The MATL programming language

Luis Mendo

February 25, 2016

1 Introduction

MATL is a programming language based on MATLAB/Octave 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.ˌl/ 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 weird/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. These are explained in §12.

1. Infinite loop that does nothing:

   MATL: `'T]`

   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
   ```

   MATLAB: `while 1,end`

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

   MATL: `j1X02$u`

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

3. 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)));
   ```

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`, or with functions corresponding to `double` and `int64`.

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 linebreaks 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 `U` (`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]. The best approach seems to be: use regular expressions to detect function or variable names, but ignoring such names if they are within 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.

For safety, the input obtained from `i` (`input`) is also checked this way before being evaluated, as is the argument to `U` (`str2num`) (even though MATLAB does not do this).

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 a maximum number 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.

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 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. For ease of presentation, the case of logical arrays is discussed first. If the input is 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 `3XyZy` and `3XyTF#Zy` (`Xy` is `eye`, and `Zy` 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? ...?".

If the input to `#` is a number n, there are two modes in which this number can be interpreted:

- *As number of inputs*: when n is in the range defined by the minimum and maximum numbers of outputs for the considered function, it indicates that the function will be called with n outputs. This is analogous to how `$` works with a number.

- *As a specific output*: when n exceeds the maximum allowed number of outputs M, it addresses the $(n-M)$-th output. It is equivalent to a logical vector with one `T` preceded by $n-M-1$ times `F`. As an example, for a function with a maximum number of outputs $M=3$, `5#` would be equivalent to `FT#`, and `6#` to `FFT#`.

  The "`6#`" notation in the preceding example is shorter, whereas "`FFT#`" is probably clearer.

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. Any `#` specification other than `0#` will give an error (and specifying `0#` is unnecessary).

- `x` (delete). It can have an arbitrary number $n = 0, 1, 2, \ldots$ 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 $n = 0, 1, 2, \ldots$ 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 elements). 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.

- `y` (duplicate element). This function is similar to `t`, but it only duplicates the lowest element among those specified as inputs. It can have an arbitrary number $n = 0, 1, 2, \ldots$ specified to `$`; by default 2. Or a logical array can be used as input specification. For example, if the stack contains `'a'`, `'b'`, `'c'`, `'b'`, either `3$y` or `TTF$y` would leave the stack as `'a'`, `'b'`, `'c'`, `'a'`. Without input specification, `k` duplicates the second element from the top.

- `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

Clipboards can be of two types: standard and automatic. Standard clipboards behave like variables: they allow copying values from the stack and pasting them (pushing them to the stack). Automatic clipboards are similar but copying is automatically triggered when certain events happen.

5.4.1 Standard clipboards

There are five standard 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 `XH`, 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 `H`, 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 `0` 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 a certain number of levels initially (necessary to hold the initial contents specified in Table 2). Higher levels are created at run-time *on the fly*. For example, if clipboard L currently has 20 levels, copying some elements to level 22 will cause MATL to create levels 21 and 22: level 21 will be empty, and level 22 will hold the copied elements. Trying to paste from level 23 gives an error, because that level doesn't exist; but level 21 now exists, and is empty. Therefore `21L` is valid code, and does nothing. `0#21L` is also valid, but `1#21L` (or even `F#21L`) is not, because a non-existent first element in level 13 is being referenced.

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. These four clipboards, plus the automatic ones, are probably enough in almost all situations. When that's not the case, clipboard L can be used 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

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 3 2]`
L, level 4	`[3 1 2]`
L, level 5	`[1 -1j]`
L, level 6	`[2 0]`
L, level 7	`[1 -1j 0]`
L, level 8	`[2 -1j]`
L, level 9	`[1 2 0]`
L, level 10	`[2 2 0]`
L, level 11	`3600`
L, level 12	`86400`
L, level 13	`1440`
L, level 14	`[31 28 31 30 ... 31]`
L, level 15	`[31 29 31 30 ... 31]`
L, level 16	2π
L, level 17	Euler-Mascheroni constant, γ
L, level 18	Golden ratio, φ

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.

5.4.2 Automatic clipboards

There are two automatic clipboards: G and M. These are multi-level clipboards, like L. A distinctive feature of clipboards G and M is that they don't have copying functions. Instead, copying happens automatically.

Clipboard G automatically stores all **user inputs**. User inputs may be obtained from explicit input functions (see §8) or implicitly (see §7.3). Each input is automatically stored on a different level of clipboard G, starting from 1. Clipboard G is initially empty, and each input fills a new level.

Function `G` is used for pasting from clipboard G. It can have 0 or 1 input arguments. With 0 input arguments, `G` retrieves the contents of all levels that currently exist in clipboard G. The default number of outputs is the current number of clipboard levels (each level contributes one output), although this can be of course modified by an output specification. The content of each level is pushed onto the stack separately. For example, if clipboard G currently has 2 levels, this produces 2 new elements in the stack. If clipboard G has no levels (no user input has occurred yet), calling `G` without inputs does nothing by default; and causes an error for any output specification other than `0`.

With 1 input argument, `G` retrieves the contents of the clipboard level specified by the argument. The default number of outputs in this case is 1. For example, `2G` retrieves the second user-input (assuming there have been two user-inputs; otherwise it causes an error). The selected user-input can also be specified using a form of `end`-based addressing, similar to `end`-based indexing in arrays (see §5.5). So `0G` would retrieve the (currently) last user-input, `-1jG` would retrieve the second-last input, etc.

This function by default has 0 input arguments if clipboard G currently has zero or one levels, and 1 input argument otherwise. This is intended to facilitate clipboard handling: when there are zero or one levels there is no need to specify which level to paste from. So without `$` or `#` specifications, function `G` behaves as follows: when only one input has been asked from the user `G` pushes that user-input; when more than one input has been asked, `2G` for example pushes the second user-input.

Clipboard M automatically stores all **function inputs** from the 4 most recent call to *normal* functions that took *at least one* input. The inputs corresponding to each function call are stored on a different level, by increasing order of age. So clipboard level 1 contains all inputs of the most recent call (to a normal function that used inputs); level 2 from the second most recent call, etc.

Function `M` is used for pasting from clipboard M. It takes one numeric input, which can be interpreted in two different ways depending on its value:

- *Combined addressing*: `M` takes as input a number 1, ..., 4. The number defines the addressed level. The default number of outputs is the number elements stored in that level (that is, the number of inputs of the function call used). So for example `[3,7,0,5]u1Mf` produces the vectors `[0,3,5,7]` (`unique([3,7,0,5])`) and `[1,2,4]` (`find([3,7,0,5])`); and `3,4+,1M*` produces `7` and `12`.

  If the referred function call hasn't actually occurred, `M` produces no outputs. For example, `3M` does nothing if there have been less than three normal function calls with inputs. However, any output specification other than `0` gives rise to an error in this case.

- *Individual addressing*: `M` takes as input a number 5, 6, ... The number

individually addresses inputs from function calls that used more than one input. Addressing is from most recent to less recent call, and from most recent (rightmost) to less recent (leftmost) input in the function signature. Only calls that had more than one input are considered, because single-input calls are already covered by combined addressing. So 5 refers to the latest input of the latest call that used several inputs; 6 refers to the second latest such input, etc.

If the addressed function input doesn't actually exist, `M` gives an error. For example, `5M` gives an error if there haven't been any normal function calls with several inputs.

As an example, consider the following code, where commas are used to separate statements for clarity: `10,20,30,3$+,[5 0 7],1,4,X0,3$f,=`. There are four function calls, with the following inputs and outputs:

1. `3$+` (addition): inputs are `10`, `20`, `30`. Produces `60`.
2. `X0` (predefined literal): input is `4`. Produces the string `'last'`.
3. `3$f` (`find`): inputs are `[5 0 7]`, `1`, `'last'`. Produces `3`.
4. `=`: inputs are `60`, `3`. Produces `F` (`false`).

In this scenario, `4M` would push `10`, `20`, `30` onto the stack. `5M` would push `3`. `9M` would push `[5 0 7]`. `10M` would push `30`, no `4`, because `4` was a single input (which could be produced with `3M`).

Note the similarity of `M` with function `L`, which is used for pasting from multi-level clipboard L. In both cases, the input argument to the function specifies the clipboard level which is being addressed, and the output specification controls which of the multiple elements (arrays) existing in that level are pushed to the stack. In function `G`, on the other hand, both the input argument and the output specification specify which levels are pushed, each level containing only one element (array). Another difference between clipboards G and M is that the former uses chronological order, whereas the latter uses reverse chronological order.

Clipboards G and M differ from H,. . . ,L in that here copying is done automatically, and it doesn't use or consume any `$` or `#` specifications. But pasting with functions `G` and `M` does consume `$` and `#` specifications, like paste functions of standard clipboards do.

The default number of inputs and outputs of `G` has been defined such that, in normal usage, an input number is used to address the intended level; but this number is spared when it's not needed because there is only one level. Note that with this function there may be several ways of obtaining the same result using different input and output specifications. For example, if clipboard G has two levels, `0$1#G` is equivalent to `1$1G`.

Perhaps a cleaner syntax could have been used to distinguish combined addressing and individual addressing in clipboard M, such as negative numbers for individual addressing, two inputs, or a different function name. However, all those alternatives either require using more characters in the code or consume more character space.

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. Indexing with **curly braces** is used for accessing cell's contents in a cell array.

Indexing can be used for two purposes: **referencing** elements of an array, or **assigning** values to elements of an array. For example, in the statement `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 the following:

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

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.

In addition, there are four functions that include a colon index. This is useful for indexing rows or columns of 2D arrays:

- `Y)` round-bracket reference indexing with final colon;
- `Z)` round-bracket reference indexing with initial colon;
- `Y(` round-bracket assignment indexing with final colon;
- `Z(` round-bracket assignment indexing with initial colon;

There are also two functions that correspond to MATLAB's reference indexing with a colon only:

- `X:` round-bracket reference indexing with colon;
- `Y:` curly-brace reference indexing with colon.

5.5.1 Basic indexing functions

`)` takes an arbitrary number of inputs $2, 3, 4, \ldots$, by default 2, and produces 1 output by default. 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 §5.4.1) represents `2:end-1`.
- `[-.5j,2,0]` represents `end/2:2:end`.
- `[1,0]` represents `1:end`.

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 §5.4.1).

> 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`. 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).

The function `)` can also be used with 2 outputs. In this case only one index can be used as input (in addition to the array to be indexed). The first output is the same

as described above, and the second is the "complementary" array formed by the index values not present in the input index. For example, `[2,4,6,8][2,3]2#)` produces the arrays `[4,6]` and `[2,8]`, as does `[2,4,6,8]FTT2#)`.

`(` 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 certain situations, such as 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$`) instead of numbers (`3$`).

A special case is *null assignment*, that is, `x(...) = []`, which is used to remove elements from an array. With function `(` this can only be done in a row or column array (numeric, character or cell) with a single index. For example, `{10 20 30}[][1 3](` produces the cell array `{20}`.

> The reason of this restriction is that neither `x([1 3],1:10) = []` nor `x([1 3],1:end) = []` are allowed in MATLAB. Instead, a colon needs to be explicitly used: `x([1 3],:) = []`; and MATL doesn't have an equivalent of `:` to be specified directly as index. However, see functions `Y(` and `Z(` below.

`X)` is analogous to `)`, but corresponds to MATLAB'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 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 MATLAB'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 `X(` will do the assignment. 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.

5.5.2 Colon indexing functions

`Y)` is similar to `)`, except that a colon index is added *after* the specified index or indices. It takes an arbitrary number of inputs; by default 2, corresponding to the array and one specified index. By default it produces 1 output. For example, if the stack contains the array `[1 2; 3 4]`, the code `1Y)` produces the first row of the array, that is, `[1 2]`. So it's equivalent to the more cumbersome `1[1 0]3$)` (the latter uses `[1 0]`, or `1:end`, instead of `:`, but they are equivalent in this case).

`Z)` is analogous to `Y)` except that the colon index is added *before* the specified index or indices. Thus if the stack contains the array `[1 2; 3 4]`, the code `1Z)` produces the first column of the array, `[1; 3]`. The equivalent code using `)` would be `[1 0]1,3$)`.

The functions `Y)` and `Z)` also have 2-output versions, analogous to that of `)`. Only one index can be used as input (in addition to the array to be indexed), and the second output is the "complementary" array. For example, `[1,2;3,4;5,6] [1,3]2#Y)` produces the arrays `[1,2;5,6]` and `[3,4]`.

`Y(` and `Z(` add a final or initial colon respectively in assignment indexing. They take at least 2 inputs; by default 3, corresponding to destination array, data, and one specified index. They produce 1 output. For example, if the stack contains `[1 2; 3 4]`, the code `5,1Y(` produces the array `[5 5; 3 4]`. It is thus equivalent to the more cumbersome `5,1[1 0]4$(` (again, using `1:end` instead of `:`)

The functions `Y(` and `Z(` support *null assignment* with a single specified index. Thus, given an array `[1 2; 3 4]`, `[]1Z(` removes its first column, producing `[2;4]`.

In this case there is no equivalent code for null assignment using `(`. The reason is, again, that for null assignment `:` and `1:end` are *not* equivalent:

`x(1,:) = []` is valid, whereas `x(1,1:end) = []` is not. In MATL `1:end` can be specified as an input to `(:`, but `:` cannot.

`X:` 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]X:` would replace the input array `[1,2;3,4]` by `[1;3;2;4]`. An equivalent result could be obtained by `Y)` or `Z)` without specifying any index (so that only `:` is applied): `[1,2;3,4]1$Y)` or `[1,2;3,4]1$Z)`.

`Y:` 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 `Y:` 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} Y:` 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 `.` (`break`) and `X.` (`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 `.` (“break”) causes the program to exit the innermost “for”, “do…while” or “while” loop.

The statement `X.` (“continue”) ends the current iteration of the innermost loop, and the loop proceeds with the next iteration, if any.

Statements `]` at the very end of the program can be omitted; see §7.

> 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 Implicit actions

7.1 Initial actions

When a MATL program is run, it begins with `format compact`, `format long`, `warning('off','all')`, `close all`. The default colormap is set as `gray(256)`. The seed of the random number generator is set using `rng('shuffle')`.

7.2 Final actions

At the end of the program, any loops or conditional branches that have not been closed by a corresponding `]` statement are implicitly closed. That is, as many `]` statements as needed are implicitly included after the last (explicit) statement in the source code.

After that, the display function `XD` is implicitly called. 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 delete all stack contents).

If called with 1 or more inputs, this function modifies clipboard M, and consumes its inputs from the stack, exactly as if the program included an explicit `XD` statement at the end. (This is probably irrelevant anyway, because it is by definition the last function call, so the clipboard or the stack can't be accessed afterwards).

> In fact, this implicit function is implemented by the compiler adding a `XD` statement at the end of the source code. This statement is marked as implicit for displaying purposes, but when generating the compiled code it is treated as if it were an explicit function call.

When the program finishes, the original MATLAB `warning` state and default colormap are restored.

7.3 Implicit inputs

In addition to the explicit `i` and `j` input functions (see §8), user input is implicitly triggered when a non-existent element of the stack is accessed by *any* statement. This implicit input is always *numeric* (evaluated), that is, corresponds to function `i`. But the implicit input method is not strictly a function call (for example, it doesn't consume `$` or `#` specifications).

In general, the stack at any given moment contains elements at positions 1, 2, ... N, where N is the current number of elements in the stack (possibly 0). Implicit inputs can be viewed as follows: the stack is indefinitely *extended below the bottom*, that is, at positions 0, -1, -2, ... with values that are not initially defined, but are resolved on the fly via implicit input. These inputs are asked from the user only when they are needed, in the order in which they are needed. If several inputs are required at the same time they follow the normal stack order, that is, the input that is deepest in the (extended) stack is entered first.

A few examples will help clarify this. Consider the program `3^`. The function `^` needs two inputs but there is only one element in the stack (namely, number 3 at stack position 1). Thus MATL needs to "use" stack position 0 as well, which triggers user input. The number or array introduced by the user, say x, will be placed *below* the original 3. So the result will be x^3. To compute 3^x with implicit input, a swap operation would be needed: `3w^` (and then it is `w` that triggers the implicit input); or explicit input could be used: `3i^` .

The program `+^` asks for three numbers x, y, z in that order, and computes z^{x+y}. It works as follows. The stack is initially empty. When interpreting `+`, MATL asks the user for two numbers, x and y. So the stack now contains x and y, bottom to top. After `+` the stack contains a single value, $x+y$. Then `^` asks for another value, z, to fill the position below $x+y$. So the stack now contains z, $x+y$. After `^` it

contains a single value, z^{x+y}. Therefore, the code is equivalent to `ii+iw^` (not to `iii+^`, as one might initially think).

The program `TF$t` will implicitly ask for two inputs, and then push a copy of the first, leaving the stack with three elements. Note that even though `t` here copies only one element, that element is *two* positions below the current bottom of the (empty) stack, and consequently two implicit inputs are required.

Unlike the implicit display function described earlier, the implicit input method is *not a function call*. It does not use or consume `$` or `#` specifications, and it doesn't affect clipboard M. It does produce an automatic copy of the input to (a new level of) clipboard G.

> It would be nice to have implicit input of numbers or strings depending on the function. However, this seems difficult to do. For example, the second argument to `sum`, `rand` or `num2str` may be a number or a string, and it can't be told which one would be appropriate.

7.4 Automatic file input and output

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 statement of the program is executed. (There is also a function for manually reading a file; see §8).

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).

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[2],

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

and most functions from a list by Mathworks[3].

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

All MATLAB functions supported by `bsxfun` (as well as some others) have automatic singleton expansion in MATL.

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, and their overwriting the function-input clipboard.

> Some functions I didn't include, and the reasons why:
>
> - `true`, `false` functions; they can be done with `zeros` and `ones` (which are 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`.
> - `isempty`: I removed it because it can be done easily with `numel` followed by logical negation.
> - `length`. I followed `@chappjc`'s advice[4]: "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`.
> - `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.

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

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

Table 3: Table of MATL statements

		X	Y	Z
	separator			
!	.' (transpose) / permute	rot90	system	full
"	for	repmat	repelem (run-length decoding)	blanks
#	specify outputs	display stack (debug)		fopen, fwrite, fclose
$	specify inputs		char(vpa(...))	fopen, fread, fclose
%	comment	class	cast	typecast
&		intersect	and	bitand
'	*Not used. String delimiter*		run-length encoding	now / clock
(	( ) assignment indexing / split	{ } assignment indexing	( ) assignment ind. with final : / split	( ) assignment ind. with initial : / split
)	( ) reference indexing	{ } reference indexing	( ) reference ind. with final :	( ) refererence ind. with initial colon
*	.*	kron	matrix product	Cartesian product
+	+	conv	conv2	conv2(..., 'valid')
,	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	predefined literals	
3	*Not used*	predefined literals	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)	linearize array	comma-separated list	bitset
;		acos	asin	atan2
<	<	min	cummin	
=	==	isequal	strcmp	strncmp
>	>	max	cummax	
?	if			sparse
@	push "for" value / "while" index		perms	randperm
A	all	all(..., 1)	dec2base. Larger base, any symbols	base2dec. Larger base, any symbols
B	logical(dec2bin(...)-'0')	bin2dec(char(...+'0'))	dec2bin	bin2dec
C		histcounts	im2col	im2col(..., 'distinct')
D	disp(num2str(..., ...))	disp(num2str(...))	sprintf / fprintf	disp
E	multiply by 2	replace elements in array		
F	*Not used. False (literal)*		format	
G	Paste from clipboard G (user-input)	plot	imwrite / imagesc / image / imshow	control appearance of graphics
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 clipboard L (multi-level)	Copy to clipboard L (multi-level)	gallery	
M	Paste from clipboard M (function-input)			
N	stack size		NaN	isnan
O	zeros	datestr	datenum	datevec
P	flip	flipud	pi	pdist2
Q	increment by 1	accumarray		polyval
R	triu	triu(...,1)	tril	tril(...,-1)
S	sort	sortrows	circshift	sign
T	*Not used. True (literal)*		toeplitz	
U	str2num	str2double		
V	num2str			
W				
X	*Not used*	regexp	regexprep	
Y	*Not used*		inf	isinf
Z	*Not used*			
[	*Not used. Array delimiter*	ind2sub		
\	mod	mod(...-1)+1	matrix \	
]	end (loops or conditional branches)	sub2ind		
^	.^	sqrt	matrix ^	
_	unary minus			
`	do...while	while	tic	toc
a	any	any(..., 1)	padarray	
b	bubble		strsplit	
c	char	cat	strcat	strjoin
d	diff	diag	blkdiag	gcd
e	reshape / 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 / floor	upper / ceil		
l	ones		log. With two inputs, specifies base	log2
m	ismember	ismember(...,'rows')	mean	lcm
n	numel	nchoosek	interp1	norm
o	double	int64	round	fix
p	prod	prod(..., 1, ...)	cumprod	isprime / totient function
q	decrement by 1	quantile	n-th prime / next prime	primes
r	rand	randn	randi	randsample
s	sum	sum(..., 1, ...)	cumsum	std
t	duplicate elements			strrep
u	unique	unique(...,'rows')		strjust
v	vertcat	remove all blanks	strtrim	deblank
w	swap			
x	delete from stack	clc		
y	duplicate element	eye	hypot	size
z	nnz	nonzeros		
{	*Not used. Cell array delimiter*	num2cell	mat2cell	mat2cell(x,ones(size(x,1),1),size(x,2))
|	abs	union	or	bitor
}	else		cell2mat	split array
~	Not	setxor	xor	bitxor / bitcmp

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, consumption of input/output specifications and overwriting of function-input clipboard

	Consume inputs	Consume input/ output specifications	Overwrite function-input clipboard
Normal	Yes	Yes	Yes[a]
Meta	Yes	No	No
Stack rearranging	No	Yes	No
Clipboard	No	Yes	No

[a] If the function call has inputs

- 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?
- `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.
- `convmtx`: `convmtx(x,n)` can be realized as `conv2(x, eye(n))`.

9 The MATL compiler

9.1 Usage

The official MATL compiler is written in MATLAB R2015b. It takes a source program in MATL and produces an output (compiled) program that is run in MATLAB. It requires version R105b or newer, although most of the functionality probably works in older versions too. The compiler can also work on Octave; see §10. In addition, an online compiler is available; see §11.

The compiler 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. Implicit statements (see §7) are also indicated, as comments only.

  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`; `CB_G`; `CB_M`).

  (For debugging see also function `X#`, which displays the current stack contents).
- `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.
- `o`: online. This option is used in the online compiler. It produces the following changes:
  - removes the default input prompts;
  - doesn't generate file `defout`;
  - forbids functions that read files or URL's;
  - forbids functions that write to files;
  - forbids functions that evaluate arbitrary MATLAB functions or system commands.

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`, `d`);

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 space followed by 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 I4+`: runs the program `3,4+` 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 '1t8:"2$t+'`: 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.

```
1        % number literal
t        % duplicate
8        % number literal
:        % vector of equally spaced values
"        % for
  2      % number literal
  \      % modulus after division (element-wise, singleton expansion)
  $      % input specification
  t      % duplicate
  +      % addition (element-wise, singleton expansion)
         % (implicit) end
         % (implicit) convert to string and display
```

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

- `matl -h sort` would produce the result shown in Figure 2.

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 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.

10 Compatibility with Octave

The compiler can also run on Octave, which is a free alternative to MATLAB. Version 4.0.0 is required, although most of the functionality probably works in older versions too.

Although MATLAB and Octave have a large degree of similarity, compatibility between them is not total. This has some implications that are described in the following.

10.1 Compiler

The debug mode of the compiler uses MATLAB's openvar function to visualize the stack, clipboards and input/poutput specifications. This can't be done in Octave, because openvar doesn't exist.

Octave allows the " symbol as string delimiter, in addition to '. This implies that MATL programs that include " need to be enclosed in quotation marks, in addition to those that include '. Note that MATL only uses ' as string delimiter.

An internal aspect of the compiler is that Octave randomly initializes the random number generators automatically, so the compiled code doesn't need to do it.

10.2 MATL functions

The definitions of MATL functions are based on MATLAB. If there are differences between a MATLAB function and the corresponding Octave function, the corresponding MATL function is defined following that of MATLAB. This means that the compiler should reproduce that behaviour even when working on Octave, in order to achieve consistent results of MATL programs.

The choice of MATLAB over Octave, admittedly arbitrary, is based on the fact that the author has more experience with MATLAB, and by no means is meant to suggest any of them is better than the other. In any case, differences between MATLAB and Octave functions are few and small, so a programmer used to Octave will not have difficulties using MATL functions.

The compiler should ensure consistent behaviour of MATL programs regardless of the underlying platform being MATLAB or Octave. In the cases when inconsistencies between MATLAB and Octave have been identified, the compiler has been programmed to correct them, if feasible. The exceptions are cases where

the correction would be too difficult or when the difference is unimportant (for example, because it occurs using a function with inputs for which behaviour is undocumented).

The way the compiler deals with differences between MATLAB and Octave is as follows. If an Octave function `foo` behaves differently from the corresponding MATLAB function, the compiled file `MATLc.m` in Octave, which is a function, includes a subfunction that intercepts Octave's `foo` (only within `MATLc.m`). This subfunction applies the needed modifications, and usually includes a call to Octave's original `foo` via a `builtin` statement. This approach implies that the definitions in the MATL function definition file (`funDef.txt`) are the same for MATLAB and Octave. These definitions assume MATLAB; any deviations that might be caused by Octave are dealt with by the compiler directly.

Similarly, if a MATLAB function doesn't exist in Octave, the compiler includes it as a subfunction in the compiled file.

These compatibility functions are defined in separate files in a *compatibility folder*, called `compatibility`. Each file is read as text by the compiler, if needed, to insert it as a subfunction in the compiled code.

The following list describes the cases where a discrepancy between a MATLAB and an Octave function has been detected, and indicates if the compiler addresses this or not.

- `num2str` behaves differently in Octave and in MATLAB[5]. The compiler tries to achieve consistent behaviour, adapting the compiled code to give the same output on MATLAB and on Octave in the most common cases.

- `im2col` behaves differently in Octave and in MATLAB when the block size exceeds the array size. For example, `im2col(1:8, [2 1])` outputs `[]` in MATLAB, and gives an error in Octave.

  `im2col` allows char input in MATLAB, but not in Octave.

  Another difference exists regarding 3D input arrays. For example, `im2col(cat(3, magic(3), -magic(3)), [1 2])` gives different output in MATLAB and in Octave. Apparently MATLAB collapses all dimensions of the input array beyond the first.

  The above have been fixed. The compiler enforces MATLAB's behaviour when working in Octave.

- MATLAB's `im2col` ignores the third and subsequent components of the second input: `im2col(1:8, [1 2 3 4])` gives the same result as `im2col(1:8, [1 2 3 4])`. On the other hand, Octave gives an error.

  With the `'distinct'` option the behaviour is reversed: MATLAB gives an error, and Octave produces a result filled with zeros.

[5] http://stackoverflow.com/q/34483961/2586922

These differences aren't addressed in the compiler, because they are cases that are probably outside the intended use of `im2col`.

- `spiral` doesn't exist in Octave. The compiler defines it as part of the compiled code if needed.

- `triu`, `tril` with '[]' and a nonzero number as inputs give an error in Octave. This has been corrected so that they return '[]' as in MATLAB.

- `unique` in Octave behaves differently than in recent MATLAB versions (it behaves like in old MATLAB versions). For the purposes of the compiler this function is modified so that it finds the first occurrence of each element, not the last; and second and third outputs are always column vectors.

  Support for the `'stable'` flag has been added, only in the single-output case.

- In `union`, `intesect`, `setdiff`, `setxor`: support for the `'stable'` flag has been added, only in the single-output case.

- The second output of `ismember` in Octave gives the index of the first occurrence, not the last as in recent versions of MATLAB. Also, it gives error when the first two inputs are char and numeric, whereas in MATLAB it works. Both issues are fixed.

- `randsample` with scalar first input produces a row vector in Octave, instead of a column vector as in MATLAB. This has been fixed in the compiler, forcing a column vector in Octave in that case.

- Octave's `nchoosek`, unlike MATLAB's, requires first input to be numeric. This has been fixed, so that it can be a one-dimensional character array or cell array.

- Octave's `vpa` displays unwanted output. This hasn't been solved; however, it only happens in the first call to a symbolic function.

- When using Octave's `vpa`, conversion from symbolic to char looses precision. This is solved by converting the output of Octave's `vpa` to string using `pretty` (instead of `char`) and removing unwanted whitespace. Also, in Octave trailing decimals equal to zero are produced. This has been fixed for real numbers.

- Octave's `vpa` seems to have trouble with long strings containing non-integer numbers. For example, compare the output of `vpa('1/1234567891011')` with that of `vpa('1/.1234567891011')`. This isn't addressed by the compiler.

- `sum` and `mean` don't support the `'omitnan'` in Octave. This has been added.
- `diff` and `mod` don't allow char inputs in Octave. This has been fixed.
- Null assignment with less indices than dimensions works differently in MATLAB and in Octave. The code `x = cat(3,[1 2; 3 4],[1 2; 3 4]); x(:,2) = [];` produces the result `[1 1 2; 3 3 4]` in MATLAB (trailing dimensions are collapsed), but gives `cat(3, [1;3], [1;3])` in Octave (no collapsing). This difference isn't addressed by the compiler.
- `dec2bin` and `dec2base` don't allow char inputs in Octave. This has been fixed.
- `hypergeom` doesn't exist in Octave. This is partly addressed by the compiler. The function is computed either by direct sum (when some of the numerator coefficients is a non-positive integer, in which case the sum is finite) or using a third-party function[6], which only works for certain ranges of input values.
- Octave's `disp`, unlike MATLAB's, produces some output for empty input. This has been fixed.
- Octave's `str2func` works in the form `y = str2func('max')`, but not in the form `y = str2func('@(x) max(x)')`, unlike MATLAB's. This has been fixed. (This function is necessary for MATL's equivalent to `accumarray`.)
- In Octave arrays of type `char` cannot be converted to `logical`. This has been fixed.
- The three-input version of `circshift` doesn't exist in Octave. Also, Octave requires, unlike MATLAB, that the number of elements of the second input doesn't exceed the number of dimensions of the first. This has been fixed.

11 MATL online

MATL code can be executed online at `@Dennis`' awesome *Try it online!* platform[7]. The online compiler runs on Octave. Text output is displayed normally, and graphical output is displayed (with limitations) by Octave's fancy ASCII graph capabilities.

[6] http://www.mathworks.com/matlabcentral/fileexchange/5616-generalized-hypergeometric-function

[7] http://matl.tryitonline.net

12 Example programs explained

In the following, the examples given in §1.3 are explained. This serves to illustrate some of the features of MATL. See §8 and §A for the detailed definition of the functions involved.

12.1 Example 1: infinite loop

Code: `‘T]`.

This is a “do-while” loop. This type of loop is always entered at least once. Within the loop, `T` pushes a `true` literal to make sure the loop condition is met. This literal is consumed and the program proceeds with the next iteration, indefinitely.

The code could be shortened to `‘T`, because MATL automatically closes loops at the end of the program if they have not been previously closed.

12.2 Example 2: first 10 Fibonacci numbers

Code: `1t8:"2$t+`.

The first statement, `1`, pushes number 1 onto the stack. `t` makes a copy of it, so the stack now contains `1`, `1`. Statement `8` pushes an 8. Function `:` by default takes one input, and transforms the top of the stack, which is `8`, into vector `[1 2 3 4 5 6 7 8]`. So the stack now contains, bottom to top: `1`, `1`, `[1 2 3 4 5 6 7 8]`.

Statement `"` begins a “for” loop applied on vector `[1 2 3 4 5 6 7 8]`. Thus there will be 8 iterations, each corresponding to an entry of this vector. Since the vector is consumed by `"`, at the beginning of the first iteration the stack contains `1`, `1`. The loop body begins with `2$`, which specifies that the next function will use the top 2 elements as inputs. Thus the following `t` copies those 2 elements, and then `+` computes their sum to produce `2`.

Note that the `]` that closes the `"` loop is closing missing. This is allowed, because it is implicitly inserted by MATL at the end of the program.

So at the end of the first iteration the stack contains `1`, `1`, `2`. The second iteration again operates on the top 2 elements, `1`, `2`, to produce their sum `3`; and so on. The initial `1`, `1` remain on the stack as the first 2 members of the Fibonacci sequence, and the 8 loop iterations produce the next 8 Fibonacci numbers, to complete the total of 10.

At the end of the program, function `XD` is implicitly called. By default this function displays all elements in the stack, from bottom to top.

12.3 Example 3: unique characters from string in original order

Code: `j1X02$u`.

`j` inputs a string (without any prompt string by default). `X0` is a predefined-literal function; for input `1` it produces the string `'stable'` (see §A). `2$` specifies that the next function will use 2 inputs.

Note that literal `2$` is separated from the "0" in function `X0`. This is because when MATL encounters an "X" it reads the next character to form a two-character statement, and then proceeds reading a new statement. So no comma is needed here between "0" and "2".

The last statement is function `u`, which corresponds to `unique`. This function takes as inputs the string typed by the user and the `'stable'` flag to produce the desired result, which is displayed by the implicit final call to `XD`.

12.4 Example 4: unique characters from string in original order, manual approach

Code: `jtt!=XRa~)`. This produces the same result as in the previous example, but without using `u`.

`j` inputs a string. This is duplicated twice by `tt`, and the last copy is transposed by `!`. Function `=` consumes both copies (normal and transposed) and compares them element-wise with singleton expansion, producing a logical matrix. Entry (m,n) of this matrix is `true` if and only if the m-th character of the input string equals the n-th character. Thus the matrix is symmetrical and contains `true` values along the diagonal. The stack now conbtains the original string and the comparison matrix.

`XR` corresponds to `triu(..., 1)`, and thus keeps the part of the matrix above the diagonal, making all other entries `false`. This will assure that only duplicates with respect to *preceding characters* are detected.

`a` corresponds to `any`, with 1 input by default. The result is a logical row vector with a `true` at entry m if and only if the upper-triangular matrix produced by the previous function (`XR`) has at least a `true` value at column m; that is, if that character was a duplicate of a preceding character. Function `~` negates this logical vector, so that `true` indicates characters that should be kept. The stack now contains the original string and this logical vector. Function `)`, which takes 2 inputs by default, indexes the string with the logical vector to produce the desired result, which will be implicitly displayed.

13 Acknowledgments

Thanks to the following people (in alphabetical order) for their contributions:

- `@AndrasDeak` for his very extensive testing of the compiler.
- `@beaker` for a helpful discussion[8] that led me to reconsider including a form of `end`-based indexing, for telling me about Octave's `repelems` function, and for pointing out several errata.
- `Ben Barrowes` for his function `genHyper`.
- `@David` for finding a good implementation of the "n-th prime" function, as well as some errata in this document.
- `@Dennis` for hosting MATL in his wonderful *Try it online!* platform[9].
- `@excaza` for his suggestion regarding the hypergeometric function.
- `@flawr` for his suggestions, including the "n-th prime" function and implicit input of data, and for finding several bugs.
- `@pragmatist1` for suggesting the use of a regular expression to check contents of array literals.
- `@RainerP` for a helpful discussion related to "if" and the automatic function clipboard.
- `@rayryeng` for coming up with the initial idea[10] and for his invaluable help in getting MATL online. Also for his suggestion to include predefined constants, for reading an early version of this document and suggesting changes, and for his support with this project.

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

[8] http://chat.stackoverflow.com/transcript/message/26114026#26114026
[9] http://tryitonline.net
[10] http://chat.stackoverflow.com/transcript/message/25584017#25584017

`!`	1–2 (1)	1	With 1 input: `.'` (`transpose`), or `permute(..., [2 1 ...])` for multidimensional arrrays. With 2 inputs: `permute`
`X!`	1–2 (2)	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)
`Z"`	1	1	`blanks`
`X#`	0	0	display stack as a cell array (for debugging)
`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.
`X$`	1– (2)	0– (1)	execute Matlab function specified by first input, using the rest of the inputs as arguments.
`Y$`	1–2 (2)	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`
`X&`	2–4 (2)	1–3 (1)	`intersect`. Uses the `'stable'` flag by default. If one input is char and the other is numeric, the latter is converted to char
`Y&`	1– (2)	1	`&` (`and`), element-wise with singleton expansion
`Z&`	2–3 (2)	1	`bitand`, element-wise with singleton expansion. If first input is `char` it is automatically converted to `double`
`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 (0)	1	`now`. With 1 input: the input should be numeric with values from 1 to 6, which are used as indices into the output of `clock`
`(`	3– (3)	1	assignment `( )` indexing. Null assignment (`x(...) = []`) can only be done with a single index
`X(`	3– (3)	1	assignment `{ }` indexing
`Y(`	2– (3)	1	assignment `(..., :)` indexing. Null assignment (`x(..., :) = []`) can only be done with a single index (in addition to the colon)

`Z(`	2– (3)	1	assignment `(:, ...)` indexing. Null assignment (`x(:, ...) = []`) can only be done with a single index (in addition to the colon)
`)`	2– (2)	1–2 (1)	reference `( )` indexing. If 2 outputs: only one input index can be used. The second output produces the "complementary" array `y=x; y(ind)=[]`, where `y` and `ind` are the inputs
`X)`	2– (2)	1	reference `{ }` indexing
`Y)`	1– (2)	1–2 (1)	reference `(..., :)` indexing. If 2 outputs: only one input index can be used. The second output produces the "complementary" array `y=x; y(ind,:)=[]`, where `y` and `ind` are the inputs
`Z)`	1– (2)	1–2 (1)	reference `(:, ...)` indexing. If 2 outputs: only one input index can be used. The second output produces the "complementary" array `y=x; y(:,ind)=[]`, where `y` and `ind` are the inputs
`*`	1– (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 a number *n* of arrays of possibly different sizes, generates an *n*-column matrix whose rows describe all combinations of elements taken from those arrays
`+`	1– (2)	1	`+` (`plus`), element-wise with singleton expansion
`X+`	2–3 (2)	1	`conv`. Converts first two inputs to `double`. *See also* `Y+`, `Z+`
`Y+`	2–4 (2)	1	`conv2`. Converts first two inputs to `double`. *See also* `X+`, `Z+`
`Z+`	2–3 (2)	1	`conv2(..., 'valid')`. Converts first two inputs to `double`. *See also* `X+`, `Y+`
`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`. Uses the `'stable'` flag by default. If one input is char and the other is numeric, the latter is converted to char
`Y.`	0–1 (1)	0	`pause` (without outputs)
`Z.`	2–3 (2)	1	`bitget`. If first input is `char` it is automatically converted to `double`
`/`	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
`Y5`	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`
`X:`	1	1	linearize to column array (index with `(:)`)
`Y:`	1	0– (△)	generate comma-separated list from cell array (`{:}`) and push each element onto the stack
`Z:`	2–4 (3)	1	`bitset`. If first input is `char` it is automatically converted to `double`
`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`. If first or second inputs are numeric they are converted to char
`Z=`	3	1	`strncmp`. If first or second inputs are numeric they are converted to char
`>`	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`
`Y?`	0	1	
`Z?`	1–6 (3)	1	`sparse`
`Y@`	1	1	`perms`
`Z@`	1–3 (1)	1	`randperm` (produces a row vector as output). With 3 outputs: third output indicates number of permutations, each in a different row.
`A`	1–2 (1)	1	`all`. *See also* `XA`
`XA`	1	1	`all(..., 1)`. *See also* `A`

`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`	2–4 (2)	1	`im2col`. If the second input is a scalar n, it is transformed into [1 n] if the first input is a row vector, or to [n 1] otherwise. *See also* `ZC`
`ZC`	2–3 (2)	1	`im2col(..., 'distinct')`. If the second input is a scalar n, it is transformed into [1 n] if the first input is a row vector, or to [n 1] otherwise. *See also* `YC`
`D`	0– (1)	0	If 1 input: `disp(num2str(..., '%.15g '))`. 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, '%.15g '))`, 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(...)`. *See also* `D`, `XD`, `ZD`
`ZD`	0– (1)	0	`disp` for each input. *See also* `D`, `XD`, `YD`
`E`	1	1	`(...)*2`
`XE`	3	1	replace in first input all occurrences of each element of the second input by the corresponding element of the third input. Output has the same class and size as the first input
`YF`	0–1 (1)	0	`format`
`G`	0–1 (␣)	0– (⌐)	paste from user-input clipboard G. With 0 input arguments: addresses all levels. With 1 input argument: addresses level specified by the input
`XG`	1– (1)	0	`plot`. Calls `drawnow` to update figure immediately

`YG`	2– (2)	0	`imwrite`, `imagesc`, `image` or `imshow`. If last input is a scalar: `0` corresponds to `imwrite`, `1` to `imagesc`, `2` to `image` and `3` to `imshow`. The corresponding function is called with the remaining inputs. If last input is numeric and not a scalar: `imshow` is called with all inputs. If last input is char: `imshow` is called with all inputs. For `imagesc` and `image`, the function call is followed by `axis ij, axis image`. For `imagesc`, `image` and `imwhow`, `drawnow` is called to update figure immediately
`ZG`	1– (2)	0–1 (0)	Depending on numeric first input, calls a graphic function with the remaining inputs. 1: `axis`. 2: `colormap`. With 0 outputs, calls `drawnow` to update figure immediately. 3: `hold`
`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`	1	0– (†)	paste from multi-level clipboard L. Input specifies level
`XL`	1– (2)	0	copy to multi-level clipboard L. Topmost input specifies level
`YL`	1– (2)	1– (1)	`gallery`. Also includes functions `magic`, `hilb`, `invhilb`, `hadamard`, `pascal`, `spiral`
`M`	1	0– (*)	paste from function-input clipboard M. Input specifies level (1 to 4) or individual input (5 or larger)
`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`. *See also* `XP`
`XP`	1	1	`flipud`. *See also* `P`
`YP`	0	1	`pi`
`ZP`	2–5 (2)	1	`pdist2`. Only predefined distance functions are allowed
`Q`	1	1	`(...)+1`

XQ	2–6 (2)	1	accumarray. Optional fourth argument is a function string, which will be transformed into function handle. Function strings can be of the form 'max' (name of a function), '@(x)max(x)' (anonymous function defined in terms of x), or 'max(x)' (if the string contains (and doesn't begin by @, '@(x)' is automatically prepended)
ZQ	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 an array (single-array mode: sort) / sort an array based on another (two-array mode). Single-array mode works like Matlab's sort. If 2 inputs, a negative value of the second input corresponds to descending order. In two-array mode, this function takes first 2 inputs as equal-length vectors where the second is not char. The second vector is sorted and its order is applied to the first; and an optional third input specifies direction as a string, or as a negative number in the non-singleton dimension of the second vector. The outputs are the two sorted arrays. (In two-array mode, if the two input arrays are scalar the result is the same as if the second input is interpreted as dimension, corresponding to single array mode)
XS	1–2 (1)	1–2 (1)	sortrows
YS	2–3 (2)	1	circshift. If second input is a scalar, the shift is applied along the first non-singleton dimension
ZS	1	1	sign
YT	1–2 (2)	1	toeplitz
U	1	1–2 (1)	str2num with content checking. If input is not char, it is automatically converted to char
XU	1	1	str2double
V	1–2 (1)	1	num2str
XX	2–9 (2)	1–6 (♢)	regexp. With 2 inputs: regexp(..., ..., 'match'). If first or second inputs are numeric they are converted to char
YX	3–5 (3)	1	regexprep. If first, second or third inputs are numeric they are converted to char
YY	0– (0)	1	inf function. If 0 inputs: produces literal inf.
ZY	1	1	isinf
X[	2	1– (2)	ind2sub
\	2	1–2 (1)	mod, element-wise with singleton expansion. With 2 outputs: second output is 'floor(.../...)'.
X\	2	1	mod(...-1)+1, element-wise with singleton expansion

`Y\`	2	1	left matrix division, `\` (`mldivide`)
`X]`	3– (3)	1	`sub2ind`
`^`	2	1	`.^` (`power`), element-wise with singleton expansion
`X^`	1	1	`sqrt`
`Y^`	2	1	`^` (`mpower`)
`Z^`	2	1	Cartesian power. Given a number *n* and an arrays, computes the Cartesian power of the array times itself *n* times. *See also* `Z*`
`_`	1	1	unary `-` (`uminus`)
`Y‘`	0	0–1 (0)	`tic`
`Z‘`	0–1 (0)	0–1 (1)	`toc`
`a`	1–2 (1)	1	`any`. *See also* `Xa`
`Xa`	1	1	`any(..., 1)`. *See also* `a`
`Ya`	2–4 (2)	1	`padarray`. It allows the first input to be `char`; and then the output is also `char`. If the second input is `logical` or the pad value is `char` they are converted to `double`
`b`	0– (3)	0	bubble up element in stack
`Yb`	1– (1)	1–2 (1)	`strsplit`. If second input is a numeric vector, it is converted to char
`c`	1– (1)	1	`char`
`Xc`	3– (3)	1	`cat`
`Yc`	1– (2)	1	`strcat`. Numeric inputs are converted to char
`Zc`	1–2 (1)	1	`strjoin`. If second input is numeric it is converted to char
`d`	1–3 (1)	1	`diff`
`Xd`	1–2 (1)	1	`diag`
`Yd`	1– (2)	1	`blkdiag`
`Zd`	1–2 (2)	1–3 (1)	`gcd`, element-wise with singleton expansion. With 1 input and 1 output, computes the greatest common divisor of all elements of the input
`e`	1– (3)	1	With more than 1 input: `reshape`. With 1 input: `squeeze`.
`Ze`	1	1	`exp`
`f`	1–3 (1)	1–3 (1)	`find`
`Xf`	2–4 (2)	1	`strfind`. Works also with numeric arrays or cell arrays of numeric arrays. In this case each numeric array is linearized, and results are row vectors
`Yf`	1	1	`factor`. For negative input, it computes the factors of the abolute value and removes 1 from result.
`g`	1	1	`logical`
`Xg`	1– (2)	1– (□)	`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` for strings or cell arrays of strings; `floor` for numerical arrays
`Xk`	1	1	`upper` for strings or cell arrays of strings; `ceil` for numerical arrays
`l`	0– (0)	1	`ones` (if 0 inputs: produces output 1)
`Yl`	1–2 (1)	1	`log`. If 2 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`	1–2 (2)	1	`lcm`, element-wise with singleton expansion. With 1 input, computes the least common multiple of all elements of the input
`n`	1	1	`numel`
`Xn`	2	1	`nchoosek`
`Yn`	1–5 (2)	1	`interp1`. `'pp'` option not supported
`Zn`	1–2 (1)	1	`norm`
`o`	1	1	`double`
`Xo`	1	1	`int64`
`Yo`	1–3 (1)	1	`round`
`Zo`	1	1	`fix`
`p`	1–3 (1)	1	`prod`. If first input is `char` it is converted to `double`. *See also* `Xp`
`Xp`	1–3 (1)	1	`prod(..., 1, ...)`. If first input is `char` it is converted to `double`. *See also* `p`.
`Yp`	1–3 (1)	1	`cumprod`. Allows char input
`Zp`	1	1	For input with positive entries: `isprime`. For input with non-positive entries: Euler's totient function for absolute value of each entry
`q`	1	1	`(...)-1`
`Xq`	2–3 (2)	1	`quantile`
`Yq`	1	1	For input with positive entries: n-th prime for each value n in the input array. For input with non-positive entries: next prime for absolute value of each entry
`Zq`	1	1	`primes`
`r`	0– (0)	1	`rand`

`Xr`	0– (0)	1	`randn`
`Yr`	1– (1)	1	`randi`
`Zr`	2–4 (2)	1	`randsample`. Does not support stream specification
`s`	1–4 (1)	1	`sum`. *See also* `Xs`
`Xs`	1–3 (1)	1	`sum(..., 1, ...)`. *See also* `s`
`Ys`	1–3 (1)	1	`cumsum`. Allows char input
`Zs`	1–4 (1)	1	`std`
`t`	0– (1)	0	duplicate elements in stack. The duplicated elements are those specified as inputs
`Zt`	3	1	`strrep`. The first input can be a numeric array (not a cell array of numeric arrays). In this case the other inputs can be char or numeric (not cell); each input array is linearized; result is a row vector; and result is char if third input is, even if not substitutions have been actually done
`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`
`Xv`	1	1	remove space from input string or cell array strings. If input is a numeric array, it is converted to char
`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`	0– (2)	0	duplicate one element in stack. The duplicated element is the lowest among those specified as inputs
`Xy`	1–4 (1)	1	`eye`
`Yy`	2	1	`hypot`, element-wise with singleton expansion
`Zy`	1–2 (1)	1– (1)	`size`
`z`	1	1	`nnz`
`Xz`	1	1	`nonzeros`
`X{`	1–2 (1)	1	`num2cell`
`Y{`	2– (3)	1	`mat2cell`
`Z{`	1	1	`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
`|`	1	1	`abs`
`X|`	2–4 (2)	1–3 (1)	`union`. Uses the `'stable'` flag by default. If one input is char and the other is numeric, the latter is converted to char
`Y|`	1– (2)	1	`|` (`or`), element-wise with singleton expansion
`Z|`	2–3 (2)	1	`bitor`, element-wise with singleton expansion. If first input is `char` it is automatically converted to `double`

`Y}`	1	1	`cell2mat`
`Z}`	1–2 (1)	0– (▽)	split array into elements in linear order. With 2 inputs: split into subarrays along the dimension indicated by the second input
`~`	1	1	`~` (`not`)
`X~`	2–4 (2)	1–3 (1)	`setxor`. Uses the `'stable'` flag by default. If one input is char and the other is numeric, the latter is converted to char.
`Y~`	2	1	`xor`, element-wise with singleton expansion
`Z~`	1–3 (2)	1	`bitxor`, element-wise with singleton expansion. With 1 numeric input (and optionally a second string input): `bitcmp`. In both cases, if first input is `char` it is automatically converted to `double`

† Number of elements in clipboard (H, I, J, K), or in clipboard level (L).

‡ Number of elements in stack.

* Number of elements in clipboard level for combined addressing; or 1 for individual addressing.

△ Number of elements of input cell array.

▽ Number of elements or subarrays that will be produced.

⊔ 0 if clipboard currently has 0 or 1 levels; 1 otherwise.

⊓ Number of levels addressed according to input specification.

◊ According to specified keywords.

□ Number of inputs.

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'	6	'descend'
10	'first'	11	'forward'	12	'legacy'
13	'ascend'				

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	'CollapseDelim iters'	2	'DelimiterType'	3	'RegularExpression'
4	'center'	5	'left'	6	'local'
10	'right'				

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

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'
20	'distinct'	21	'sliding'	22	'indexed'

Function X6					
1	'get'	2	'post'	3	'Timeout'
4	'png'	5	'ppm'	6	'gif'
7	'bmp'	9	'none'	11	'Frames'
12	'BackgroundColor'	13	'Index'	14	'Info'
15	'ReductionLevel'	16	'PixelRegion'	17	'V79Compatible'
20	'Border'	21	'Colormap'	22	'DisplayRange'
23	'InitialMagnification'	24	'Parent'	25	'Reduce'

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'		

Function X9					
1	'%.15g '	10	'seed'		

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'
10	'manual'	11	'tight'	12	'fill'
13	'ij'	14	'xy'	15	'equal'
16	'image'	17	'normal'	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	'@(x){x}'	5	'@(x){x.'')}'	6	'{sort(x)}'
7	'{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'	5	['0':'9' 'A':'F']	6	' ':'~'
11	'aeiou'	12	'AEIOU'	13	'aeiouAEIOU'
20	['Mon';'Tue';'Wed'; 'Thu';'Fri';'Sat'; 'Sun']	21	[298 302 288 305 289 296 310]		

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

Function `Y3`					
1	'magic'	2	'hilb'	3	'invhilb'
4	'hadamard'	5	'pascal'	6	'spiral'
10	'binomial'	11	'cauchy'	12	'chebspec'
13	'chebvand'	14	'chow'	15	'circul'
16	'clement'	17	'compar'	18	'condex'
19	'cycol'	20	'dorr'	21	'dramadah'
22	'fiedler'	23	'forsythe'	24	'frank'
25	'gcdmat'	26	'gearmat'	27	'grcar'
28	'hanowa'	29	'house'	30	'integerdata'
31	'invhess'	32	'invol'	33	'ipjfact'
34	'jordbloc'	35	'kahan'	36	'kms'
37	'krylov'	38	'lauchli'	39	'lehmer'
40	'leslie'	41	'lesp'	42	'lotkin'
43	'minij'	44	'moler'	45	'neumann'
46	'normaldata'	47	'orthog'	48	'parter'
49	'pei'	50	'poisson'	51	'prolate'
52	'qmult'	53	'randcolu'	54	'randcorr'
55	'randhess'	56	'randjorth'	57	'rando'
58	'randsvd'	59	'redheff'	60	'riemann'
61	'ris'	62	'sampling'	63	'smoke'
64	'toeppd'	65	'toeppen'	66	'tridiag'
67	'triw'	68	'uniformdata'	69	'wathen'
70	'wilk'				

Function `Y4`					
1	'\d+'	2	'[+-]?\d+(\.\d*)?'	3	'[A-Z]'
4	'[a-z]'	5	'[A-Za-z]'		