an empty list, the specification E1 K4 E2 D2 K4 has the list value 3 4 5 6 2, as no numbers representing K4 are placed into the list.

The two data types, number and list, intersect: a list with one number on it is just a number, so is identical to a number in Numipulator. This means that it is valid to specify a number input using a list reference name. At run-time, the system checks whether the list has exactly one number on it. If so, this number is taken to be the input. If it does not have exactly one number on it, a run-time error is thrown. This is important for the programmer to understand, as it means, overall, that any list output with a single number on it may be used anywhere in Numipulator as if it were a simple number. Numipulator therefore does not include alternative number-output versions of some list-output functions. For example, the If-Then-Else (Simple) box type has two number inputs (a and b) and two list inputs (lc and ld), and the drop-down functions are *if a < b: lc, else ld* and five others with different comparison operators. The result will be either lc or ld, depending on the comparison result. While lc and ld can contain lists with many or no items, they can also contain single-number lists, in which case the result can be used anywhere as though it were a number. Strictly, Numipulator does not need separate functions with number outputs, but some are included, such as Number Operations, as users would be familiar with these.

The six built-in tokens representing numbers or lists operate in the same way as reference names above. The number values are: *pi*, *e** (e - the base of natural logs), *click* (1 if a cell of the Graphics Formatter has been tapped, 0 otherwise) and *keypad* (1-9 depending on the Animation Zone keypad key tapped, 0 if none). The two list values are: *L0* (empty list) and *now* (8 numbers related to time/date at the start of the cycle, inc. year, month, date, hour, minute, and second).

5 Memory Boxes

Numipulator has no variables in which to store state from one cycle to another, so a different approach is required. In essence, the Numipulator processing engine modifies the inputs to certain types of box: Memory Number or Memory List. To understand these, the Memory Number shown on the left of Figure 6 (Y1) is best considered as shown on the right, with two implicit identity functions made explicit. Associated with it are, unusually, two values that can be evaluated in a cycle: (1) the principal value, Y1, whose input is the single input field, *mem*, and whose value is determined by the identity function; (2) a second value, newY1, whose input is the single input field, *next*, and whose value is determined by the identity function. Now, newY1 and *mem* are actually the same field, as shown. This means that the output newY1, specified by *next*, becomes the input, *mem*, to Y1 in the next cycle, so will become the next value of Y1. The system therefore

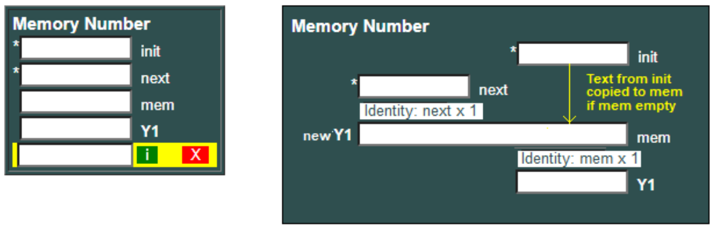


Fig. 6. Number Memory box with explanation

makes an output value from one cycle an input to the next cycle by modifying the input specification. State changes are therefore reflected in changes to the program (input specifications), made by the processing engine. Note: Only Y1 above may be used as an input to other boxes; newY1 may not. However, newY1 *will* be shown when displaying Values or showing the Flow, completely separate from Y1 (see Figure 4 with both Z1 and newZ1).

Only the system, not the programmer, can write to *mem*. It needs a specification for the first cycle, so the user must provide this in a separate field, *init*. If the processing engine finds that *mem* is empty, or the Start button is pressed, the text in this is copied from *init* into *mem*, to provide its initial specification.

6 Processing Engine

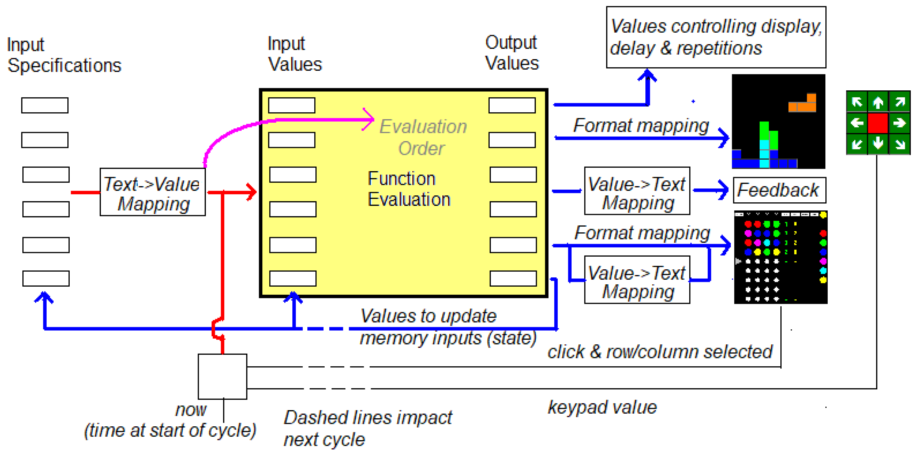


Fig. 7. Numipulator architecture including processing engine

The processing engine of Numipulator, its procedural component shown in Figure 7, is not complex. For an app started by the Start button, which continues repetitions until the termination condition succeeds, the simplified steps, ignoring syntax, circularity or run-time errors, are:

1. Any *init* fields specified in Memory boxes are copied into the *mem* fields; *init* fields for the Row/Column values of the Graphics Formatter are also copied into their associated fields.
2. All boxes with text in their inputs are analysed to yield the numbers contained within them, and any other significant tokens. Text->number input mapping is applied where needed.
3. The dependencies of boxes are determined, and an appropriate bottom-up evaluation order is determined.
4. *Start of an evaluation cycle*: The values of *now*, *click* and *keypad* are determined, and the last two are reset to 0.
5. The values of all inputs and outputs are determined by the Function Evaluator, based on the evaluation order determined. For each Memory box, the *next* value output replaces the *mem* input specification.
6. The termination condition is checked to see whether more cycles are required. If so, step 7 is followed; if not, step 8.
7. *Repetition required*: The Animation Zone, if used, is displayed. If Control2 (display) is defined and has a non-zero value, the Graphics Formatter display is updated, otherwise it is left unchanged. User taps are accepted until the end of the evaluated delay time (Control1). Step 4 is then followed again.
8. *After the final cycle*: All results are output and the Graphics Formatter and Animation Zone, if used, are displayed based on the final results.

No changes can be made to input specifications during a set of repetitions except those made by the system to the Memory boxes and Row/Column inputs through user interaction with the Graphics Formatter. The Evaluation Order remains constant throughout the repetitions.

7 Saving, Loading and Standalone Apps

Numipulator is a single HTML/CSS/Javascript file. As such, it cannot save a file to a system folder. The current program is, however, automatically saved to local storage when the Numipulator page is closed, and reloaded when the package is reopened. For longer term storage, the standard way of saving a program is to press a button to create a text version of the program and settings wrapped up in loadable form, and to copy and save this into a text file. The program can be reloaded by pasting the text file contents into a field and pressing a load button. The contents of the text file may also be pasted into a field in the Numipulator Standalone Maker, which creates code for a standalone browser app, with the graphical interface visible but reformatted and most of the program hidden. The code generated can be saved (.html), sent to others or put onto a web site.

Numipulator has over 30 built-in apps that can be loaded, including over 20 games and puzzles (one using only 10 boxes). Apart from being helpful or fun as products, their programs can be inspected by the user as examples of

Numipulator programming. Research by Tan et al[7] to determine the learning methods that are preferred by university students concluded that the students rated learning by referring to programming examples the highest.

8 Issues

One issue found with Numipulator is redundant evaluation in apps with many phases. Triple Jewels (Figure 2) is a game in which the player selects two adjacent jewels in a grid to be swapped with an aim of forming lines of 3 or more jewels of the same type, which then disappear to be replaced by ones falling from above. For this, 9 different phases were identified, including filling, checking adjacency, swapping, checking for triples, and removing jewels. Each phase may require different functions to generate outputs for the display and state. The functions required to generate a table with jewels added are quite different from those to generate one with jewels being removed or swapped. The program must include them all, and, importantly, all sets of functions will be evaluated at each cycle, regardless of the phase, because of the strict bottom-up evaluation employed by Numipulator. If-Then-Else boxes, or others, are used to select which outputs to use for each phase. As a simple example of this issue, in the Memory Pairs puzzle (Figure 2), a new 36-tile layout is generated in each cycle, even though this is used only at the start of the game. This is clearly inefficient, but is a feature of evaluating the same complete set of functions at each cycle.

The lack of ability to write and call new functions can be frustrating when several linked built-in functions are required to determine a particular output, and those same linked functions are required for some other output as well. The programmer has two choices in this case: repeat the sequence using two or more sets of boxes, or make use of repetitions in some way. For the Poker Squares game (Figure 2), the scores for each of the 10 lines are evaluated consecutively in 10 cycles. However, during the development of over 30 apps, this issue did not arise many times; the fact that the built-in list and table operations allow any number of sets of parameters to be applied to the input, with the results appended to each other, helped greatly. The powerful table functions, which include sorting and summing individual rows, proved invaluable when programming SudokuAssistant. This app can show the user which numbers are valid in an unfilled cell (e.g. 126 means 1, 2 and 6 are valid); these functions, using 81-column tables, enable full analysis and display to be carried out in one cycle only.

Although the programmer can employ tens of thousands of symbols and emojis in graphical cells, the graphics available with Numipulator are still limited. Programming a game in Numipulator can be seen as akin to painting by numbers or building a model with Lego; although the end product might lack finesse or have some rough edges, it will still be very usable, and the process of producing it should be enjoyable.

9 Conclusion

A number of its features make Numipulator stand out: its very simple syntax, enabled by its form-based interface; its integration of three types of declarative programming (element reference programming, text to number input mapping and number to text output mapping); its declarative generation of graphical displays; its dynamic declarative control capabilities and its potential for use in teaching. Overall, though, any fully declarative programming environment that allows the programmer to develop dynamic, interactive, graphical games similar to Space Invaders, Tetris, Bejeweled, Yahtzee, Connect 4, Wordle and Rummy is a valuable addition to the declarative programming showcase.

References

1. Jenkins, T.: On the difficulty of learning to program. In: 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences, 4, pp. 53–58. (2002)
2. Bosse, Y., Gerosa, M. A.: Why is programming so difficult to learn?: Patterns of Difficulties Related to Programming Learning Mid-Stage. In: ACM SIGSOFT Software Engineering Notes, 41(6), pp. 1-6. (2017). doi:10.1145/3011286.3011301
3. Cheah, C.S.: Factors Contributing to the Difficulties in Teaching and Learning of Computer Programming: A Literature Review. In: Contemporary Educational Technology, 12(2), ep272 (2020). https://doi.org/10.30935/cedtech/8247
4. Tisza, G., Markopoulos, P.: Understanding the role of fun in learning to code. In: International Journal of Child-Computer Interaction, 28(2), (2020). doi:10.1016/j.ijcci.2021.100270
5. Kay, A.: Computer software. In: Scientific American, 251, 3 (Sep. 1984), pp. 52–59. (1984)
6. Burnett M., Atwood, J., Djang,R., Reichwein, J., Gottfried, H., Yang, S.: Forms/3: A First-Order Visual Language to Explore the Boundaries of the Spreadsheet Paradigm. In: Journal of Functional Programming, 11(2), pp. 155-206. (2001). https://doi.org/10.1017/S0956796800003828
7. Tan, P.-H., Ting, C.-Y., Ling, S.-W.: Learning Difficulties in Programming Courses: Undergraduates' Perspective and Perception. In: International Conference on Computer Technology and Development, Kota Kinabalu, Malaysia, pp. 42-46. (2009). doi:10.1109/ICCTD.2009.188