The MATL programming language

Luis Mendo

December 14, 2015

1 Introduction

MATL is a programming language based on MATLAB and suitable for code golf.

The MATL language is stack-oriented. Data are pushed onto and popped out of a stack. Functions may take a number of elements from the stack (usually those at the top) and push one or more outputs onto the stack. Since the functions use data that's already present in the stack, reverse Polish notation (or postfix notation) is used.

This in line with other code-golfing languages, and allows more compact syntax for calling functions on data. To ease stack handling, values from the stack can also be copied and pasted using several clipboards. These are similar to variables in other stack-based code-golf programming languages.

The main goal in designing the language has been to keep it as close to MATLAB as possible. Think of it as MATLAB shorthand with data on a stack. MATL includes most commonly used MATLAB functions. It should be very easy for a MATLAB user to start programming in MATL within minutes.

1.1 The name

"MATL" /ˈmæt.l̩/ is pronounced to rhyme with "cattle".

The spelling is purposely *short* (few letters) and *unusual* (words ending in "-tl" are rare in English). This is intended to reflect the facts that

- the language has been designed for code golfing (*short* programs); and
- doing so requires some features that make it an esoteric (*unusual*) language.

The name is written in capitals to follow MATLAB's official name, which is also all capitals. (Besides, that way it looks more important!)

1.2 Notation

This is how MATL code is displayed in this document: `iXyD`. Or sometimes with spaces for greater clarity: `i Xy D`.

MATLAB code is displayed like this: `n = input(''); disp(eye(n)))`.

> Auxiliary or "meta" stuff is typeset this way. This includes discussions as to why something has been done in a particular way, or things that remain to be done.

1.3 Examples

Here are some simple example programs, just to give an idea of MATL.

- Infinite loop that does nothing:

  MATL: `` `T] ``

  MATLAB: `while 1,end`

- Get unique characters from an input string, maintaining their order:

  MATL: `j1X0H$u`

  MATLAB: `unique(input('','s'),'stable')`

- Same as above but done manually, that is, without using MATLAB's `unique` or its MATL equivalent `u`:

  MATL: `jtt!=XRa~)`

  MATLAB:
  ```
  s=input('','s');s(~any(triu(bsxfun(@eq,s,s'),1)));
  ```

- First 10 Fibonacci numbers:

  MATL: `1t8:"2$t+]`

  MATLAB:
  ```
  x=[0 1];for n=1:10,disp(x(2)),x=[x(2) sum(x)]; end
  ```

2 The stack. Data types

The stack can be thought of as a vertical arrangement of elements. Those elements may be popped from the stack, pushed onto the top of the stack, or rearranged. Functions take stack elements as inputs, and push new elements as outputs.

Any element contained in the stack will be an *array* of one of the following types:

- Numerical arrays;
- Logical arrays;
- Character arrays;
- Cell arrays.

Either of these arrays can be 2D or multidimensional. As in MATLAB, arrays of one dimension are considered to be 2D arrays with size 1 in one of the dimensions. Numbers are considered 2D numerical arrays of size 1×1.

2D numerical arrays are full or sparse. Their default class is `double`. MATL can work with all the other numerical classes. Conversion is done with a function corresponding to MATLAB's `cast`.

3 Statements and separators

A MATL program is divided into **statements**, **separators** and possibly **comments**.

A **statement** can be

- a literal;
- a function; or
- a control flow modifier (loops, conditional branches).

Literals can have a varying number of characters depending on their content. Functions and control flow modifiers consist of either

- one character (different from `X`, `Y` or `Z`); or
- two characters, the first of which is `X`, `Y` or `Z`.

Separators are sometimes needed to separate literals of the same type. For example, `12` means the number 12, whereas `1,2` means number 1 followed by number 2. Similarly, `'Padmé','Anakin'` means string `'Padmé'` followed by string `'Anakin'`, whereas `'Padmé''Anakin'` would be interpreted as a single string containing an apostrophe between the two names (quotation marks within strings are duplicated, as in MATLAB).

Separators are not needed if the literals are of different type, such as numerical and logical: for example: `T2` represents a MATLAB literal `T` (corresponding to `true` in MATLAB) followed by literal `2`.

The character `,` (comma) is the standard separator. Blank spaces and new lines also serve as separators. For clarity (but not for code-golfing purposes!), it may be convenient to display the program with spaces or line breaks inserted between statements.

Using several lines also allows inserting **comments** in MATL code. The convention is similar to MATLAB: `%` marks the start of a comment, and then everything until the end of the current line is considered to be a comment (that is, ignored).

4 Literals

Literals can be

- Numbers (equivalent to 1×1 numerical arrays)
- Numerical arrays, 2D (numerical vectors and matrices)
- Logical arrays, 2D (logical vectors and logical matrices)
- Character arrays, 2D (strings or 2D character arrays)
- Cell arrays, 2D

The effect of a literal statement is to push the element represented by that literal onto the top of the stack.

Literals correspond to one of the data types referred to in §2, but limited to 2D. It's not possible to define a multidimensional array directly as a literal (this limitation exists in MATLAB too). But it is possible, for example, to produce a 3D numerical array by concatenating several 2D arrays along the third dimension.

At any given time, the stack contents will consist of either literals or outputs produced by functions.

4.1 Numbers

Standard MATLAB formats are allowed, except that only `j` (not `i`) can be used for the imaginary unit, and only `e` (not `E`) can be used for exponents in scientific notation. Examples of valid number literals are `1.2`, `-.2j`, `1.`, `-1.2e3`, `+.25e-3j`.

Characters used in number literals "stick" to each other. Therefore consecutive number literals may need a separator to avoid confusion. For example, `12` represents number 12 (`1` sticks to `2`), whereas `1,2` represents numbers 1 and 2. Similarly, `1+2` represents numbers 1 and +2 (`+` sticks to `2`), whereas `1+,2` represents number 1, function `+` and then number 2. On the other hand, `5.6e-7j24` is interpreted as number $5.6 \cdot 10^{-7} j$ followed by 24, even if no separator is included, because the first number literal is already complete when the `j` character is processed.

An exception to numerical characters sticking is that characters `X`, `Y` or `Z` always grab the following character to form a two-character statement. For example `X123` represents function `X1` followed by number literal `23`.

`j` is interpreted as imaginary unit when used as part of literals, and as a function elsewhere. Again, a separator may be needed to avoid function `j` being interpreted as part of a literal. For example, `4j` is a literal representing number $4j$, whereas `4,j` is a literal representing number 4 followed by function `j`. To get number $j = \sqrt{-1}$ use the number literal `1j` (or the array `[j]`; see §4.2).

Complex numbers in general can't be introduced directly as literals; only real or imaginary numbers can. For example, `1+2j` would be interpreted as literal `1` followed by literal `+2j`. (However, array literals allow infix operators, as discussed in §4.2; so `[1+2j]` would be interpreted as the complex number $1+2j$.)

4.2 Numerical arrays

Notation is the same as in MATLAB: `[1 2 3]`, `[1,2,3]`, `[1,2j;3e4,-.2e-5]`, `[]`.

Colon notation can also be used: `[1:4;3 7 5 8]`, `[1:2:5;0 0 -1]`. For row vectors in colon notation the square brackets symbols may be omitted: `.5:.5:10`. In that case, a separator may be needed depending on what comes next.

The described infix colon notation can only be used in array literals. The colon symbol can also denote a function with similar meaning; but as happens for all MATL functions, it takes its inputs from the stack, and thus uses postfix notation. A separator is sometimes needed to distinguish which use of the symbol `:` is meant. For example, `5:8` is a numerical array literal as described above, whereas `5:,8` (or `5,:8`) is the `:` function applied to number literal `5`, followed by number literal `8`.

`j` can be used to represent the imaginary unit within array literals. Thus `[0 j]` and `[0 1j]` are equivalent.

Operators can be used within MATL literals as in MATLAB, and they are interpreted as infix. For example, the literal `[1/2 1+1/4]` would define an array with the two numbers 0.5 and 1.25. Numerical arrays can also contain `P`, `Y`, `N`, which correspond to `pi`, `inf` and `NaN` respectively. These can only be used *within array literals* (but there are MATL functions to obtain those values outside array literals). Also, `M` and `G` can be used within arrays as shorthand for `-1` and `-1j` respectively. For example, `[1,2,j;Y,N,G]` corresponds to `[1,2,j;inf,NaN,-j]`.

Only arithmetic operators, numbers, as well as `P`, `Y`, `N`, `M`, `G` can appear within literals. Arrays cannot contain calls to MATLAB functions (for example, `[cos(2); sqrt(3)]` is not allowed).

The above is also valid for array literals entered from keyboard by means of the `i` (`input`) function, and for strings used as inputs to `Yt` (`str2num`).

Numerical array literals are full (as opposed to sparse). Although 2D sparse

Table 1: Characters that have special meaning within number literals or array literals. Italic text indicates meaning is the same as in MATLAB

Character	Numerical literals	`[ ]` or `{ }` arrays
`T`, `F`	—	Logical values “true”, “false”
`Y`	—	Infinity
`N`	—	Not a number
`M`, `P`, `G`	—	Numbers -1, π, $-\sqrt{-1}$
`j`	In imaginary numbers	*In complex numbers; number* $\sqrt{-1}$
`e`	*Exponent*	*Exponent*
`( )`	—	*Grouping*
`& + - < > ^ |`	—	*Operators (infix)*
`.`	*Decimal point*	*Decimal point; make element-wise*
`*`, `/`	—	*Matrix operations (infix)*
`\`	—	*Matrix operation (infix)*
`:`	—	*Colon operator*
`;`	—	*End row*
`=`	—	*In relational operators*
`[ ]`	—	*Array builders*
`]`	—	*Grouping arrays*
`{ }`	—	*Cell array buliders*
`~`	—	*In relational operator; “not” operator*

numerical arrays are supported, they cannot be introduced directly as literals.

Table 1 contains all characters that have *special meaning within number literals or array literals*. Note that outside array literals they have a different, possibly unrelated meaning.

> It’s important to define carefully what is permitted within arrays. For example: should these be allowed? `[sqrt(1:3)]`, `[cos(1) sin(1)]`, `[path]`. Allowing arbitrary MATLAB code within literals (that will be evaluated with `eval`) brings flexibility, but it’s dangerous. For example, the literal `[rmpath('C:\path\to\files')]` would remove a folder from the path in MATLAB.
>
> One possibility to prevent that unwanted behaviour is to check the array contents strictly: a numerical array is `[...]` with contents separated by commas/spaces and semicolons. Each content must be a number, a numerical array, or a character array (which will be interpreted as numbers); and content sizes must much to form a 2D array. That involves hard work when parsing the array. It’s much easier to let MATLAB do that work (via the compiled code), which it does very well.
>
> In fact, this problem already exists with MATLAB’s `input` function: it *evaluates* what the user types, which is a dangerous thing to do. If the user

types `[cos(1) sin(2]` as input, evaluating that is probably fine. But what if they type `addpath('c:\path\to\folder')`?

Possible solutions have been discussed in Stack Overflow[1]. Thanks to a suggestion by `@pragmatist1`, the best approach seems to be: use regular expressions to detect function or variable names, but ignoring such names if they are within of a string (for example, `['path' 115]` should be allowed, and would be equivalent to `'paths`). The key here is that the regular expression doesn't need to check if the array is well formed (for example, `[[1 2; 3 4], [5 6]]` is not well formed), because that can be done later by evaluating it in MATLAB, once evaluation has been deemed safe. The regular expression only needs to make sure the array doesn't contain any reference to variables or functions, excluding string contents (a string can safely contain anything).

The above applies also to checking the contents of cell arrays.

The input obtained from `i` (`input`) is also checked this way before being evaluated, for safety. However, MATLAB does not do this. Do you see any reason why input checking should be removed?

4.3 Logical arrays

`T` and `F` correspond to `true` and `false` respectively, and can be used for defining 2D arrays in MATLAB. So `[T F T; F F T]` or `[T,F,T;F,F,T]` define a logical 2D array.

For logical row vectors, the notation `[T F T]` or `[T,F,T]` can be simplified to `TFT`. A separator may be needed if a new logical array follows: `TFT,TT`. But it's not necessary in other cases: `TFT3.5`.

4.4 Character arrays

They are defined as in MATLAB. Quotation marks within strings need to be duplicated. Two example character arrays are `'I''m sorry, Dave'` (a row character array, or string) and `['Deckard'; 'Rachael']` (a 2D character array).

Numbers or number arrays can be used as components of character arrays; and then they are interpreted as ASCII codes of characters (as in MATLAB). For example, `['My food ' [105 115] ' problematic']` is the same as `'My food is problematic'`.

Strings (row character arrays) can also be defined using colon notation: for example, `'d':'j'` is equivalent to `'defghij'`, and `'a':4:'z'` is equivalent to `'aeimquy'`. As in MATLAB, the first and last operands must be of type char.

[1] http://stackoverflow.com/q/33124078/2586922

4.5 Cell arrays

They are defined as in MATLAB. Each cell can contain any of the preceding literals, or other cell arrays. An example cell array literal is `{'Great-Crack', 'But seeds are not pods!', {1,2,3,'travel','swift','petal'}}`.

5 Functions

Functions operate on the contents of the stack, and produce new contents that are popped onto the stack. Function outputs can be of any of the types introduced in §2.

Functions take their input data in the same order in which they are in the stack, that is, bottom to top. So for example `5,4/` gives the result 1.25. This applies even if the input data are not taken from the top. Consider for example a two-input function. As will be seen in §5.2, this could take its inputs from arbitrary positions in the stack, say from the top and three positions below. In that case, the element that is *lowest* in the stack will be the *first* input to the function.

Outputs are pushed to the stack in the order in which they are returned by the function. That is, if the function returns two outputs, the first output will be pushed first; and then the second, which will be left on top.

There are four types of functions:

- **Normal functions** take their inputs from the stack and push their outputs onto the stack. An important particular class of normal functions is that of **indexing functions**, which will be dealt with separately.
- **Meta-functions** take their inputs from the stack, and produce no output on the stack. They are used for modifying the behaviour of normal functions; specifically, for defining their numbers of inputs and outputs.
- **Stack rearranging functions** take a group of elements from the stack and rearrange them.
- **Clipboard functions** give access to the clipboards, which store values from the stack and push them back. They behave like variables.

Inputs used by normal functions and meta-functions are always consumed, that is, disappear from the stack. For stack rearranging functions or clipboard functions that's not necessarily the case.

5.1 Normal functions

Normal functions operate on stack inputs and produce outputs that are pushed onto the stack. A given function may accept a variable number of inputs and produce a variable number of outputs.

The full list of normal functions is given in §8, and their detailed definitions appear in Appendix §A. Most MATL functions have a MATLAB equivalent. For example, MATL's `u` corresponds to MATLAB's `unique`. In MATL, each function has a minimum and maximum numbers of allowed inputs and outputs, usually the same as in MATLAB; but in addition *default numbers of inputs and outputs* are specified. Continuing with the example, `u` by default takes one input and produces one output. So if the top of the stack contains `[1 1 2 3 2]`, the function `u` would produce `[1 2 3]`. The meta-functions `$` and `#` (see §5.2) can be used to change the number of inputs and outputs of a function.

In some cases there exists a variation of the MATL function that corresponds to another one with a certain argument input fixed. For example, MATL's `Xu` corresponds to MATLAB's `unique(..., 'rows', ...)`.

MATL functions take inputs of the same data types as their corresponding MATLAB function. For example, in MATLAB numerical arrays can be added with character arrays or with logical arrays, and the result is a numerical array. Similarly, the MATL `+` function accepts these types of inputs and produces a numerical array as output. Thus `T1+D` would display a result 2 on screen (`D` by default displays the top of the stack), and `T'a'T++D` would display `99` (the character code for `'a'` is 97).

Some normal functions simply push fixed, predefined literals onto the stack depending on their input. These are functions `X0`,...,`Z9`. Also, clipboards initially contain certain predefined values. The former are used mainly for strings (such as `'stable'` or `'@mean'`), whereas the latter are used for numeric values (such as `2` or `[0 -1 1]`). These literals are used often, and calling the corresponding function requires less characters than actually typing the literal.

Thanks to `@rayryeng` for the suggestion to include predefined constants[2].
Also for reading an early version of this document and suggesting changes.

5.2 Meta-functions

Meta-functions are `$` and `#`. They are used for specifying, respectively, the number of inputs and outputs that will be used by the next function, whether it's a normal, stack rearranging or clipboard function. *The input/output specifications done with meta-functions* `$` *and* `#` *are consumed (deleted)* by the next *normal, stack rearranging or clipboard function*. If that function doesn't allow the specified number of inputs or outputs an error occurs. If a `$` or `#` specification is issued before the previous one has been consumed by a function, the new specification replaces the previous one.

The `$` meta-function **specifies the inputs** to be used by the next function. `$` takes as argument the top element of the stack, and consumes it. That element can

[2] http://chat.stackoverflow.com/transcript/message/26114065#26114065

be a *number* such as `3`, or a *logical array* such as `TTF`. In the first case, the next function will use the three highest elements on the stack as inputs. In the second case, the logical values are interpreted as a logical index into the stack elements, starting from the top. So `TTF$` indicates that the function will use as inputs the two elements below the top element.

If the `$` specification is a logical array which is not a vector, it is interpreted in *column-major* order, that is, it's implicitly linearized. Also, *leading* `F` values are ignored. Thus, both `FTFT$` and `[F,F,F;F,T,T]$` would be equivalent to `TFT$`. This is in line with how MATLAB treats logical indices: they are automatically linearized and trailing `false` values are ignored. Note that here it's leading (not trailing) values that are ignored, because logical indexing into the stack is based on its top (not on its bottom): given $\cdots$`TF`, the rightmost value (`F`) refers to the top; the value to its left (`T`) refers to the second highest element in the stack, etc.

Consider as an example the `f` function, which corresponds to MATLAB's `find`. The code `[0,4,7,0]f` would produce the output `[2,3]`, just as `find([0 4 7 0)` in MATLAB. `[0,4,7,0]1,2$f` would produce 2, corresponding to the two-input MATLAB call `find([0 4 7 0], 1)`. The same result could be obtained with `[0,4,7,0]1TT$f`; and also with `[0,4,7,0]1FFFTT$f`.

For an example with non-contiguous inputs, assume the stack contains, bottom to top, `[0,4,7,0]`, `'abc'`, `1`. Then after executing `TFT$f` it will contain `'abc'`, `2`.

The `#` meta-function **specifies the outputs** to be produced by the following function. `#` takes as input either a number or a logical array. If it's a number *n*, it indicates that the function will be called with *n* outputs. If it's a logical array, it indicates which of the possible outputs will be asked for from that function. For example, if the stack contains `[0,4,7,0]`, `1` bottom to top, the code `2$FTT#f` will produce `4`, `2`, corresponding to `[~, col, val] = find([0 4 7 0], 1)` in MATLAB.

Again, logical arrays are interpreted in *column major order* when passed to `#`. However, in this case `F` values are *not ignored*. This is consistent with MATLAB's behaviour: `[ii,~] = find([0 4 7 0])` and `ii = find([0 4 7 0])` produce *different* results. In general, functions behave differently depending on the number of outputs with which they are called, regardless of whether some of those outputs will be ignored; and this is true even for ignored *trailing* outputs. Another example es the `size` function: `s = size(eye(3))` produces `[3 3]`, whereas `[s,~] = size(eye(3))` produces `3`. These would be done in MATL as `3Xyy` and `3XyTF#y` (`Xy` is `eye`, and `y` is `size`, which is called with one output argument by default).

> This may be a little surprising. One tends to think that adding outputs to a function call won't affect the preceding outputs. But in some cases it does. So saying "Use the first output of `find` to obtain the result" is not strictly correct, because it's ambiguous: "Which first output? The one I get when calling the function with two outputs? With one output? . . . ?".

Note that, even if the function inputs can be taken from arbitrary positions of the stack (using `$` with a logical array), the outputs are always pushed onto the top of the stack.

When MATL processes a function, if no `$` or `#` specifications exist (for example, because they have been consumed by a previous function call), the default number of inputs or outputs is used for the current function. If `$` or `#` specifications are actually issued, they don't need to be right before the function. For example, there can be literals in between (but not another function, as it would consume those specifications). So, if the stack contains `[0,4,7,0]`, `1`, after executing `TTF$'abc'FTT#f` it will contain `'abc'`, `4`, `2`.

To delete the specifications currently held by `$` or `#`, and thus have the next function apply the default values, use `[]$` or `[]#` respectively.

> Another possibility would have been to define the language so that the number argument to `$` and `#` was interpreted as a bitmap indicating which inputs (and how many) are used. In other words, the number would be binary decoded and the result interpreted as a logical array. So `7$` in this alternative definition would correspond to `3$` or `TTT$` in the above definition: use as inputs the three highest elements, thus 111 in binary, which is number 7.
>
> Pros of this alternative approach: most functions have three inputs/outputs or less, and this would thus result in more compact code (`TFT$` would become `5$`). Cons: (1) If a very large number of inputs were used the argument to `$` would become long. For example, MATLAB's `cat` often accepts a lot of inputs (maybe from a comma-separated list). 20 inputs would require number $2^{20} - 1 = 1048575$ as argument to `$`. (2) Less natural semantics, specially for some stack rearranging functions.
>
> So I think it's best to interpret integer numbers as number of inputs, and use logical arrays for bitmaps, as has been defined above. If desired, a number can be transformed into a logical vector corresponding to its binary expansion very easily (there's a function for that).

5.3 Stack rearranging functions

These functions rearrange the elements in the stack (by duplicating, moving, deleting, copying or pasting). They don't strictly have outputs; rather, they operate on the stack elements directly. These functions have no outputs, so any `#` specification other than `0#` will give an error (and specifying `0#` is unnecessary).

- `x` (delete). It can have an arbitrary number $n = 0, 1, 2, \dots$ or a logical array as `$` specification; default is 1. It deletes the n top elements from the stack; or the elements specified by the logical array. For example, `TTF$x` deletes the two elements below the top of the stack.

- `w` (swap). It can have an arbitrary number $n = 0, 1, 2, \dots$ specified to `$`; by default 2. For $n > 0$ it swaps elements 1 (top) and n in the stack, leaving those in between intact. For $n = 0$ it does nothing. If the `$` specification is a logical array, the first and last indicated elements are swapped.

- `t` (duplicate). It can have an arbitrary number $n = 0, 1, 2, \ldots$ specified to `$`; by default 1. It copies the top n elements from the stack, and pastes them (without deleting them from their original positions) at the top of the stack, maintaining their order. For example, if the stack contains `'a'`, `'b'`, after `2$t` the stack will contain `'a'`, `'b'`, `'a'`, `'b'`.

  If a logical array is used as argument to `$`, it specifies which elements should be copied, maintaining their relative order. For example, `TTF$t` would copy the third- and second-highest elements and paste them at the top of the stack.

- `b` (bubble up). It can have an arbitrary number $n = 0, 1, 2, \ldots$ specified to `$`; by default 3. It makes the n-th highest element "bubble up" to the top, shifting elements $1, \ldots, n-1$ one position down. For $n = 0$ it does nothing.

  If a logical array is used as argument to `$`, the bubbling is done only for the indicated stack elements. For example, if the stack initially contains `'a'`, `'b'`, `'c'`, `'d'` from bottom to top, `TTFT$b` would leave the stack as `'b'`, `'d'`, `'c'`, `'a'`.

- `N` (stack size). It pushes the number of elements that the stack currently has. It has no inputs (so the only allowed input specification would be `0$`, which is unnecessary anyway).

5.4 Clipboard functions

There are five clipboards (or arbitrarily many, depending on interpretation; more about that below). They are identified with the capital letters "H", ..., "L". Clipboards are used for copying from and pasting to the stack.

The first four clipboards behave as would be expected: they store values that can be retrieved later. Clipboard L is special: it consists of an arbitrarily large amount of "levels", and each level (beginning at 1) behaves as an independent clipboard. Thus clipboard L is actually an "infinite" collection of clipboards. Compared with clipboards H, ..., K, copying and pasting with clipboard L requires at least one extra character (to specify level), as will be seen.

Access to the clipboards is provided by means of the following functions:

- `XH`, `XI`, `XJ`, `XK` (copy to specified clipboard): copy a number of elements from the stack to one of the clipboards, without removing them from the stack. These functions have an arbitrary number of inputs $0, 1, 2 \ldots$, and no outputs. How many or which elements should be copied to the given clipboard is specified with `$`. For example, `2$XH` indicates copy the two top elements to clipboard H; and `TF$XH` indicates copy the element right below the top. By default the top element is copied, corresponding to `1$`.

Copying elements to a clipboard removes its previous contents.

If the argument to `$` is 0, or a logical array containing only `F` values, the copying functions clear the clipboard contents.

- `H`, `I`, `J`, `K` (paste from specified clipboard): paste copied elements onto the stack, in the same relative order in which they originally were. The argument to `#` indicates which elements should be pasted, among those that the clipboard currently contains. By default all contents of the clipboard are pasted. These functions have no inputs.

- `XL` (copy to some level of clipboard L): behaves like `H`, but takes one last additional input that specifies the level used within clipboard L. The default number of inputs is 2. So `1XL` (or `1,2$XL`, or `2$1XL`) copies the top of the stack to clipboard L, level 1. The input that specifies the level is automatically converted to `double`. So `TXL` would do the same thing.

  As usual, a logical array can also be used as argument `$`. If the stack contains from bottom to top `'a'`, `'b'`, `'c'`, the code `TTFT$4XL` copies `'a'`, `'b'` to clipboard L4 (note that the last `T` refers to `4`, which specifies the level).

- `L` (paste from some level of clipboard L): behaves like `XH`, but takes as input one number that specifies the level used within clipboard L.

As can be seen from the above, clipboards H, I, J, K provide easier access (fewer characters) than clipboard L. In many cases four clipboards will be enough. On the other hand, clipboard L provides extended capabilities if needed, at the cost of at least one extra character to specify the level.

Clipboards H, I, J and K, as well as the first levels of clipboard L, initially contain the values indicated in Table 2.

> Using for example `H` instead of `2` may be useful to avoid separators (commas or spaces) if that 2 is surrounded by other numbers. Values 0 and 1 can be obtained with functions `O` and `U` (which correspond to `zeros` and `ones` respectively; by default the take no inputs and produce a single number as output).

Clipboard L only has 12 levels initially (necessary to hold the initial contents specified in Table 2). Higher levels are created at run-time *on the fly*. For example, copying some elements to level 14 will cause MATL to create levels 13 and 14: level 13 will be empty, and level 14 will hold the copied elements. Trying to paste from level 15 gives an error, because that level doesn't exist; but level 13 now exists, and is empty. Therefore `13L` is valid code, and does nothing. `0#13L` is also valid, but `1#13L` (or even `F#13L`) is not, because a non-existent first element in level 13 is being referenced.

Table 2: Initial content of clipboards

Clipboard	Contents
H	`2`
I	`3`
J	`1j`
K	`4`
L, level 1	`[1 0]`
L, level 2	`[0 -1 1]`
L, level 3	`[1 2 0]`
L, level 4	`[2 2 0]`
L, level 5	`[1 -1j]`
L, level 6	`[2 0]`
L, level 7	`[1 -1j 0]`
L, level 8	`[1 3 2]`
L, level 9	`[3 1 2]`
L, level 10	`3600`
L, level 11	`86400`

Clipboard L effectively provides an infinite amount of "variables", but requires more characters compared with variables in other languages such as CJam. Clipboards H, I, J and K use less characters. Why only four such clipboards? The main reason is that MATLAB has a lot of "high-level" vectorized functions compared with other languages. It's desirable to have many of those functions in MATL, and that requires reserving a significant part of character space for them, at the expense of clipboards/variables. Also, characters should be reserved in order to be able to include new functions in the future.

Also, I have a feeling that true stack-oriented programming should use as few variables/clipboards as possible, and that four clipboards are enough in almost all situations. For problems when that's not the case, one can use clipboard L with a small penalty in number of characters.

Clipboard L starts at level 1, not 0, to match MATLAB indexing. This is important for debugging. The MATL debugger makes use of MATLAB's debugging capabilities to show relevant variables. Therefore the first level of clipboard L will be shown (by MATLAB) as 1, not as 0. A less important, more subtle advantage is that this makes the above mentioned implicit conversion to `double` unnecessary: characters are converted implicity by MATLAB, and `true` used as a logical index has the same effect as the integer index `1`; whereas `false` wouldn't have the desired effect of selecting the first (0-th) level.

5.5 Indices and indexing functions

Indexing with **round brackets** is used in MATLAB for accessing array's contents and cells of a cell array. In MATL the indexing functions `(` and `)` are used for that.

Indexing with **curly braces** in MATLAB is used for accessing cell's contents in a cell array. The MATL indexing functions `X(` and `X)` are used for that.

Like in MATLAB, indexing in MATL can be used for two purposes: **referencing** elements from an array, or **assigning** values to elements of an array. For example, in `a(2,:) = b(1,:)`, reference indexing is being used for `b`, and assignment indexing is being used for `a`.

The four basic indexing functions in MATL are thus:

- `)` round-bracket reference indexing;
- `(` round-bracket assignment indexing;
- `X)` curly-brace reference indexing;
- `X(` curly-brace assignment indexing.

Also, there are two additional functions that correspond to MATLAB's reference indexing with colon:

- `Y)` round-bracket reference indexing with colon;
- `Z)` curly-brace reference indexing with colon.

> I find it more natural to use a closing parenthesis `)` for reference indexing and an opening parenthesis `(` for assignment indexing. The latter suggests "opening" the array so it can accept new contents.

`)` takes an arbitrary number of inputs $2, 3, 4, \ldots$, by default 2. The first input is the array to be indexed, and the others are interpreted as indices into that array. For example, `a(2,[1 3 4])` would be done in MATL, assuming `a` is currently at the top of the stack, as `2[1,3,4]3$)`. The three inputs would be consumed, and the indexed array `a(2,[1 3 4])` would now occupy the top of the stack.

The indices can be integer-valued or logical. For example, `2[1,3,4]3$)` would be equivalent to `2TFTT3$)`. As in MATLAB, a logical index need not have the same size as the dimension it's indexing into. `a(2,[true false true true])` works even if `size(a,2)` is greater than 4. In general, trailing `false` (or `F`) values are ignored.

To allow `end`**-based indexing**, the following convention is used in MATL: whenever an array used as index has 3 or less elements and at least one of them has *zero real part* it is interpreted as and `end`-based index. An imaginary value such as `-2j` corresponds to `end-2`, and colons are implicitly assumed between the two or three values of the array. So, for example,

- `[2,-1j]` (or `[2,J]`; see §4.2) represents `2:end-1`.
- `[-.5j,2,0]` represents `end/2:2:end`.
- `[1,0]` represents `1:end`, that is `(:)`.

These `end`-based indices can be used in MATL as any other index. For example, `a(2,:)` would be realized in MATL, assuming the top of the stack contains `a`, as `2[1,0]3$)` (or `2,1L,3$)`, using the fact that `1L` by default produces the literal `[1,0]`; see Appendix §A).

> I had initially dropped the idea to include `end`-based indexing because it seemed complicated: MATLAB's `end` is used *within* an array, and the known size of that array determines its meaning. But MATL's stack-oriented character and absence of infix notation means that the `end`-based array needs to be pushed onto the stack by itself, and exist independently of whichever array it will end up indexing into.
>
> But I picked it again following a discussion with `@beaker`[3] (thanks also for pointing out some errata[4]). I came up with the idea that the imaginary unit can be used as an "`end` label", which is left unresolved until an indexing function such as `)` establishes its relationship with a specific array.
>
> Common `end`-based indexing expressions are predefined as initial contents of clipboard L; see Table 2).

`(` takes an arbitrary number of inputs, by default 3. The first input is the array to which values will be assigned; the second is an array containing the values to be assigned to the first; and the following inputs are the indices into the first array. Consider for example `a(5,6) = b`. In MATL, if the stack contains `a` and `b` on top, the corresponding assignment would be `5,6,4$)`. Or, combining this with reference indexing into the right-hand size, `a(5,[6 7]) = b(8,[9 10])` would be `8 [9,10] 3$( 5 [6,7] 4$)`. Note that the initial `8 [9,10] 3$(` replaces the top of the stack `b` by `b(8, [9 10])`.

The order of inputs used by `)`: "destination", "data", "indices" naturally adapts to processing in a loop: in each iteration some processing is done, resulting in the data to be assigned to a destination array that was previously pushed onto the stack. The index of the assignment is then popped onto the stack and `(` is called.

Of course, the inputs to `(` and `)` can be specified by logical arrays (`TTT$`) instead of numbers (`3$`).

`X)` is analogous to `)`, but corresponds to MATAB's `{...}` reference indexing. It takes an arbitrary number of inputs $2, 3, 4, \ldots$, by default 2. The first input should be a cell array, and the others are interpreted as indices into that cell array. The indices can be integer or logical values. The result of that indexing in MATLAB

[3] http://chat.stackoverflow.com/transcript/message/26114026#26114026

[4] http://chat.stackoverflow.com/transcript/message/26879794#26879794

would be a comma-separated list of the indexed cell's contents. In MATL, each element of that list is pushed onto the stack in the indexing order.

`X(` is similar to `(`, but corresponds to MATAB's `{...}` assignment indexing; and has some differences. In MATLAB, an assignment like `a{[1,2]} = ...` can be done in two ways:

- `[a{[1,2]}] = b{[2001,2010]}` (comma-separated list generated from a cell array on the right-hand side);

- `[a{[1,2]}] = deal('Follow the white rabbit', '101')` (`deal` on the right-hand side).

The first is actually equivalent to `(...)` indexing: it can be done as `a(...) = b(...)` or, more generally, `a(...) = reshape(b(...),...)`. The second makes more sense for MATL, is actually what `X(` does.

Considering two indices for greater generality, a typical assignment of this type might be, in MATLAB, `[a{[1,2],[3,4]}] = deal({'Mostly harmless', [],42,5})`. In MATL it's done as follows. `X(` takes as inputs, in this order: a destination cell array (corresponding to `a`), an arbitrary number of data (4 in this case), an arbitrary number of indices (2 in this case) that define the exact destination of those data within the cell array, and *a number that specifies how many indices exist*. So in this case 8 inputs should be specified for `X(`. If the stack contains, bottom to top: destination cell array, `'Mostly harmless'`, `[]`, `42`, `5`, `[1,2]`, `[3,4]`, `2`, then `8$X(` does the assignment.

The reason why the number of indices needs to be specified as an additional input is to avoid ambiguities with `end`-based indexing. Suppose the number of indices were not specified, and consider the following inputs to `X(`: destination cell array, `'Mostly harmless'`, `[]`, `42`, `5`, `[1,0]` (note that the latter is an index interpreted as `1:end`). The destination cell array, (`a`), initially has size $2 \times 1 \times 2$. In these conditions, there are *two* possible interpretations for the assignment: either `[a{1:end}] = deal({'Mostly harmless',[],42,5})` (assign four values, using linear indexing) or `[a{42,5,1:end}] = deal({'Mostly harmless',[]})` (assign two values along the third dimension).

> This ambiguity occurs only because of `end`-based indexing. With integer-valued or logical indices, interpreting one input as index instead of data would increase or maintain the number of indexed positions in the destination array, while at the same time would reduce the amount of available data, so there would only be (at most) one valid possibility.
>
> Having to specify the number of indices is not very nice, specially compared with `(`, where it's not needed. But it seems unavoidable. And after all, MATLAB also requires `deal` in this case, which is not needed in round-bracket assignments.

According to the above, the minimum number of inputs for `X(` would be 4: destination array, at least one data element, at least one index, and number of indices. The most common case is probably that of a single data element and a single index. To alleviate notation in this case, this can be realized with only 3 inputs, omitting last one specifying the number of indices. That is, when `X(` is used with 3 inputs they are interpreted as destination array, one data element and one index. This is the default number of inputs for `X(`. Any other number above 3 is also allowed, and in that case the last input tells the number of indices.

> What a mess! Couldn't this `X(` stuff have been made simpler?
>
> I would have liked to, but I haven't found a way to make curly-brace assignment simpler while keeping consistence with `$` semantics. MATLAB notation `[a{ind1,ind2}] = deal(data1, data2, data3, ...)` is not terribly simple to begin with; and having everything in a stack can sometimes obscure things a bit (although of course it has other advantages).
>
> In many cases the assignment will be to a single cell with a single index (`a{ind} = data`). In MATL that case is simple indeed, and totally analogous to round-bracket indexing: push `a`, then `val`, then `ind`, and `3$X(` will do the assignment. And in the more general case one only has to remember to include the number of indices as a final input.
>
> All in all, notation may get a little complicated in the general case. On the positive side, some people consider code obfuscation to be, well, a good thing... specially in a weird/esoteric language worth that name!
>
> On a more serious note, MATL indexing supports integer/logical indexing; multidimensional/linear/partially linear indexing; curly braces (cell contents) and round brackets (cells, standard arrays); reference (getting values from an array) and assignment (putting values into an array); all working on a stack. I hope notation is just about as complicated as it needed to be.

`Y)` takes an input array and applies MATLAB's `(:)` reference indexing. This has the effect of linearizing the array into a column. The number of inputs and outputs is 1. For example, `[1,2;3,4]Y)` would replace the input array `[1,2;3,4]` by `[1;3;2;4]`.

`Z)` takes one input, which must be a cell array, and applies MATLAB's `{:}` reference indexing. This has the effect of "unboxing" the cell array, that is, it produces its contents (in column-major order), which are pushed onto the stack. The output specification for `Z)` can be arbitrary, and controls how many or which of the input contents are pushed onto the stack. By default all contents are pushed. For example, `{{'Gödel' 'Escher' 'Bach'} 'Hofstadter' 1979} Z)` would consume the input cell array and push onto the stack, in this order: cell array `{'Gödel' 'Escher' 'Bach'}`, string `'Hofstadter'`, and number `1979`.

6 Control flow: conditional branches and loops

6.1 Conditional branches: "if...else"

An "if" branch is begun with `?`, ended by `]`, and can optionally contain `}`. These correspond to MATLAB's `if`, `end` and `else` respectively. `?` pops (consumes) the top array from the stack, and the statements until `}` or `]` are executed if the real part of that array has all nonzero elements (as in MATLAB). Otherwise the statements between `}` and `]`, if any, are executed.

6.2 Loops: "for", "do...while" and "while"

A "for" loop is begun by `"` (double quotation symbol) and ended by `]`. The statement `"` takes (consumes) the top array from the stack. That element array is split into pieces, namely the columns of the array (or in general second-dimension slices, like MATLAB's `for`); and those pieces are iterated on (loop variable). In each iteration, all statements until `]` are executed. The current piece of the array can be obtained with the statement `@`; see below. The loop normally ends when all pieces have been iterated on. However, statements `.` (conditional `break`) and `X.` (conditional `continue`) can modify this behaviour, as will be discussed later.

A "do...while" loop is begun by `` ` `` (back tick) and ended by `]`. A "do...while" loop is different from MATLAB's "while" loop in that the loop condition is evaluated at the end, not at the beginning. MATL initially executes all statements between `` ` `` and `]`. It then takes (consumes) the top array of the stack and checks if its real part has all nonzero elements (like MATLAB's `while` does). If so it goes back to the beginning of the loop; else it exits the loop. As an example, ``10`1-t]`` counts from `9` down to `0`. Statements `.` and `X.` can also be used in "do...while" loops. `@` in this type of loops pushes the current iteration index.

A "while" loop is begun by ``X` `` and ended by `]`. In this case the loop condition is evaluated at the beginning, as happens with MATLAB's `while` loops. Statements `.` and `X.` can also be used, and `@` pushes the current iteration index.

The statement `@` within a "for" loop pushes the current value of the loop variable, that is, the current part of the array that is being iterated on. Within a "do...while" or "while" loop it pushes the current iteration index, starting at 1. The output of `@` is that corresponding to the *innermost* loop.

The statement `.` takes (consumes) the top of the stack. If the real part of that array has all nonzero elements (as in MATLAB's `if`), the innermost "for", "do...while" or "while" loop is exited. Similarly, `X.` takes (consumes) the top of the stack, and if the real part of that array has all nonzero elements, the current iteration of the innermost loop is ended, and the loop proceeds with the next iteration, if any.

> I initially defined MATL with "'for" and 'while" loops only, as in MATLAB. But I realized that all "while" loops I was writing had a `T` before

` to force the loop code to be executed at least once. So I introduced "do...while" loops.

7 Initial and final implicit actions

When a MATL program is run, it begins with `format compact`, `format long`, `warning('off','all')`. The seed of the random number generator is set using `rng('shuffle')`.

If a file called `defin` exists when program execution is started, its contents are read in as bytes and converted to char type. A string (row vector of characters) containing all those values is pushed onto the stack before the first program statement is executed. (There is also a function for manually reading a file; see §8).

The display function `XD` is implicitly called at the end of the program. This function (see §8 and §A) by default takes each element from the stack, converts it to string and prints it on screen. Cell array inputs are unboxed and their contents are displayed in linear order. Nested cell arrays are allowed. For example, consider that the stack contains two elements: number `1` and the cell array `{'aa' {3 4; 5 6}}`. Then the displayed values will be, in this order: `1`, `aa`, `3`, `5`, `4`, `6`.

The number of inputs for the implicit display function can be modified by one final call to `$`. For example, to display only the top of the stack specify `1$` at the end of the program. To prevent the implicit display function from displaying any stack contents use `0$` (or empty the stack).

All screen output generated by the program is saved to a text file called `defout`. This file is overwritten if it previously existed. (There is also a function for manually writing to a file; see §8).

When the program finishes, the original MATLAB `warning` state is restored.

> Is there anything else that should be defined as initial conditions?
> Set `digits` to a large value such as 50 (for `vpa`)?

8 Table of functions

Table 3 shows the set of MATL functions, as well as control flow modifiers, separator and comment characters. The detailed definition of each function appears in Appendix §A.

> The table is under development. Functions may be added, removed, or changed.
>
> `e` and `j`, when used as a functions, may need a separator to isolate them from a preceding number.
>
> Similarly `:`, when used as a function, may need a separator to prevent it from being interpreted as part of a colon numerical vector.

Most MATL functions correspond to MATLAB functions. These have been selected based on perceived most common use. They include `@Divakar`'s list[5], and most functions from a list by Mathworks[6].

In some cases the MATLAB function has been extended or slightly modified. This is indicated in the function description of Appendix §A.

All MATLAB operators supported by `bsxfun` have automatic singleton expansion in MATL.

> Thanks to `@flawr`[7] and `@David`[8] for the suggestion of an "n-th prime function, and to `@David` for finding a good implementation

Most strings that are used as inputs to functions (for example, `'stable'` option for `unique`) can be produced by calling one of the functions `X0`...`Z9` with a numeric input. See §A for details.

Table 4 summarizes function behaviour regarding their consumption of inputs and of input/output specifications.

> Some functions I didn't include, and the reasons why:
>
> - `true`, `false` functions; they can be done with `zeros` and `ones` (which is included) followed by conversion to `logical` or by negation (each is just one character).
> - `ipermute`: I don't think it's used often; and can be substituted by `permute`.
> - `ndims`: it can be realized in two characters with `size` followed by `numel`.
> - `length`. I followed `@chappjc`'s advice[9]: "Never use `length`. Ever."
> - I initially thought it would be cool to have function equivalent to `eval` in MATL. But it would be hard to implement, and probably not very useful anyway.
> - `cellstr`: function `mat2cell(x, ones(size(x,1),1), size(x,2), ..., size(x,ndims(x)))` exists. It's a generalization of `cell2str` that also works for numeric and logical values, and for any number of dimensions.
> - `shiftdim`: it seems unnecessary with `permute`.
> - `clock`: it seems unnecessary when `now` is available.

[5] http://stackoverflow.com/users/3293881/divakar

[6] http://es.mathworks.com/help/matlab/functionlist.html

[7] http://chat.stackoverflow.com/transcript/message/27351795#27351795

[8] http://chat.stackoverflow.com/transcript/message/27352363#27352363

[9] https://web.archive.org/web/20150915161950/http://stackexchange.com/users/3302774/chappjc

Table 3: Table of MATL statements

		X	Y	Z
	separator			
!	.' (transpose)	rot90	system	full
"	for	repmat	repelem (run-length decoding)	
#	specify outputs		rng	fopen, fwrite, fclose
$	specify inputs		char(vpa(...))	fopen, fread, fclose
%	comment	class	cast	typecast
&	and	intersect		bitand
'	*Not used. String delimiter*		run-length encoding	now
(	round-bracket assignment indexing	curly-brace assignment indexing		
)	round-bracket reference indexing	curly-brace reference indexing	linearize array	comma-separated list
*	.*	kron	matrix product	Cartesian product
+	+	conv	conv2	convmtx
,	separator	cos	sin	tan
-	-	setdiff	deconv	
.	break	continue	pause	bitget
/	./	angle	matrix /	unwrap
0	*Not used*	predefined literals	predefined literals	
1	*Not used*	predefined literals	predefined literals	
2	*Not used*	predefined literals		
3	*Not used*	predefined literals		
4	*Not used*	predefined literals		
5	*Not used*	predefined literals		
6	*Not used*	predefined literals		
7	*Not used*	predefined literals		
8	*Not used*	predefined literals		
9	*Not used*	predefined literals		
:	colon (function)			bitset
;		acos	asin	atan2
<	<	min	cummin	
=	==	isequal	strcmp	strncmp
>	>	max	cummax	
?	if			sparse
@	push "for" value / "while" index	spiral	perms	randperm
A	all		dec2base. Larger base, any symbols	base2dec. Larger base, any symbols
B	logical(dec2bin(...)-'0')	bin2dec(char(...+'0'))	dec2bin	bin2dec
C		histcounts		cell
D	disp(num2str(…, ...))	disp(num2str(…))	sprintf / fprintf	disp
E				
F	*Not used. False (literal)*		format	
G		plot	image	colormap
H	Paste from clipboard H	Copy to clipboard H		
I	Paste from clipboard I	Copy to clipboard I		
J	Paste from clipboard J	Copy to clipboard J		
K	Paste from clipboard K	Copy to clipboard K		
L	Paste from level of clipboard L	Copy to level of clipboard L		
M				
N	stack size		NaN	isnan
O	zeros	datestr	datenum	datevec
P	flip	pdist	pdist2	polyval
Q				
R	triu	triu(…,1)	tril	tril(…,-1)
S	sort	sortrows	circshift	squareform
T	*Not used. True (literal)*		toeplitz	
U				
V				
W				
X	*Not used*	regexp	regexprep	
Y	*Not used*		inf	isinf
Z	*Not used*			
[	*Not used. Array delimiter*	ind2sub	floor	
\	mod	mod(…-1)+1	matrix \	
]	end (loops or conditional branches)	sub2ind	ceil	
^	.^	sqrt	matrix ^	
_	unary minus			
`	do...while	while	tic	toc
a	any			
b	bubble		strsplit	
c	char	cat	strcat	strjoin
d	diff	diag	blkdiag	gcd
e	reshape	permute	squeeze	exp
f	find	strfind	factor	
g	logical	ndgrid		gammaln
h	horzcat	{…, …}	hankel	hypergeom
i	input	urlread	imread	
j	input(…,'s')	real	imag	conj
k	lower	upper		
l	ones	abs	log. With two inputs, specifies base	log2
m	ismember	ismember(…,'rows')	mean	lcm
n	numel	nchoosek	interp1	
o	double	pi	round	fix
p	prod	cumprod	primes	isprime
q	accumarray	quantile	n-th prime	
r	rand	randn	randi	randsample
s	sum	cumsum	num2str	std
t	duplicate	str2double	str2num	strrep
u	unique	unique(…,'rows')		strjust
v	vertcat		strtrim	deblank
w	swap			
x	delete from stack	clc		
y	size	eye	hypot	
z	nnz	nonzeros	isempty	
{	*Not used. Cell array delimiter*	num2cell	mat2cell	mat2cell(x,ones(size(x,1),1),size(x,2))
|	or	union	norm	bitor
}	else		cell2mat	split array
~	Not	setxor	xor	bitxor

Normal function	Normal function: indexing	Normal function: literal	Meta-function	Stack rearranging function
Clipboard function	Control flow	Used only in literals	Separator, comment	Not used

Table 4: Behaviour of functions regarding consumption of inputs and of input/output specifications

	Consume inputs	Consume input and output specifications
Normal	Yes	Yes
Meta	Yes	No
Stack rearranging	No	Yes
Clipboard	No	Yes

- `iscolumn`, `isvector` etc: they can be realized with `size`, and are not much used anyway, I think.
- `issorted`: it can be easily realized by comparing with the result of `sort`. Of course `issorted` would be more time-efficient, but time efficiency is not the main purpose of code golfing.
- `rem`: probably having `mod` is enough.
- matrix inverse: not used very often; and can be done with `eye` and matrix division
- `det`, `rank`, matrix pseudoinverse, matrix decompositions, matrix power: probably too specialized
- `isequaln`: not used very often, is it?
- `swapbytes`: too specialized. Or is it worth including it?
- `sign`: it can be done in three characters with `abs` and element-wise division.
- `factorial`: it can be done with `:` and `prod`.
- `bitshift`, `bitcmp`: other bit-wise functions have been included. These two can be easily done with arithmetic operations.
- `strcmpi`: there's `strcmp`.
- `discretize`: the third output of `histcounts` gives that.

9 The MATL compiler

9.1 Usage

The official MATL compiler is written in MATLAB. It takes a source program in MATL and produces an output (compiled) program that is run in MATLAB.

It can be used in any of the following ways:

- `matl -options program`, or `matl -options 'program'` (command syntax);

- `matl('-options','program')` (function syntax).

In either case, the first input specifies processing options, starting with the character `-`; and the second input is a string or character array that contains the MATL program, or the name of a text file where the MATL program is stored. Both input arguments are optional.

The **first input** argument can specify the following options for the `matl` command:

- `p`: parse. Writes parsed program, with one line for each statement and using indenting, in text file `MATLp.txt`.

- `l`: listing. The parsed MATL program listing is shown on screen with one line for each statement and using indenting. Implies `p`.

  Two single-digit numeric options can be used to specify the indenting base (number of spaces to be included before any statement) and the indenting step (number of additional spaces for each nesting level). Default values are 4 and 2 respectively. If only one numeric option is provided, it is interpreted as the indenting base.

  If a *third* numeric option is provided, it is interpreted as the minimum number of spaces before comment symbol. A comment symbol is included at the end of each line, with all comment symbols vertically aligned and separated at least the specified number of spaces from the corresponding statement. This is useful for adding explanations to the code.

  Numeric options can be digits `0`, ..., `9`; or they can be *capital* letters `A`, `B`, ..., `Z`, which are interpreted as numbers 10 (`A`), 11 (`B`), ..., 35 (`Z`).

  File `MATLp.txt` containing the parsed program uses the indenting step, but not the indenting base.

  If options `c`, `r` or `d` have also been provided, the `matl` command stops at this point and waits for a key press before continuing, in order to give the user time to see or copy the displayed listing.

- `e`: listing with comments. It's like option `l` but automatic comment texts are added depending on the statement. These provide a good starting point to explain what the code does.

  In addition to the three numeric options used for option `l`, a *fourth* number can be specified (using capital letters 10, 11, ...). This is interpreted as the number of spaces between comment symbols and comment text. If less than four numeric options are provided, the rest take their default values. The third and fourth numeric options have default values 6 and 1 respectively.

  File `MATLp.txt` doesn't include comments.

As happens with `l`, if options `c`, `r` or `d` have been provided in addition to `e`, the `matl` command stops at this point and waits for a key press before continuing.

- `c`: compile. Produces a `.m` file called `MATLc.m` that can be run in MATLAB. Implies `p`.
- `r`: run. Runs the compiled program in MATLAB. Implies `p` and `c`. If an error occurs in the program, the error message includes a link to open the parsed MATL file at the line of the statement that caused the error.
- `d`: debug. Runs the compiled program in MATLAB in debug mode. Implies `p` and `c`. Breakpoints are set at the beginning of (the MATLAB code corresponding to) each MATL statement, to allow step-by-step execution. The variable editor is opened to show the MATL stack (variable `STACK`), input and output specifications (`S_IN` and `S_OUT`), and clipboard contents (`CB_H`, ..., `CB_L`).
- `f`: file. Indicates that the second input argument will not be MATL code, but the name of a file containing the code.
- `v`: verbose. Causes the compiler to provide detailed information about what it's doing.
- `h`: help. provides command-line help.

If no options are provided (or only option `f` is provided), the `matl` program defaults to `-r` (or `-rf`).

The **second input** argument is a string that represents one of the following, depending on the selected options:

1. The program code (options `p`, `l`, `e`, `c`, `r`, `r`,);
2. The name of a file containing the program code (option `f`);
3. Search text to get help (option `h`).

If the program code, file name or search text have commas, spaces, or other symbols that cause MATLAB to misinterpret the string, it needs to be enclosed in quotation marks; and then quotation marks contained within the string need to be duplicated.

In cases 1 and 2 the second input argument may be omitted. In this event the `matl` command waits for input from the keyboard containing the MATL program or the file name. The MATLAB prompt is changed to a single `>` symbol, which indicates MATL input mode. A program may be entered in a single line or in several lines (`Enter` key), and the end of the program is indicated by a blank line (`Enter` key twice). A file name is entered in a single line.

In case 3, if no string is provided as second input, general help about `matl` options is displayed. If a string is provided, it may contain the name of a MATL statement, or arbitrary search text. In the former case the information about that statement will be printed. In the latter, the search is based on case-insensitive partial matching with descriptions of MATL statements (which include equivalent MATLAB function names) and with automatic comment texts.

Examples:

- `matl`: waits for user input, and runs the program represented by that input. Files `MATLp.txt` and `MATLc.m` are produced.
- `matl 10;"@D]`: runs the program `10;"@D]`. Files `MATLp.txt` and `MATLc.m` are produced.
- `matl -d '3,4+D'`: runs the program `3,4,+D` in debug mode (quotation marks are needed because the code contains a comma). Files `MATLp.txt` and `MATLc.m` are produced.
- `matl -cf file.txt`: compiles the program contained in file `file.txt` and produces files `MATLp.txt` and `MATLc.m`.
- `matl('-l82v', myProgram)`: parses the program held in character array `myProgram`, shows the parsed result without automatic comments, using the numeric options for indenting, and produces file `MATLp.txt`. Detailed information about the process is also shown on screen.
- `matl -e0291 0T10;"tDtb+]`: parses the program provided as string input (code for the Fibonacci sequence given in §1.3) and shows the parsed result with automatic comments, using the numeric options for indenting. The parsed file `MATLp.txt` is also produced. The result printed on screen is as shown in Figure 1.
- `matl -h sort` would produce the result shown in Figure 2.

Thanks to `@AndrasDeak` for his help in testing the compiler.

9.2 Structure

The `matl` program consists of a **main function**, `matl`, which calls three other functions, corresponding to the parser, compiler, and runner/debugger.

The **parse function**, `matl_parse`, separates the program into statements. It includes the array checking referred to in page 6. Optionally it calls a display function, `matl_disp`, to print the listing of parsed code on screen. The output of the parser is a MATLAB struct array `S` in which each entry is a statement, with

```
0         % number literal
U         % array of ones
10        % number literal
;         % vector of equally spaced values
"         % for
  t       % duplicate
  D       % display
  t       % duplicate
  b       % bubble up element in stack
  +       % addition (element-wise, singleton expansion)
]         % end
```

Figure 1: Example of parsed code with automatic comments

```
S    sort
     1--3 (1);  1--2 (1)
     sort. If 2 inputs: a negative value of the second input
     corresponds to descending order
XS   sort rows
     1--2 (1);  1--2 (1)
     sortrows
```

Figure 2: Example of command-line help

fields describing the source code of the statement, the statement type and other information useful for the compiler.

The **compile function**, `matl_compile`, generates MATLAB code corresponding to each statement: literals, functions and control flow statements.

Each function is defined by the MATLAB code that the compiler generates for that function. That code is divided into *preamble*, *function body* and *postamble*. Most of the preamble and postamble code is common to different groups of functions: take inputs from stack, push results onto stack, delete `$` and `#` specifications). The function body, as well as some parameters to be used in the preamble and postamble, are specific to each function, and define what the function actually does.

Functions are defined by means of a *function definition file*, `funDef.txt`. It is a tab-separated plain text file that for each function contains the function body (directly as MATLAB code) and information that controls how the preamble and postamble code should be generated. This information includes allowed numbers of inputs and outputs, and whether the inputs should be consumed (deleted from the stack). The file also contains the text used in automatic comments.

The information about allowed and default inputs and outputs for a given function needs not (and cannot, for certain functions) be a fixed number. Consider as an example function `H`, which pastes the contents of clipboard H. Its default number of outputs is the number of elements contained by the clipboard, and thus can only be determined at run-time. This means that this default number is not known in advance. Instead, in the function definition file this parameter is defined by a *string* that gets directly inserted into the compiled code. The string can refer to the run-time information it needs, such as number of elements in the clipboard.

The function definition file is processed by a `genFunDef` function, which generates a *function definition struct array*, `F`, to be used by the compiler. This array contains the same information as the text file, and is saved into a `funDef.mat` file for future use. When the compiler encounters a function in the MATL parsed code, it looks it up in array `F` and generates the compiled (MATLAB) code accordingly. To speed up compilation, the generation of struct array `F` is only done when the function definition file is found to be newer than the processed file `funDef.mat`; otherwise the latter is loaded.

For functions that generate predefined literals, these are specified on a separate *predefined literal file*, `preLit.txt`. It is a plain text file that for each function defines a set of key-value pairs. This is processed by a `genPreLit` function, which generates a *predefined literal struct array*, `L`, to be used by the compiler. This array contains the same information as the text file, and is saved into a `preLit.mat` file for future use.

Lines of compiled code are stored in a cell array of strings `C`, from which the compiled file `MATLc.m` will be written.

The function definition file and predefined literal file centralize all information about functions. This allows to define new functions without actually modifying the compiler code.

The **run function**, `matl_compile`, executes the compiled program. If an error is found, an error message is issued with a link to the MATL statement that generated the error. In debug mode it inserts breakpoints and opens relevant variables (taking advantage of MATLAB's `openvar`).

The **help function**, `matl_help`, provides command-line help. It uses a struct array `H` that contains help information for all MATL functions and statements. This struct array is generated from the function definition file, the predefined literal file and additional information by a `genHelp` function, and is stored in file `help.mat`.

Appendix A Detailed function definitions

MATL functions are defined in Table 5. The notation for allowed and default numbers of inputs and outputs is as follows. Consider for example the `u` function (which corresponds to MATLAB's `unique`). Then "`u` 1–4 (1) 1–3 (1)" means that this function can accept 1 to 4 inputs, with 1 as default; and can produce 1 to 3 outputs, with 1 as default. This notation may be abbreviated if the number of inputs or outputs is unbounded or is fixed: "(3– (3) 1" means that the number of inputs can be any integer starting at 3 and the number of outputs is always 1.

Table 5: Function definitions

`!`	1	1	`.'` (`transpose`)
`X!`	1–2 (1)	1	`rot90`
`Y!`	1	0–2 (2)	`system`
`Z!`	1	1	`full`
`X"`	2– (3)	1	`repmat`
`Y"`	2– (2)	1	`repelem` (run-length decoding)
`#`	1	0	specify outputs for next function
`Y#`	1–2 (1)	0	`rng`
`Z#`	1–3 (1)	0	Appends first input to file `inout`, creating it if necessary. If the input is an array it is converted to char and written. If the input is a cell array input, the contents of each cell are converted to char and written, with a newline (character 10) in between. With 2 inputs: second input specifies filename; if empty defaults to `inout`. With 3 inputs: third input specifies whether any previous contents of file should be kept.
`$`	1	0	specify inputs for next function
`X$`	1– (2)	0– (1)	execute Matlab function specified by first input, using the rest of the inputs as arguments.
`Y$`	1–2 (1)	1	`char(vpa(...))`
`Z$`	0–1 (0)	1	Reads bytes from specifed file. The output is a row vector of char. If 0 inputs or empty input: file name is `inout`.
`X%`	1	1	class of input (`class` with one input)
`Y%`	2–3 (2)	1	`cast`
`Z%`	2	1	`typecast`
`&`	2	1	`&` (`and`), element-wise with singleton expansion
`X&`	2–4 (2)	1–3 (1)	`intersect`
`Z&`	2–3 (2)	1	`bitand`
`Y'`	1	2	run-length encoding (inverse of `repelem`). Input may be an array or cell array. Numeric values must be finite
`Z'`	0	1	`now`
`(`	3– (3)	1	assignment `( )` indexing
`X(`	3– (3)	1	assignment `{ }` indexing
`)`	2– (2)	1	reference `( )` indexing
`X)`	2– (2)	1	reference `{ }` indexing
`Y)`	1	1	linearize to column array (`(:)`)
`Z)`	1	0– (△)	generate comma-separated list from cell array (`{:}`) and push each element onto stack
`*`	2	1	`.*` (`times`), element-wise with singleton expansion
`X*`	2	1	`kron`
`Y*`	2	1	matrix product, `*` (`mtimes`)

`Z*`	1– (2)	1	Cartesian product. Given an n of vectors of possibly different sizes, generates an n-column matrix whose rows describe all combinations of elements taken from those vectors
`+`	2	1	`+` (`plus`), element-wise with singleton expansion
`X+`	2–3 (2)	1	`conv`
`Y+`	2–4 (2)	1	`conv2`
`Z+`	2	1	`convmtx`
`X,`	1	1	`cos`
`Y,`	1	1	`sin`
`Z,`	1	1	`tan`
`-`	2	1	`-` (`minus`), element-wise with singleton expansion
`X-`	2–4 (2)	1–2 (1)	`setdiff`
`Y.`	0–1 (1)	0	`pause` (without outputs)
`Z.`	2–3 (2)	1	`bitget`
`/`	2	1	`./` (`rdivide`), element-wise with singleton expansion
`X/`	1	1	`angle`
`Y/`	2	1	right matrix division, `/` (`mrdivide`)
`Z/`	1–3 (1)	1	`unwrap`
`X0`	1	1	predefined literal depending on input
`Y0`	1	1	predefined literal depending on input
`X1`	1	1	predefined literal depending on input
`Y1`	1	1	predefined literal depending on input
`X2`	1	1	predefined literal depending on input
`Y2`	1	1	predefined literal depending on input
`X3`	1	1	predefined literal depending on input
`Y3`	1	1	predefined literal depending on input
`X4`	1	1	predefined literal depending on input
`Y4`	1	1	predefined literal depending on input
`X5`	1	1	predefined literal depending on input
`X6`	1	1	predefined literal depending on input
`X7`	1	1	predefined literal depending on input
`X8`	1	1	predefined literal depending on input
`X9`	1	1	predefined literal depending on input
`:`	1–3 (1)	1	`colon` (with three inputs `x`, `y`, `z` produces `x:y:z`; with two inputs `x`, `y` produces `x:y`). If one input: produces `1:x`
`Z:`	2–3 (2)	1	`bitset`
`X;`	1	1	`acos`
`Y;`	1	1	`asin`
`Z;`	2	1	`atan2`, element-wise with singleton expansion
`<`	2	1	`<` (`lt`), element-wise with singleton expansion
`X<`	1–3 (1)	1–2 (1)	`min`. If 2 inputs: element-wise with singleton expansion
`Y<`	1–3 (1)	1	`cummin`
`=`	2	1	`==` (`eq`), element-wise with singleton expansion
`X=`	2– (2)	1	`isequal`

`Y=`	2	1	`strcmp`
`Z=`	3	1	`strncmp`
`>`	2	1	`>` (`gt`), element-wise with singleton expansion
`X>`	1–3 (1)	1–2 (1)	`max`. If 2 inputs: element-wise with singleton expansion
`Y>`	1–3 (1)	1	`cummax`
`Z?`	1–6 (3)	1	`sparse`
`X@`	1	1	`spiral`
`Y@`	1	1	`perms`
`Z@`	1–2 (1)	1	`randperm`
`A`	1–2 (1)	1	`all`
`YA`	2–4 (2)	1	`dec2base`. If second input has more than one element: it defines the symbols, which can be characters or numbers. The number of symbols defines the base, which can exceed 36
`ZA`	2	1	`base2dec`. If second input has more than one element: it defines the symbols, which can be characters (case-sensitive) or numbers. The number of symbols defines the base, which can exceed 36
`B`	1–2 (1)	1	`logical(dec2bin(...)-'0')`
`XB`	1	1	`bin2dec(char(...+'0'))`
`YB`	1–2 (1)	1	`dec2bin`
`ZB`	1	1	`bin2dec`
`XC`	1–7 (2)	1–3 (1)	`histcounts`
`YC`	1– (2)	1	`cell`
`D`	0– (1)	0	If 1 input: `disp(num2str(..., '%.16g '))`. If several inputs: `disp(num2str(eachInput,lastInput))`, where `eachInput` loops over all inputs but the last. In either case, (nested) cell arrays are (recursively) unboxed in linear order. *See also* `XD`, `YD`, `ZD`
`XD`	0– (‡)	0	`disp(num2str(eachInput))`, where `eachInput` loops over all inputs. (Nested) cell arrays are (recursively) unboxed in linear order. *See also* `D`, `YD`, `ZD`
`YD`	1– (2)	0–2 (1)	`sprintf`. If 0 outputs: prints to screen using `fprintf(...)` (without file identifier). *See also* `D`, `XD`, `ZD`
`ZD`	0– (1)	0	`disp` for each input. *See also* `D`, `XD`, `YD`
`YF`	0–1 (1)	0	`format`
`XG`	1– (1)	0	`plot`. Calls `drawnow` to update figure immediately
`YG`	0–3 (1)	0	`image(...), axis ij, axis image`. Calls `drawnow` to update figure immediately
`ZG`	1	0–1 (0)	`colormap`. With 0 outputs, calls `drawnow` to update figure immediately
`H`	0	0– (†)	paste from clipboard H
`XH`	0– (1)	0	copy to clipboard H
`I`	0	0– (†)	paste from clipboard I

`XI`	0– (1)	0	copy to clipboard I
`J`	0	0– (†)	paste from clipboard J
`XJ`	0– (1)	0	copy to clipboard J
`K`	0	0– (†)	paste from clipboard K
`XK`	0– (1)	0	copy to clipboard K
`L`	0	0– (†)	paste from multi-level clipboard L. Input specifies level
`XL`	1– (2)	0	copy to multi-level clipboard L. Topmost input specifies level
`N`	0	1	number of elements in the stack
`YN`	0– (0)	1	`NaN` function. If 0 inputs: produces literal `NaN`.
`ZN`	1	1	`isnan`
`O`	0– (0)	1	`zeros` (if 0 inputs: produces output 0)
`XO`	1–4 (1)	1	`datestr`
`YO`	1–6 (1)	1	`datenum`
`ZO`	1–3 (1)	1–6 (1)	`datevec`
`P`	1–2 (1)	1	`flip`
`XP`	1–2 (1)	1	`pdist`
`YP`	2–5 (2)	1	`pdist2`
`ZP`	2–3 (2)	1	If 2 inputs `p` and `x`: `y=polyval(p,x)`. If 3 inputs `p`, `x` and `mu`: `y=polyval(p,x,[],mu)`
`R`	1–2 (1)	1	`triu`. *See also* `XR`.
`XR`	1	1	`triu(..., 1)`. *See also* `R`.
`YR`	1–2 (1)	1	`tril`. *See also* `ZR`.
`ZR`	1	1	`tril(..., -1)`. *See also* `YR`.
`S`	1–3 (1)	1–2 (1)	`sort`. If 2 inputs: a negative value of the second input corresponds to descending order
`XS`	1–2 (1)	1–2 (1)	`sortrows`
`YS`	2–3 (2)	1	`circshift`
`ZS`	1–2 (1)	1	`squareform`
`YT`	1–2 (2)	1	`toeplitz`
`XX`	2–9 (2)	1–6 (1)	`regexp`
`YX`	3–5 (3)	1	`regexprep`
`YY`	0– (0)	1	`inf` function. If 0 inputs: produces literal `inf`.
`ZY`	1	1	`isinf`
`X[`	2	1– (2)	`ind2sub`
`Y[`	1	1	`floor`
`\`	2	1	`mod`, element-wise with singleton expansion
`X\`	2	1	`mod(...-1)+1`, element-wise with singleton expansion
`Y\`	2	1	left matrix division, `\` (`mldivide`)
`X]`	3– (3)	1	`sub2ind`
`Y]`	1	1	`ceil`
`^`	2	1	`.^` (`power`), element-wise with singleton expansion
`X^`	1	1	`sqrt`
`Y^`	2	1	`^` (`mpower`)
`_`	1	1	unary `-` (`uminus`)

Y‘	0	0–1 (0)	tic
Z‘	0–1 (0)	0–1 (1)	toc
a	1–2 (1)	1	any
b	0– (3)	0	bubble up element in stack
Yb	1– (1)	1–2 (1)	strsplit
c	1– (1)	1	char
Xc	3– (3)	1	cat
Yc	2– (2)	1	strcat
Zc	1–2 (1)	1	strjoin
d	1–3 (1)	1	diff
Xd	1–2 (1)	1	diag
Yd	1– (2)	1	blkdiag
Zd	2	1–3 (1)	gcd
e	2– (3)	1	reshape
Xe	2	1	permute
Ye	1	1	squeeze
Ze	1	1	exp
f	1–3 (1)	1–3 (1)	find
Xf	2–4 (2)	1	strfind
Yf	1	1	factor
g	1	1	logical
Xg	1– (2)	1– (2)	ndgrid
Zg	1	1	gammaln
h	1– (2)	1	horzcat
Xh	0– (‡)	1	concatenate into cell array ({...,...})
Yh	1–2 (2)	1	hankel
Zh	3	1	hypergeom. If any input is of type char: returns char output
i	0–2 (0)	1	input with content checking. If 0 inputs: uses default prompt string. *See also* j.
Xi	1–5 (1)	1–2 (1)	urlread
Yi	1– (1)	1–3 (1)	imread
j	0–1 (0)	1	input(..., 's'). If 0 inputs: uses default prompt string. *See also* i.
Xj	1	1	real
Yj	1	1	imag
Zj	1	1	conj
k	1	1	lower
Xk	1	1	upper
l	0– (0)	1	ones (if 0 inputs: produces output 1)
Xl	1	1	abs
Yl	1–2 (1)	1	log. If two inputs: second input specifies logarithm base
Zl	1	1–2 (1)	log2
m	2–4 (2)	1–2 (1)	ismember. *See also* Xm
Xm	2–3 (2)	1–2 (1)	ismember(..., 'rows', ...). *See also* m

`Ym`	1–4 (1)	1	`mean`
`Zm`	2	1	`lcm`
`n`	1	1	`numel`
`Xn`	2	1	`nchoosek`
`Yn`	1–5 (2)	1	`interp1`. `'pp'` option not supported
`o`	1	1	`double`
`Xo`	0	1	`pi`
`Yo`	1–3 (1)	1	`round`
`Zo`	1	1	`fix`
`p`	1–3 (1)	1	`prod`
`Xp`	1–3 (1)	1	`cumprod`
`Yp`	1	1	`primes`
`Zp`	1	1	`isprime`
`q`	2–6 (2)	1	`accumarray`
`Xq`	2–3 (2)	1	`quantile`
`Yq`	1	1	Finds the n-th prime for each value n in the input array
`r`	0– (0)	1	`rand`
`Xr`	0– (0)	1	`randn`
`Yr`	1– (1)	1	`randi`. If 1 input: `randi(...,1)`
`Zr`	2–4 (2)	1	`randsample`. Does not support stream specification
`s`	1–4 (1)	1	`sum`
`Xs`	1–3 (1)	1	`cumsum`
`Ys`	1–2 (1)	1	`num2str`
`Zs`	1–4 (1)	1	`std`
`t`	0– (1)	0	duplicate elements in stack
`Xt`	1	1	`str2double`
`Yt`	1	1–2 (1)	`str2num` with content checking
`Zt`	3	1	`strrep`
`u`	1–4 (1)	1–3 (1)	`unique`. *See also* `Xu`.
`Xu`	1–3 (1)	1–3 (1)	`unique(...,'rows',...)`. *See also* `u`.
`Zu`	1–2 (1)	1	`strjust`
`v`	1– (2)	1	`vertcat`
`Yv`	1	1	`strtrim`
`Zv`	1	1	`deblank`
`w`	0– (2)	0	swap elements in stack
`x`	0– (1)	0	delete from stack
`Xx`	0	0	`clc`
`y`	1–2 (1)	1– (1)	`size`
`Xy`	1–4 (1)	1	`eye`
`Yy`	2	1	`hypot`, element-wise with singleton expansion
`z`	1	1	`nnz`
`Xz`	1	1	`nonzeros`
`Yz`	1	1	`isempty`
`X{`	1–2 (1)	1	`num2cell`
`Y{`	2– (3)	1	`mat2cell`

<table>
<tr><td>Z{</td><td>1</td><td>1</td><td>mat2cell(x, ones(size(x,1),1), size(x,2),...,size(x,ndims(x))). It’s a generalization of cellstr that works for numeric, logical or char arrays of any number of dimensions</td></tr>
<tr><td>|</td><td>2</td><td>1</td><td>| (or), element-wise with singleton expansion</td></tr>
<tr><td>X|</td><td>2–4 (2)</td><td>1–3 (1)</td><td>union</td></tr>
<tr><td>Y|</td><td>1–2 (1)</td><td>1</td><td>norm</td></tr>
<tr><td>Z|</td><td>2–3 (2)</td><td>1</td><td>bitor</td></tr>
<tr><td>Y}</td><td>1</td><td>1</td><td>cell2mat</td></tr>
<tr><td>Z}</td><td>1</td><td>1– ($^{\triangledown}$)</td><td>split array into its elements in linear order</td></tr>
<tr><td>~</td><td>1</td><td>1</td><td>~ (not)</td></tr>
<tr><td>X~</td><td>2–4 (2)</td><td>1–3 (1)</td><td>setxor</td></tr>
<tr><td>Y~</td><td>2</td><td>1</td><td>xor, element-wise with singleton expansion</td></tr>
<tr><td>Z~</td><td>2–3 (2)</td><td>1</td><td>bixtor</td></tr>
</table>

† Current number of elements in clipboard (H, I, J, K), or in clipboard level (L).
‡ Current number of elements in stack.
$^{\triangle}$ Number of elements of input cell array.
$^{\triangledown}$ Number of elements of input array.

Some things I’m not sure of:

Is it better to have 1 or 2 inputs to vpa by default? If 1, what’s an appropriate initial condition for digits?

Functions X0...Z9 produce predefined literals depending on their numeric input. These are specified in Table 6.

Table 6: Output of predefined literal functions

Function `X0`					
1	'stable'	2	'sorted'	3	'rows'
4	'last'	5	'reverse'	10	'first'
11	'forward'	12	'legacy'		

Function `X1`					
1	'bank'	2	'compact'	3	'hex'
4	'short'	5	'shortg'	6	'long'
7	'longg'	8	'loose'	9	'rat'
10	'longe'	11	'longeng'	12	'shorte'
13	'shorteng'				

Function `X2`					
1	'double'	2	'int8'	3	'int64'
4	'uint8'	5	'uint64'	6	'char'
7	'logical'	8	'single'	10	'int16'
11	'int32'	12	'uint16'	13	'uint32'
14	'like'	15	'decimals'	16	'includenan'
17	'native'	18	'omitnan'	19	'significant'

Function `X3`					
1	'start'	2	'end'	3	'tokenExtents'
4	'match'	5	'tokens'	6	'names'
7	'split'	8	'once'	10	'ignorecase'
11	'preservecase'				

Function `X4`					
1	'CollapseDelimiters'	2	'DelimiterType'	3	'RegularExpression'
4	'center'	5	'left'	6	'local'
10	'right'				

Function `X5`					
1	'full'	2	'same'	3	'valid'
6	'linear'	7	'nearest'	8	'next'
9	'previous'	10	'spline'	11	'pchip'
12	'cubic'	13	'v5cubic'	14	'extrap'

Table 6: Output of predefined literal functions—*continued*

Function X6					
1	'get'	2	'post'	3	'Timeout'
9	'none'	11	'Frames'	12	'BackgroundColor'
13	'Index'	14	'Info'	15	'ReductionLevel'
16	'PixelRegion'	17	'V79Compatible'		

Function X7					
1	'BinWidth'	2	'BinLimits'	3	'Normalization'
4	'count'	5	'probability'	6	'countdensity'
7	'pdf'	8	'cumcount'	9	'cdf'
10	'BinMethod'	11	'auto'	12	'scott'
13	'fd'	14	'integers'	15	'sturges'
16	'sqrt'				

Function X8					
1	'cityblock'	2	'minkowski'	3	'chebychev'
4	'cosine'	5	'correlation'	6	'hamming'
10	'euclidean'	11	'seuclidean'	12	'mahalanobis'
13	'spearman'	14	'jaccard'	15	'fro'
16	'Smallest'	17	'Largest'	18	'tomatrix'
19	'tovector'				

Function X9					
1	'shuffle'	2	'default'	3	'twister'
4	'simdTwister'	5	'combRecursive'	6	'multFibonacci'
10	'v5uniform'	11	'v5normal'	12	'v4'

Table 6: Output of predefined literal functions—*continued*

Function `Y0`					
1	`'Color'`	2	`'LineStyle'`	3	`'LineWidth'`
4	`'Marker'`	5	`'MarkerSize'`	6	`'on'`
7	`'off'`	8	`'CDataMapping'`	9	`'scaled'`
20	`'auto'`	21	`'LineJoin'`	22	`'chamfer'`
23	`'miter'`	24	`'round'`	25	`'square'`
26	`'diamond'`	27	`'pentagram'`	28	`'hexagram'`
29	`'visible'`	30	`'Clipping'`	31	`'MarkerEdgeColor'`
32	`'MarkerFaceColor'`	40	`'direct'`	41	`'AlphaData'`
42	`'AlphaDataMapping'`	43	`'Cdata'`	44	`'Xdata'`
45	`'Ydata'`	46	`'ZData'`	50	`'autumn'`
51	`'bone'`	52	`'colorcube'`	53	`'cool'`
54	`'copper'`	55	`'flag'`	56	`'gray'`
57	`'hot'`	58	`'hsv'`	59	`'jet'`
60	`'lines'`	61	`'parula'`	62	`'pink'`
63	`'prism'`	64	`'spring'`	65	`'summer'`
66	`'winter'`				

Function `Y1`					
1	`'mean'`	2	`'min'`	3	`'max'`
4	`'{sort(x)}'`	5	`'{sort(x).''}'`	10	`'sum'`
11	`'x(end)'`	12	`'{cumsum(x)}'`	13	`'{cumsum(x).''}'`
14	`'nansum'`	15	`'nanmean'`	16	`'nanmin'`
17	`'nanmax'`	18	`'{cummax(x)}'`	19	`'{cummax(x).''}'`

Function `Y2`					
1	`'A':'Z'`	2	`'a':'z'`	3	`['A':'Z' 'a':'z']`
4	`'0':'9'`	11	`'aeiou'`	12	`'AEIOU'`
13	`'aeiouAEIOU'`				