9 A VIM Execution Engine

by Chris Domas

The power of vim is known far and wide, yet it is only when we push the venerable editor to its limits that we truly see its beauty. To conclusively demonstrate vim's majesty, and silence heretical doubters, let us construct a copy/paste/search/replace Turing machine, using vanilla vim commands.

First, we lay some ground rules. Naturally, we could build a Turing machine using the built-in vimscript, but it is already known that vimscript is Turing-complete, and this is hardly sporting. vim ex commands (the requests we make from vim when we type a colon) are abundant and powerful, but these too would make the task simple, and therefore would fail to illustrate the glory of vim. Instead, we strive to limit ourselves to normal vim commands - yank, put, delete, search, and the like.

With these constraints in mind, we must decide on the design of our machine. For simplicity, let us implement an interpreter for the widely known BrainFuck (BF) programming language. Our machine will be a simple text file that, when opened in vim and started with a few key presses, interprets BF code through copy/paste/search/replace style vim commands.

Let us begin by giving our machine some memory. We create data tape in the text file by simply adding the following:

```
_t:
0 0 0 0 0 0 0 0 0 0
```

We now have ten data cells, which we can locate by searching for `_t`.

Now what of the BF code itself? Let us add a Fibonacci number generator to the file:

```
_p:
>++++++++++>+>+[[+++++[>++++++++
<-]>.<++++++[>--------<-]+<<<]>.
>>[[-]<[>+<-]>>[<<+>+>-]<[>+<-[>
+<-[>+<-[>+<-[>+<-[>+<-[>+<
-[>+<-[>[-]>+>+<<<-[>+<-]]]]]]]]
]]]+>>>]<<<]
```

Progress! Now we add lines to accommodate input and output, although these will be left empty for now:

```
_i:

_o:
```

To perform output, our program will need to convert the numeric memory cells to ASCII values. This can easily be done by adding an ASCII lookup table to our program:

```
_a:
... __65 A__66 B__67 C__68 D ... _127 ._uuu
 .
```

The arrangement of underscores and spaces will assist us in navigating the table with vim commands. Providing an "unknown" `uuu` allows us to process values outside the ASCII range.

Now for the fun part—how do we execute our BF program using just our simple vim commands? We would envision a small set of commands running continuously to interpret the program. Of course, we could manually type out these commands ourselves, over and over, to perform the execution (and we indeed encourage this as an enjoyable exercise!), but in the unfortunate situation in which an interpreted program fails to halt, we may come to find this process laborious. Instead, we will insert the keys for these commands directly into our vim file. When complete, we can automatically run the commands on the first line of the file by typing:

```
ggyy@"
```

If the first line, in turn, moves to other lines, and repeats this process of yanking a line of commands (yy) and executing the yanked buffer (`@"`), execution can continue indefinitely, without any additional user action.

N.B.T.V.A.

The Narrow Bandwidth TV Association (founded 1975) is dedicated to low definition and mechanical forms of ATV and introduces radio amateurs to TV at an inexpensive level based on home-brew construction. NBTV should not be confused with SSTV which produces still pictures at a much higher definition. As TV base bandwidth is only about 7kHz, recording of signals on audiocassette is easily achieved. A quarterly 12-page newsletter is produced and an annual exhibition is held in April/May in the East Midlands. If you would like to join, send a crossed cheque/postal order for £4 (or £3 plus a recent SPRAT wrapper) to Dave Gentle, G4RVL, 1 Sunny Hill, Milford, Derbys, DE56 0QR, payable to "NBTVA".

So to begin, let us simplify the process of navigating the text file by setting marks at key points. At the start of our text file, we add commands to set a mark at the beginning of the file:

```
gg0mh
```

A mark at the memory tape:

```
/_t^Mnjmt‘h
```

A mark at the BF code:

```
/_p^Mnjmp‘h
```

A mark at the input, output, and ASCII table:

```
/_o^Mnjmo‘h/_i^Mnjmi‘h/_a^Mnjma‘h
```

Although these steps are not strictly necessary, they will simplify navigating the file for future commands.

Now for execution! BF contains 8 instructions: increment the current data cell (`+`), decrement the current data cell (`-`), move to the next data cell (`>`), move to the previous data cell (`<`), a conditional jump forward (`[`), a conditional jump backward (`]`), output the current data cell (`.`), and input to the current data cell (`,`). Let us construct a table of vim commands to carry out each of these operations; each label will act as a marker for looking up the corresponding commands:

```
_c:
_>-???X
_<-???X
_[-???X
_]-???X
_+-???X
_--???X
_.-???X
_,-???X
_f:_???X
_b:_???X
```

We again apply the trick of special characters around each operation to simplify the search process—we may find many `>`'s in our file, but there will be only one `_>-`. We mark the end of the command with an X. We preemptively supply additional `_f` and `_b` commands, to carry out the conditional part of the BF branch operations `[` and `]`. We will determine the exact commands for each momentarily, which will replace the unknown `???` above. For now, let us continue the previous process of adding marks to each for quick navigation.

```
/_c^Mnjma‘h/_c^Mnf_mf‘h/_b^Mnf_mb
```

Now that our marks are set, we add to the top of our file the commands to execute the first instruction in the BF program:

```
‘pyl‘c/_\V^R"^Mf–ly2tX@"
```

This will move to the BF program (‘p), yank one BF instruction (yl), move to the command table (‘c), find the BF instruction in the table, `(/_\V^R"^M)` move to the list of commands for that instruction (f-l), yank the list of commands (y2tX)—skipping an X embedded in the command, and seeking forward to the terminating X—and execute the yanked commands (@"). With this, our execution begins!

Let's now complete our table by determining the commands to execute each BF instruction. `>` and `<` are particularly simple. For `>`:

```
‘twmt‘p mpyl‘c/_\V^R"^Mf–ly2tX@"
```

Plainly, this is: move to the memory tape (‘t), move forward one memory cell (w), mark the new location in the tape (mt), move back to the BF program (‘p), move forward one character to progress over the now executed BF instruction ( ), mark the new location in the BF program (mp), yank the next BF instruction (yl), and follow the previous process `(‘c/_\V^R"^Mf-ly2tX@")` to locate that instruction in the command table, yank its commands, and execute them.

`<`, then, is similarly implemented as:

```
‘tbmt‘p mpyl‘c/_\V^R"^Mf–ly2tX@"
```

What of `+` and `-`? `+` can be performed with:

```
‘t^A‘p mpyl‘c/_\V^R"^Mf–ly2tX@"
```

This is virtually identical to the `<` and `>` implementation. This time, we move to the current data cell and increment it with ^ A. Strictly speaking, this is a violation of the copy/paste/search/replace type execution we have been using. However, with minimal effort, the increment could be performed via a lookup table (as we do for the ASCII conversion)—we simply elide this for brevity.

Simply replacing ^ A (increment) with ^ X (decrement), `-` is derived:

```
‘t^X‘p mpyl‘c/_\V^R"^Mf–ly2tX@"
```

Now, certainly, our interpreter is not useful without input and output, so let us add `.` and `,` commands. `.` may be

```
‘tyw‘a/_\(^R"\|uuu\)^Mellyl‘op$mo‘p mpyl‘c/_
    \V^R"^Mf–ly2tX@"
```

This of course is: move to the memory tape (‘t), yank a cell (yw), move to the ASCII table (‘a), search for the yanked cell or, if it is not found, move to the `uuu` marker, `(/_\(^R"\|uuu\)^M)`, move over the marker characters (ell), yank the corresponding ASCII character (yl), move to the output (‘o), paste the ASCII character (p), move to the end of the output ($), mark the new output location (mo), and finally, move back to the BF program, move over the executed instruction, grab the next instruction, locate its commands, and execute them, as before.

```
(‘p mpyl‘c/_\V^R"^Mf–ly2tX@")
```

Data input with `,` is similarly:

```
‘iy  mi‘a/ ^R"_^MT_ye‘txt p‘p mpyl‘c/_\V^R"^
    Mf–ly2tX@"
```

Which simply performs the reverse lookup and stores the result in the current memory cell.

We are close, but, alas!, nothing is ever simple, and BF's conditional looping becomes more complicated. The BF `[` instruction means precisely "*if the byte at the data pointer is zero, then instead of moving the instruction pointer forward to the next command, jump it forward to the command after the matching* `]` *command.*"

```
‘tyt  ‘f/\(^R"\|n\)x^Mf–ly2tX@"
```

Meaning, navigate to the memory tape (‘t), yank a memory cell (yt), navigate to the forward assist commands (‘f), search for either the yanked cell, or, if it is not found, the character n, followed by x `(/\(^R"\|n\)x^M)`, and yank and execute the given commands, using the process as before `(f-ly2tX@")`. This search allows us to achieve the conditional portion of the `[` instruction—we will include a marker for only "0", so only a memory cell of "0" will find a match—all others will be directed to the "n" character. Our forward assist then appears as:

```
_f:_0x:–‘p% mpyl‘c/_\V^R"^Mf–ly2tX@"X_nx:–‘p
    mpyl‘c/_\V^R"^Mf–ly2tX@"X
```

If the memory cell is 0, the previous search matches _0x, and the commands following it are yanked and executed. If the memory cell is not 0, the previous search matches _nx, and the commands following it instead are yanked and executed. For 0, we have: go to the BF program (‘p), navigate to the corresponding `]` instruction (%), move to the instruction after this ( ), mark the new location in the program (mp), and then yank and execute the next instruction, as before. `(yl‘c/_\V^R"^Mf-ly2tX@")` For non-0, we have: go to the BF program (‘p), navigate to the next instruction ( ), mark the new location in the program (mp), and then yank and execute the next instruction, as before. `(yl‘c/_\V^R"^Mf-ly2tX@")`

`]` is now straightforward. Following the same patterns, we have:

```
‘tyt  ‘b/\(^R"\|n\)x^Mf–ly2tX@"
```

for the conditional search, and

```
_b:_0x:–‘p mpyl‘c/_\V^R"^Mf–ly2tX@"X_nx:–‘p%
    mpyl‘c/_\V^R"^Mf–ly2tX@"X
```

as the backward assist commands. An ardent observer may argue the the vim % command violates our copy/paste/search/replace design, and, alas!, this is so. However, we argue that a series of searches, increments, and decrements—like those

```
             :%s/\^A/\="\<C-A>"/g|%s/\^X/\="\<C-X>"/g|%s/\^R/\="\<C-R>"/g|%s/\^M/\n/g|06
             0f-ly$@"
                                           ### launch with gg2yy@" ###
                                           ###### @xoreaxeaxeax ######

_c:          _s1-gg0mh`h/_t^Mnjmt`h/_p^Mnjmp`h/_o^Mnjmo`h/_i^Mnjmi`h/_s2^Mnf-ly$@"njmt_j
             _s2-`h/_a^Mnjma`h/_c^Mnf:mc`h/_f^Mnf_mf`h/_b^Mnf_mb`pyl`c/_\V^R"^Mf-ly2tX@"
              z_>-`twmt`p mpyl`c/_\V^R"^Mf-ly2tX@"Xs_<-`tbmt`p mpyl`c/_\V^R"^Mf-ly2tX@"X
             _f:_0x:-`p% mpyl`c/_\V^R"^Mf-ly2tX@"Xa_nx:-`p mpyl`c/_\V^R"^Mf-ly2tX@"Xmpyl
             _b:_0x:-`p mpyl`c/_\V^R"^Mf-ly2tX@"Xm_nx:-`p% mpyl`c/_\V^R"^Mf-ly2tX@"Xly2t
             _+-`t^A`p mpyl`c/_\V^R"^Mf-ly2tX@"Xo_--`t^X`p mpyl`c/_\V^R"^Mf-ly2tX@"X_/--
             _]-`tyt `b/\(^R"\|n\)x^Mf-ly2tX@"Xd_[-`tyt `f/\(^R"\|n\)x^Mf-ly2tX@"X^$0x:-
             _v.$7yy_.-`tyw`a/_\(^R"\|uuu\)^Mellyl`op$mo`p mpyl`c/_\V^R"^Mf-ly2tX@"Xelly
             _$`p mpy`pyl`a_,-`iy  mi`a/ ^R"_^MT_ye`tvt p`p mpyl`c/_\V^R"^Mf-ly2tX@"X_#-
_o:


_i:
100^M

_t:
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

_a:
___0 .___1 .___2 .___3 .___4 .___5 .___6 .___7 .___8 .___9 .__10 ^M_11 .__12 .__13 .__14 .__15 ._
__16 .__17 .__18 .__19 .__20 .__21 .__22 .__23 .__24 .__25 .__26 .__27 .__28 .__29 .__30 .__31 ._
__32  __33 !__34 "__35 #__36 $__37 %__38 &__39 `__40 (__41 )__42 *__43 +__44 ,__45 -__46 .__47 /_
__48 0__49 1__50 2__51 3__52 4__53 5__54 6__55 7__56 8__57 9__58 :__59 ;__60 <__61 =__62 >__63 ?_
__64 @__65 A__66 B__67 C__68 D__69 E__70 F__71 G__72 H__73 I__74 J__75 K__76 L__77 M__78 N__79 O_
__80 P__81 Q__82 R__83 S__84 T__85 U__86 V__87 W__88 X__89 Y__90 Z__91 [__92 \__93 ]__94 ^__95 __
__96 `__97 a__98 b__99 c_100 d_101 e_102 f_103 g_104 h_105 i_106 j_107 k_108 l_109 m_110 n_111 o_
_112 p_113 q_114 r_115 s_116 t_117 u_118 v_119 w_120 x_121 y_122 z_123 {_124 |_125 }_126 ~_127 ._
_uuu .

_p:
+[->,----------[<+>-------------------------------------->[>+>+<<-]>>[<<+>>-]<>>>+++++++++[<<<[>+
>+<<-]>>[<<+>>-]<[<<+>>-]>>-]<<<[-]<<[>+<-]]<]>>[<<+>>-]<<>+<-[>+[>+>+<<-]>>[<<+>>-]<>+<-->>>>>>>
>+<<<<<<<<[>+<-<[>>>+>+<<<<-]>>>>[<<<<+>>>>-]<<<>[>>+>+<<<-]>>>[<<<+>>>-]<<<<>>>[>+>+<<-]>>[<<+>>
-]<<<[>>>>>+<<<[>+>+<<-]>>[<<+>>-]<[>>[-]<<-]>>[<<<<[>+>+<<-]>>[<<+>>-]<>>>-]<<<-<<-]+>>[<<[-]>>-
]<<>[-]<[>>>>>>>[-]<<<<<<<-]<<>>[-]>[-]<<<]>>>>>>>>>[-<<<<<<<<[-]<<[>>+>+<<<-]>>>[<<<+>>>-]<<<>>[>+<-
]>[[>+>+<<-]>>[<<+>>-]<>++++++++++<[>>>+<<[>+>[-]<<-]>[<+>-]>[<<++++++++++>>-]<<-<-]+++++++++>[<->
-]<[>+<-]<[>+<-]<[>+<-]>>>[<<<+>>>-]<>++++++++++<[>>>+<<[>+>[-]<<-]>[<+>-]>[<<++++++++++>>>+<-]<<-
<-]>>>>[<<<<+>>>>-]<<<<>[-]<<+>]<[[>+<-]+++++++++[<+++++++++>-]<-><.[-]>>[<<+>>-]<<-]>+++++[<++++++++++>
-]<.[-]>>>>>>>>]<<<<<<<<<>[-]<[-]<<-]++++++++++.[-]#
```

Figure 20 – VIM Execution Engine

we have already shown - could be used to implement %'s functionality in a more perfect manner; we leave this as an exercise for the purists.

But lo! With the implementation of the 8 BF instructions, our execution engine is complete! Figure 20 shows a cleanly formatted version of the final machine. The demonstration machine uses our copy/paste/search/replace commands to calculate the prime numbers up to 100. For ease of use, we add an introductory %s search and replace sequence—momentarily allowing ourselves to enter ex commands—in order to insert the control characters (^ M, ^ R, etc.) needed throughout the rest of the machine. This provides us a pure-ASCII file, without the need to enter special characters. Simply copy the below, paste into vanilla vim, launch with `gg2yy@"`, and witness the awesome Turing-complete power of our benevolent editor![54]

[54] `unzip pocorgtfo12.pdf vimmmex.tar.gz`
`git clone https://github.com/xoreaxeaxeax/vimmmex`