

The **runner** package: interpreters for various languages*

Bruno Le Floch

2013/04/20

Contents

1	runner documentation	2
2	The brainfuck language	4
3	The forth language	5
3.1	Useful variables	8
A	runner implementation	8
A.1	Variables and constants	9
A.2	Providing arrays for the interpreters	10
A.3	Output from programs	11
A.4	Misc	11
A.5	Running programs	11
A.6	Keys	13
A.7	User command	13
A.8	Messages	13
B	brainfuck implementation	14
B.1	Variables	14
B.2	Current cell	14
B.3	Structure to run the code	15
B.4	The eight language commands	16

*This file has version number 0.0a, last revised 2013/04/20.

C	forth implementation	18
C.1	Memory, dictionary and stacks	19
C.1.1	Data space and data stack	19
C.1.2	Other stacks	22
C.1.3	Dictionary	22
C.2	Helpers	24
C.2.1	Input	24
C.2.2	Numbers	25
C.2.3	Misc	26
C.3	Core words	26
C.3.1	Definition words	26
C.3.2	Execution words	30
C.3.3	Stack words	31
C.3.4	Comparison words	33
C.3.5	Arithmetic words	34
C.3.6	Storage words	38
C.3.7	Words for display, input, and strings	41
C.3.8	Conditional words	44
C.3.9	Looping words	45
C.3.10	Ending words	46
C.3.11	Environmental queries	46
C.3.12	Misc words	48
C.4	Running the interpreter	48
C.5	Messages	49

1 runner documentation

This package aims to provide interpreters for various languages, with code written for all commonly used modern \TeX engines (\pdf\TeX , \Xe\TeX , and \Lua\TeX). Currently, the following languages are supported:

- `brainfuck`,
- `forth`,

and more will be added as I get time.

\runner The main command, `\runner`, expects four arguments (or three, since the second is optional):

1. The name of a programming language supported by `runner` (see list above), surrounded by curly braces.
2. An optional list of key–value pairs, surrounded by square brackets, which affect how the program is run, where the input is taken from, and where its output goes.
3. The program code, given as a verbatim argument (like the argument of `\verb`), *i.e.*, it must be surrounded by two identical characters which do not appear in the program.
4. The input, also a verbatim argument, delimited by two identical characters (I advise to use the same as those surrounding the program).

The program code is interpreted in the given language, and is given the last argument as its input. For instance, since the brainfuck program “`,+[-.>,+]`” outputs its input,

```
\runner{brainfuck}@,+[-.>,+]@@some input@
```

will yield “`some input`”. This text will be left to be processed by `TEX` (as if it were typed directly in the document). To store the result for later reuse, use the `output` key.

output Using the `output` key lets us redirect the output of a program to a macro, which can later be used. Keeping with the same brainfuck program, we can do for instance

```
\runner{brainfuck}[output = \result]@,+[-.>,+]@@some input@
\typeout{\result}
```

to store “`some input`” in the macro `\result`, then print it to the terminal using the `LATEX 2ε` command `\typeout`.

input **Not implemented.** A program can take its input from a macro that is already defined. Note that `\runner` still takes four arguments, and that the last argument, normally used for the input, must be empty, lest there be an error.

```
\newcommand{\someinput}{some nice text}
\runner{brainfuck}[input = \someinput]@,+[-.>,+]@@@
```

leaves “`some nice text`” for `TEX` to process.

```

\NewRunner
\RenewRunner
\DeclareRunner

```

Not implemented.

```

\NewRunner{\bfcopy}{brainfuck}@,+[-.>,+]@
\bfcopy @some input@
\bfcopy[output = \result]@other input@

```

will leave “some input” to be processed by T_EX, then store “other input” in the macro `\result`.

2 The brainfuck language

This `brainfuck` implementation has an unbounded array, both on the left and right, and each cell can hold arbitrary integer values.¹ The brainfuck language has eight language commands.

- > Increment the cell pointer.
- < Decrement the cell pointer.
- + Increment the value stored in the current cell.
- Decrement the value stored in the current cell.
- . Output the ascii character corresponding to the value at the pointer. Values outside the range of available characters ([0, 255] or [0, 1114111]) raise an error.
- , Read the first character of the input and store its character code in the current cell. The end of the input string is marked by the value `-1`.
- [If the value at the current cell is zero, jump to the matching]
-] If the value at the current cell is non-zero, jump to the matching [.

Every other character is ignored.

For example, here is a “Hello World!” program (from the Wikipedia page on `brainfuck`). This program takes no input, hence we feed it an empty input.

```

\runner{brainfuck}[output = \result]@‘‘Hello World!’’ program
+++++ +++++
  [ >+++++++ >+++++++ >+++ >+ < < < < -]
>++.
>+.
++++++.
.
+++
>++.

```

¹More precisely, the size of the array is limited to the range $[-2^{31} + 1, 2^{31} - 1]$, like other T_EX integers, and the number of non-zero cells must be at most 30000 or so.

```

< < ++++++.
>.
+++
-----
-----
>+.
>.
@@@
\typeout{\result}

```

An example which takes an input and copies it reversed to the output:

```

\runner{brainfuck}[output = \result]@->,+<+[--.<+]@@Some input.@
\typeout{\result}

```

3 The forth language

This is not the place for an introduction to Forth, many can be found online.

For instance, one can define the factorial function as follows.

```

\runner{forth}[output = \result]|
: FACTORIAL ( +n1 -- +n2)
  DUP 2 < IF DROP 1 ELSE ( 0! = 1! = 1)
  DUP 1- RECURSE * THEN ; ( n1! = n1 * [n1 - 1]!)
||
CR 0 FACTORIAL . ( prints 1)
CR 3 FACTORIAL . ( prints 6)
CR 24 FACTORIAL . ( prints 12582912)|
\typeout{\result}

```

This types three lines to the terminal, containing 1, 6 and 12582912. The last number is incorrect, because the result overflows the size of a single-cell unsigned integer. We can easily check that it is the correct result modulo 2^{24} .

The aim is to make of this **forth** interpreter a “Standard System” as defined by the ANS Forth 1994 specification. This is not yet the case. The following words are not implemented:

- definitions CREATE, DOES>, >BODY,
- control-flow BEGIN, UNTIL, WHILE, REPEAT
- do-loop +LOOP,
- parse/input >IN, ACCEPT, SOURCE
- ending ABORT, ABORT", QUIT.
- number conversion <#, #, #>, #S, >NUMBER, HOLD, SIGN

- int FM/MOD, M*, S>D, SM/REM, U., U<, UM*, UM/MOD
- bitwise AND, INVERT, LSHIFT, OR, RSHIFT, XOR
- misc EVALUATE, EXIT, FIND, KEY, MOVE, STATE, [,]

The following environment variables are not implemented: /COUNTED-STRING, /HOLD, /PAD, RETURN-STACK-CELLS, STACK-CELLS.

Implementation specificities are as follows.

- All addresses are aligned and are character-aligned.
- Behaviour of EMIT for non-graphic character: ?
- Character editing of ACCEPT: ?
- Character set: ?
- Character-set-extensions matching characteristics: ?
- Conditions under which control characters match a space delimiter: ?
- Format of the control-flow stack: ?
- Conditions under which control characters match a space delimiter: ?
- Format of the control-flow stack: ?
- Conversion of digits larger than thirty-five: ?
- Display after input terminates in ACCEPT: ?
- Exception abort sequence (as in ABORT"): ?
- Input line terminator: ? User input device: ?
- Maximum size of a counted string, in chars: ?
- Maximum size of a parsed string: ?
- Maximum size of a definition name, in chars: ?
- Maximum string length for ENVIRONMENT?, in chars: ?
- Method of selecting a user input device: ?
- Method of selecting a user output device: ?
- Methods of dictionary compilation: ?
- Number of bits in one address unit: 24.
- Number representation and arithmetic: internally represented as 24-bit unsigned integers (well, further down as \TeX dimensions in a `\fontdimen` array).

- Integers $[-2^{23}, 2^{23} - 1]$, non-negative integers $[0, 2^{23} - 1]$, unsigned integers $[0, 2^{24} - 1]$, double integers $[-2^{47}, 2^{47} - 1]$, non-negative double integers $[0, 2^{47} - 1]$, unsigned double integers $[0, 2^{48} - 1]$.
- Read-only data-space regions.
- Size of buffer at `WORD`.
- One cell is one address unit.
- One character is one address unit.
- Size of the keyboard terminal input buffer: ?
- Size of the pictured numeric output string buffer: ?
- Size of the scratch area whose address is returned by `PAD`: ?
- Case-sensitivity: ?
- Prompt (see `QUIT`): ?
- Division rounding: symmetric (the quotient is rounded towards zero, the remainder has the sign of the numerator).
- Values of `STATE` when true: ?
- Values returned after arithmetic overflow: value modulo 2^{24} .
- Whether the current definition can be found after `DOES>`: ?

Here is a list of data types and their sizes in this implementation.

Symbol	Data type	# bits
flag	flag (true or false)	24
char	character	24
n	signed number	24
+n	non-negative number	24
u	unsigned number	24
x	unspecified cell	24
xt	execution token	24
addr	address (= a-addr = c-addr)	24
d	double-cell signed number	48
+d	double-cell non-negative number	48
ud	double-cell unsigned number	48
xd	unspecified cell pair	48
colon-sys	definition compilation	dep
do-sys	do-loop structures	dep
case-sys	CASE structures	dep
of-sys	OF structures	dep
orig	control-flow origins	dep
dest	control-flow destinations	dep
loop-sys	loop-control parameters	dep
nest-sys	definition calls	dep
i*x, j*x, k*x (3)	any data type	≥ 0

3.1 Useful variables

`\g_runner_languages_clist`

The list of known languages.

A runner implementation

Some support packages are loaded first, then declare the package's name, date, version, and purpose.

```

<*package>
1 \RequirePackage{expl3}[2012/12/07]
2 \RequirePackage{l3str}[2012/12/07]
3 \RequirePackage{xparse}[2012/12/07]
4 \ProvidesExplPackage
5   {runner} {2013/04/20} {0.0a} {Interpreters for various languages}
6 <@@=runner>

```


A.1 Variables and constants

<code>\g_runner_languages_clist</code>	<p>The list of known languages.</p> <pre> 7 \clist_new:N \g_runner_languages_clist 8 \clist_gset:Nn \g_runner_languages_clist 9 { 10 brainfuck, 11 forth, 12 }</pre> <p>(End definition for <code>\g_runner_languages_clist</code>. This variable is documented on page 8.)</p>
<code>\l__runner_input_str</code> <code>\l__runner_output_str</code> <code>\l__runner_program_str</code>	<p>Input and output common to all the programming languages. This assumes that there is no need to allow cross-language mixing.</p> <pre> 13 \tl_new:N \l__runner_input_str 14 \tl_new:N \l__runner_output_str 15 \tl_new:N \l__runner_program_str</pre> <p>(End definition for <code>\l__runner_input_str</code>, <code>\l__runner_output_str</code>, and <code>\l__runner_program_str</code>.)</p>
<code>\l__runner_tmpa_tl</code> <code>\l__runner_tmpp_tl</code> <code>\l__runner_tmppc_tl</code> <code>\l__runner_tmppd_tl</code> <code>\l__runner_tmppa_fp</code> <code>\l__runner_tmppa_int</code> <code>\l__runner_tmppb_int</code> <code>\l__runner_tmppc_int</code> <code>\l__runner_tmppd_int</code> <code>\l__runner_tmppa_bool</code>	<p>Scratch variables used by various languages.</p> <pre> 16 \tl_new:N \l__runner_tmpa_tl 17 \tl_new:N \l__runner_tmpp_tl 18 \tl_new:N \l__runner_tmppc_tl 19 \tl_new:N \l__runner_tmppd_tl 20 \fp_new:N \l__runner_tmppa_fp 21 \int_new:N \l__runner_tmppa_int 22 \int_new:N \l__runner_tmppb_int 23 \int_new:N \l__runner_tmppc_int 24 \int_new:N \l__runner_tmppd_int 25 \bool_new:N \l__runner_tmppa_bool</pre> <p>(End definition for <code>\l__runner_tmppa_tl</code> and others.)</p>
<code>__runner_tmp:w</code>	<p>Short-term auxiliary, defined on the fly.</p> <pre> 26 \cs_new_eq:NN __runner_tmp:w ?</pre> <p>(End definition for <code>__runner_tmp:w</code>.)</p>
<code>\l__runner_output_key_tl</code>	<p>This token list is empty if the output should be left in the input stream, and otherwise contains the macro in which the output should be stored.</p> <pre> 27 \tl_new:N \l__runner_output_key_tl</pre> <p>(End definition for <code>\l__runner_output_key_tl</code>.)</p>
<code>\g__runner_array_font_size_int</code>	<p>The size of the last font that was used as an array.</p> <pre> 28 \int_new:N \g__runner_array_font_size_int</pre> <p>(End definition for <code>\g__runner_array_font_size_int</code>.)</p>

A.2 Providing arrays for the interpreters

`_runner_array_new:Nn` Initialize an array (implemented as a font) #1 of size #2. If #2 is negative or zero, complain (we could allow zero if someone asks). Otherwise, allocate a new font, which is T_EX's `\tenrm` at a tiny size which no one would use, and zero out the 7 first parameters, to which T_EX gives values. We make sure to change size every time, otherwise we would be back with the same array. Entries go from 1 to the array size, which we do not store: the caller is responsible for keeping track of this information, and out-of-bound addressing leads to low-level T_EX errors.

```

29 \cs_new_protected:Npn \_runner_array_new:Nn #1#2
30 {
31   \cs_new_eq:NN #1 \tex_nullfont:D
32   \int_compare:nNnTF {#2} > \c_zero
33   {
34     \int_gincr:N \g_runner_array_font_size_int
35     \tex_global:D \tex_font:D #1 = cmr10~at~
36     \g_runner_array_font_size_int sp \scan_stop:
37     \int_step_inline:nnnn { 1 } { 1 } { 7 }
38     { \tex_fontdimen:D ##1 #1 = \c_zero_dim }
39     \tex_fontdimen:D \int_eval:n {#2} #1 = \c_zero_dim
40   }
41   {
42     \msg_error:nnxx { runner } { negative-array-size }
43     { \token_to_str:N #1 } { \int_eval:n {#2} }
44   }
45 }

```

(End definition for `_runner_array_new:Nn`.)

`_runner_array_item:Nn` The value stored in a given cell is retrieved by converting the `\fontdimen` dimension to a number in sp. We provide a fast version expecting the item number to be an integer.

```

46 \cs_new:Npn \_runner_array_item:Nn #1#2
47 { \tex_number:D \tex_fontdimen:D \int_eval:n {#2} #1 }
48 \cs_new:Npn \_runner_array_item:NN #1#2
49 { \tex_number:D \tex_fontdimen:D #2 #1 }

```

(End definition for `_runner_array_item:Nn` and `_runner_array_item:NN`.)

`_runner_array_gset:Nnn` Assignments are always global.

```

50 \cs_new_protected:Npn \_runner_array_gset:Nnn #1#2#3
51 {
52   \tex_fontdimen:D \int_eval:n {#2} #1
53   = \int_eval:n {#3} sp \scan_stop:
54 }

```

(End definition for `_runner_array_gset:Nnn`.)

A.3 Output from programs

`__runner_output:n` Add a string to the right of the output string.
`__runner_output:x`

```

55 \cs_new_protected:Npn \__runner_output:n #1
56   { \str_put_right:Nn \l__runner_output_str {#1} }
57 \cs_generate_variant:Nn \__runner_output:n { x }

```

(End definition for __runner_output:n and __runner_output:x.)

`__runner_output_char:n` Add a character with arbitrary character code to the right of the output.

```

58 \group_begin:
59   \char_set_catcode_other:N \^^@
60   \cs_new_protected:Npn \__runner_output_char:n #1
61   {
62     \group_begin:
63     \char_set_lccode:nn {0} {#1}
64     \tl_to_lowercase:n
65     { \group_end: \str_put_right:Nn \l__runner_output_str { ^^@ } }
66   }
67 \group_end:

```

(End definition for __runner_output_char:n.)

A.4 Misc

`__runner_break:n` A tool to jump to the end of a loop, leaving the appropriate clean-up code (argument of
`__runner_break_point:n` `__runner_break_point:n`).

```

68 \cs_new:Npn \__runner_break:n #1 #2 \__runner_break_point:n #3 { #3 #1 }
69 \cs_new_eq:NN \__runner_break_point:n \use:n

```

(End definition for __runner_break:n and __runner_break_point:n.)

A.5 Running programs

`__runner:nnnn` Four arguments: mandatory programming language name, optional key-values, verbatim program code, verbatim input. First make sure that the language is loaded. If this succeeded, then, in a group, set the various keys that affect the run, set the program, input, and output, and call the language-specific runner. Once this ends, clean up with `__runner_finish:w`, which also takes care of sending the output outside the group.

```

70 \cs_new_protected:Npn \__runner:nnnn #1#2#3#4
71   {
72     \__runner_load_language:nT {#1}
73     {
74       \group_begin:
75       \keys_set:nn { runner } {#2}
76       \str_set:Nn \l__runner_program_str {#3}
77       \str_set:Nn \l__runner_input_str {#4}
78       \tl_clear:N \l__runner_output_str
79       \use:c { __runner_#1_run: }
80       \__runner_finish:w
81     }

```

```

82     }
83   }
(End definition for \_runner:nnnn.)

```

`_runner_load_language:nTF` If the runner command for the language #1 is already defined, nothing needs to be done. Otherwise, load the ldf file if possible. This file may be defective (if it does not define the runner command), or inexistent, and we raise the appropriate error in those cases.

```

84 \prg_new_conditional:Npnn \_runner_load_language:n #1 { T }
85 {
86   \cs_if_exist:cTF { \_runner_#1_run: }
87   { \prg_return_true: }
88   {
89     \file_if_exist:nTF { runner - #1 .ldf }
90     {
91       \group_begin:
92       \ExplSyntaxOn
93       \file_input:n { runner - #1 .ldf }
94       \group_end:
95       \cs_if_exist:cTF { \_runner_#1_run: }
96       { \prg_return_true: }
97       {
98         \msg_error:nnn { runner } { defective-ldf } {#1}
99         \prg_return_false:
100       }
101     }
102     {
103       \clist_if_in:NnTF \g_runner_languages_clist {#1}
104       { \msg_error:nnn { runner } { missing-ldf } {#1} }
105       { \msg_error:nnn { runner } { unknown-language } {#1} }
106       \prg_return_false:
107     }
108   }
109 }

```

(End definition for _runner_load_language:nTF.)

`_runner_finish:w` The output string `\l__runner_output_str` is either left for \TeX to digest, or stored in the macro contained in `\l__runner_output_key_tl`.

```

110 \cs_new_protected_nopar:Npn \_runner_finish:w \group_end:
111 {
112   \use:x
113   {
114     \group_end:
115     \tl_if_empty:NTF \l__runner_output_key_tl
116     { \l__runner_output_str }
117     {
118       \tl_set:Nn \exp_not:o \l__runner_output_key_tl
119       { \l__runner_output_str }
120     }
121   }

```

```

122 }
(End definition for \__runner_finish:w.)

```

A.6 Keys

```

123 \keys_define:nn { runner }
124 {
125     output .tl_set:N = \l__runner_output_key_tl
126 }

```

A.7 User command

`__runner_process_lang_name:n` Sanitize the language name passed to `\runner`, by turning it to a string, removing all spaces, and converting to lowercase. We use `\tl_expandable_lowercase:n` because it is not affected by external settings.

```

127 \cs_new_protected:Npn \__runner_process_lang_name:n #1
128 {
129     \str_set:Nn \l__runner_tmpa_tl {#1}
130     \tl_remove_all:Nn \l__runner_tmpa_tl { ~ }
131     \tl_set:Nx \ProcessedArgument
132         { \exp_args:NV \tl_expandable_lowercase:n \l__runner_tmpa_tl }
133 }
(End definition for \__runner_process_lang_name:n.)

```

`\runner` Four arguments: mandatory programming language name, optional key-values, verbatim program code, verbatim input. The programming language name is sanitized using `__runner_process_lang_name:n`. Pass everything to `__runner:nnnn`.

```

134 \NewDocumentCommand {\runner}
135 { > { \__runner_process_lang_name:n } m 0{ } +v +v }
136 { \__runner:nnnn {#1} {#2} {#3} {#4} }
(End definition for \runner. This function is documented on page 3.)

```

A.8 Messages

```

137 \msg_new:nnnn { runner } { unknown-language }
138 { The~programming~language~'#1'~is~not~known~to~'runner'. }
139 {
140     The~code\\\\
141     \iow_indent:n {#2}\\\\
142     could~not~be~run,~because~there~is~not~support~for~'#1',~yet.
143 }
144 \msg_new:nnnn { runner } { defective-ldf }
145 { The~file~'runner-#1.ldf'~does~not~define~a~programming~language. }
146 {
147     The~language~definition~file~'runner-#1.ldf'~must~define~
148     '\iow_char:N \__runner_#1_run:'~to~enable~the~runner~package~
149     to~run~programs~written~in~'#1'.
150 }
151 \msg_new:nnnn { runner } { missing-ldf }

```

```

152 { The~file~'runner-#1.ldf'~cannot~be~found. }
153 {
154   The~'#1'~programming~language~should~be~supported~by~the~
155   'runner'~package,~but~this~requires~the~language~definition~
156   file~'runner-#1.ldf',~which~is~nowhere~to~be~found.
157 }
158 \msg_new:nnnn { runner } { negative-array-size }
159 { The~array~'#1'~cannot~be~initialized~at~size~'#2'. }
160 {
161   This~is~probably~an~internal~error~in~the~runner~package.~
162   Please~report~it.
163 }

```

B brainfuck implementation

B.1 Variables

`\l__runner_brainfuck_cell_int` Pointers to the current cell, and to the current instruction in the program.

```

\l__runner_brainfuck_instr_int 164 \int_new:N \l__runner_brainfuck_cell_int
165 \int_new:N \l__runner_brainfuck_instr_int

```

`\l__runner_brainfuck_match_prop` The [and] commands jump between various program instructions. The property list
`\l__runner_brainfuck_match_seq` holds the matches (back and forth). When building the property list, we keep track of a stack with the positions of each [.

```

166 \prop_new:N \l__runner_brainfuck_match_prop
167 \seq_new:N \l__runner_brainfuck_match_seq

```

`\l__runner_brainfuck_length_int` Length of the program, once stripped of all comments.

```

168 \int_new:N \l__runner_brainfuck_length_int

```

B.2 Current cell

`__runner_brainfuck_current_name:` Shorthand for the name of the integer holding the value at the current cell.

```

169 \cs_new_nopar:Npn \__runner_brainfuck_current_name:
170 { l__runner_brainfuck\_int_use:N \l__runner_brainfuck_cell_int _int }

```

`__runner_brainfuck_current_provide:` The first time the value at a cell is changed (from its initial value of 0), an integer register is automatically created.

```

171 \cs_new_protected_nopar:Npn \__runner_brainfuck_current_provide:
172 {
173   \cs_if_exist:cF { \__runner_brainfuck_current_name: }
174   { \int_new:c { \__runner_brainfuck_current_name: } }
175 }

```

`__runner_brainfuck_current_value:` When the current cell's integer doesn't exist yet, the value is zero.

```

176 \cs_new_nopar:Npn \__runner_brainfuck_current_value:
177 {
178   \cs_if_exist:cTF { \__runner_brainfuck_current_name: }

```

```

179         { \int_use:c { \_runner_brainfuck_current_name: } }
180         { 0 }
181     }

```

B.3 Structure to run the code

_runner_brainfuck_run:

```

182 \cs_new_protected_nopar:Npn \_runner_brainfuck_run:
183 {
184     \_runner_brainfuck_clean:
185     \_runner_brainfuck_matches:
186     \_runner_brainfuck_execute:
187 }

```

_runner_brainfuck_clean:

_runner_brainfuck_clean_aux:n

The first step is to remove all comments, by keeping only characters that are meaningful in brainfuck. Then measure the length of the resulting program.

```

188 \cs_new_protected_nopar:Npn \_runner_brainfuck_clean:
189 {
190     \tl_set:Nx \l__runner_program_str
191     {
192         \tl_map_function:NN
193         \l__runner_program_str
194         \_runner_brainfuck_clean_aux:n
195     }
196     \int_set:Nn \l__runner_brainfuck_length_int
197     { \tl_count:N \l__runner_program_str }
198 }
199 \cs_new:Npn \_runner_brainfuck_clean_aux:n #1
200 { \cs_if_exist:cT { __runner_brainfuck_#1: } {#1} }

```

_runner_brainfuck_matches:

_runner_brainfuck_matches[:
_runner_brainfuck_matches:]

The second step is to go through the program and find how left and right brackets match.

```

201 \cs_new_protected_nopar:Npn \_runner_brainfuck_matches:
202 {
203     \int_set:Nn \l__runner_brainfuck_instr_int { 1 }
204     \tl_map_inline:Nn \l__runner_program_str
205     {
206         \cs_if_exist_use:c { __runner_brainfuck_matches_##1: }
207         \int_incr:N \l__runner_brainfuck_instr_int
208     }
209 }
210 \cs_new_protected_nopar:cpn { __runner_brainfuck_matches[: }
211 {
212     \seq_push:No \l__runner_brainfuck_match_seq
213     { \int_use:N \l__runner_brainfuck_instr_int }
214 }
215 \cs_new_protected_nopar:cpn { __runner_brainfuck_matches:] }
216 {
217     \seq_pop:NN \l__runner_brainfuck_match_seq \l__runner_tmpa_tl
218     \prop_put:NVV

```

```

219     \l__runner_brainfuck_match_prop
220     \l__runner_tmpa_tl
221     \l__runner_brainfuck_instr_int
222   \prop_put:NVV
223     \l__runner_brainfuck_match_prop
224     \l__runner_brainfuck_instr_int
225     \l__runner_tmpa_tl
226   }

```

`__runner_brainfuck_execute:` Finally, run the program. Each iteration of the `\int_while_do:nn` loop goes through the instructions one by one (with `\tl_map_inline:nn`), starting at instruction `\l__runner_brainfuck_instr_int`, and for each instruction performs the corresponding function `\use:c { @@_brainfuck_##1: }`. In the absence of `[or]`, the loop reaches the end of the program, and the while loop ends there. On the other hand, `[and]` may break the `\tl_map_inline:nn` loop, and set `\l__runner_brainfuck_instr_int` to some value. Then the while loop repeats its body, and we start reading instructions again at `\l__runner_brainfuck_instr_int`.

```

227 \cs_new_protected_nopar:Npn \__runner_brainfuck_execute:
228 {
229   \int_set:Nn \l__runner_brainfuck_instr_int { 1 }
230   \int_until_do:nn
231     { \l__runner_brainfuck_instr_int > \l__runner_brainfuck_length_int }
232   {
233     \exp_args:Nf \tl_map_inline:nn
234       {
235         \str_range:Nnn \l__runner_program_str
236           { \l__runner_brainfuck_instr_int }
237           { -1 }
238       }
239     {
240       \use:c { __runner_brainfuck_##1: }
241       \int_incr:N \l__runner_brainfuck_instr_int
242     }
243   }
244 }

```

B.4 The eight language commands

`__runner_brainfuck_>:` Moving to the right or left.

```

\__runner_brainfuck_>: 245 \cs_new_protected_nopar:cpn { __runner_brainfuck_>: }
246   { \int_incr:N \l__runner_brainfuck_cell_int }
\__runner_brainfuck_<: 247 \cs_new_protected_nopar:cpn { __runner_brainfuck_<: }
248   { \int_decr:N \l__runner_brainfuck_cell_int }

```

`__runner_brainfuck_+:` Increasing or decreasing the current value, after ensuring that the corresponding integer exists.

```

\__runner_brainfuck_+: 249 \cs_new_protected_nopar:cpn { __runner_brainfuck_+: }
250   {

```



```

251     \__runner_brainfuck_current_provide:
252     \int_incr:c { \__runner_brainfuck_current_name: }
253 }
254 \cs_new_protected_nopar:cpn { __runner_brainfuck_-: }
255 {
256     \__runner_brainfuck_current_provide:
257     \int_decr:c { \__runner_brainfuck_current_name: }
258 }

```

`__runner_brainfuck_.`: Output the character corresponding to the value at the current cell.

```

259 \cs_new_protected_nopar:cpn { __runner_brainfuck_.: }
260 { \__runner_output_char:n { \__runner_brainfuck_current_value: } }

```

`__runner_brainfuck_`: Read off one character from the input string, and store its character code in the current cell. As usual, ensure that the cell's integer is defined.

```

261 \cs_new_protected_nopar:cpn { __runner_brainfuck_,: }
262 {
263     \__runner_brainfuck_current_provide:
264     \tl_if_empty:NTF \l__runner_input_str
265     { \int_set_eq:cn { \__runner_brainfuck_current_name: } \c_minus_one }
266     {
267         \tl_set:Nx \l__runner_tmpa_tl
268         { \str_head:N \l__runner_input_str }
269         \tl_set:Nx \l__runner_input_str
270         { \str_tail:N \l__runner_input_str }
271         \int_set:cn { \__runner_brainfuck_current_name: }
272         { \exp_after:wN ' \l__runner_tmpa_tl }
273     }
274 }

```

`__runner_brainfuck_[:` The [and] operators have opposite logics. Both compare the value stored at the current cell to zero, and may jump to the matching instruction as defined by `\l__runner_brainfuck_match_prop`. In case of a jump, we break the current loop through program instructions.

```

275 \cs_new_protected_nopar:cpn { __runner_brainfuck_[: }
276 {
277     \int_compare:nNnT \__runner_brainfuck_current_value: = \c_zero
278     { \__runner_brainfuck_jump: }
279 }
280 \cs_new_protected_nopar:cpn { __runner_brainfuck_] : }
281 {
282     \int_compare:nNnF \__runner_brainfuck_current_value: = \c_zero
283     { \__runner_brainfuck_jump: }
284 }
285 \cs_new_protected_nopar:Npn \__runner_brainfuck_jump:
286 {
287     \prop_get:NVN
288     \l__runner_brainfuck_match_prop
289     \l__runner_brainfuck_instr_int

```

```

290     \l__runner_tmpa_t1
291     \int_set:Nn \l__runner_brainfuck_instr_int { \l__runner_tmpa_t1 }
292     \tl_map_break:
293 }

```

C forth implementation

Maybe relevant text in the 1994 standard.

This Standard designates the following words as obsolescent:

- 6.2.0060 #TIB
- 15.6.2.1580 FORGET
- 6.2.2240 SPAN
- 6.2.0970 CONVERT
- 6.2.2040 QUERY
- 6.2.2290 TIB
- 6.2.1390 EXPECT

[...]

A system need not provide any standard words for accessing mass storage. If a system provides any standard word for accessing mass storage, it shall also implement the Block word set. [...]

The standard says “char” is a subtype of “+n”, so characters are only allowed to take values in the range $[0, 2^{23} - 1]$? Addresses are only allowed to take values in the range $[-2^{23}, 2^{23} - 1]$?

Am I allowed to always consider the data-space pointer to be aligned, separating cells by 1 address unit?

Stacks.

- Data stack only available if control stack is empty.
- Control-flow stack contains only “-sys” types.
- Return stack for definitions, do-loops, nesting info. Also used by programs.

Dictionary.

- name space,
- code space,
- data space accessible to programs
 - contiguous regions,
 - variables,

- text-literal regions,
- input buffers,
- other transient regions,

Two useful variants.

```

294 \cs_generate_variant:Nn \prop_gput:Nnn { Nx }
295 \cs_generate_variant:Nn \prop_get:NnNTF { Nx }

```

C.1 Memory, dictionary and stacks

The data space is stored in $\text{T}_{\text{E}}\text{X}$'s `\fontdimen` array. In each slot of this array, we store a 24 bit unsigned integer, *i.e.*, a number in the range $[0, 2^{24} - 1]$ (note that we could go up to 26 bits with no adverse effect). Address units are 24 bits wide. Characters are one address unit wide (note that all Unicode code points are less than 2^{23}). Cells are one character wide. We do not use 8 bit characters (and addresses) for four reasons: only 256 cells could be addressed; counted strings would be bounded to 255 characters; extracting an 8-bit part from a 24 bit value in $\text{T}_{\text{E}}\text{X}$ is not fast; and we would need to chose an encoding for Unicode (this is still needed in $\text{pdfT}_{\text{E}}\text{X}$, but only at the very last step of output).

The end of the `\fontdimen` array is also used to store the data stack, with the bottom of the stack at the end of the array. The control flow parameters are also put on top of the data stack.

Execution tokens are cells restricted to the range $[0, 2^{16} - 1]$ or $[0, 2^{15} - 1]$ (depending on the engine's capabilities), which point to a `\toks` register containing the $\text{T}_{\text{E}}\text{X}$ code to perform the execution semantics.

The return stack also uses `\toks` registers, with the bottom of the stack at the highest register.

To each word is associated a digit in $\{1, 2, 3\}$ and an execution token, with the following behaviour,

- 1 indicates normal words, whose interpretation semantics are given by the execution token, and whose compilation semantics is to append the execution token to the current definition;
- 2 indicates immediate words, whose interpretation and compilation semantics are identical, and given by the execution token;
- 3 indicates special words such as `ABORT`", which may have arbitrary interpretation and compilation semantics: the execution token gives the interpretation semantics, and the compilation semantics follows in the next `\toks` register.

C.1.1 Data space and data stack

`\g__runner_forth_data_array` The data array contains both the data space and the data stack. The three integers give the size of the array, the first free address, and the top of the data stack: `data_here ≤ data_top` must always hold.

```

\g__runner_forth_data_size_int
\l__runner_forth_data_here_int
\l__runner_forth_data_top_int
296 \int_new:N \g__runner_forth_data_size_int

```

```

297 \int_gset:Nn \g__runner_forth_data_size_int { 65536 }
298 \int_new:N \l__runner_forth_data_here_int
299 \int_new:N \l__runner_forth_data_top_int
300 \__runner_array_new:Nn \g__runner_forth_data_array
301 \g__runner_forth_data_size_int

```

__runner_forth_put_here:n Put a value at the position given by the data_here integer, then increment that integer.

```

302 \cs_new_protected:Npn \__runner_forth_put_here:n #1
303 {
304   \__runner_array_gset:Nnn \g__runner_forth_data_array
305     { \l__runner_forth_data_here_int } {#1}
306   \int_incr:N \l__runner_forth_data_here_int
307 }

```

__runner_forth_pop_int:N Get data from the data stack. Note that things are popped backwards, so that the last argument gets the top of the stack.

```

\__runner_forth_pop_int:NN
\__runner_forth_pop_int:NNN
\__runner_forth_pop_int:NNNN
308 \cs_new_protected:Npn \__runner_forth_pop_int:N #1
309 {
310   \int_compare:nNnTF
311     \l__runner_forth_data_top_int < \g__runner_forth_data_size_int
312     {
313       \int_set:Nn #1
314       {
315         \__runner_array_item:NN \g__runner_forth_data_array
316         \l__runner_forth_data_top_int
317       }
318       \int_incr:N \l__runner_forth_data_top_int
319     }
320     { \msg_error:nn { runner/forth } { empty-stack } }
321 }
322 \cs_new_protected:Npn \__runner_forth_pop_int:NN #1#2
323 {
324   \__runner_forth_pop_int:N #2
325   \__runner_forth_pop_int:N #1
326 }
327 \cs_new_protected:Npn \__runner_forth_pop_int:NNN #1#2#3
328 {
329   \__runner_forth_pop_int:N #3
330   \__runner_forth_pop_int:N #2
331   \__runner_forth_pop_int:N #1
332 }
333 \cs_new_protected:Npn \__runner_forth_pop_int:NNNN #1#2#3#4
334 {
335   \__runner_forth_pop_int:N #4
336   \__runner_forth_pop_int:N #3
337   \__runner_forth_pop_int:N #2
338   \__runner_forth_pop_int:N #1
339 }

```

<pre> __runner_forth_push:n __runner_forth_push:nn __runner_forth_push:nnn __runner_forth_push:nnnn __runner_forth_push:nnnnn __runner_forth_push:nnnnnn </pre>	<p>Push data on the top of the data stack, making sure that we do not step onto the data space.</p> <pre> 340 \cs_new_protected:Npn __runner_forth_push:n #1 341 { 342 \int_compare:nNnTF 343 \l__runner_forth_data_top_int > \l__runner_forth_data_here_int 344 { 345 \int_decr:N \l__runner_forth_data_top_int 346 __runner_array_gset:Nnn \g__runner_forth_data_array 347 \l__runner_forth_data_top_int 348 {#1} 349 } 350 { \msg_error:nn { runner/forth } { out-of-memory } } 351 } 352 \cs_new_protected:Npn __runner_forth_push:nn #1#2 353 { 354 __runner_forth_push:n {#1} 355 __runner_forth_push:n {#2} 356 } 357 \cs_new_protected:Npn __runner_forth_push:nnn #1#2#3 358 { 359 __runner_forth_push:n {#1} 360 __runner_forth_push:n {#2} 361 __runner_forth_push:n {#3} 362 } 363 \cs_new_protected:Npn __runner_forth_push:nnnn #1#2#3#4 364 { 365 __runner_forth_push:n {#1} 366 __runner_forth_push:n {#2} 367 __runner_forth_push:n {#3} 368 __runner_forth_push:n {#4} 369 } 370 \cs_new_protected:Npn __runner_forth_push:nnnnn #1#2#3#4#5 371 { 372 __runner_forth_push:n {#1} 373 __runner_forth_push:n {#2} 374 __runner_forth_push:n {#3} 375 __runner_forth_push:n {#4} 376 __runner_forth_push:n {#5} 377 } 378 \cs_new_protected:Npn __runner_forth_push:nnnnnn #1#2#3#4#5#6 379 { 380 __runner_forth_push:n {#1} 381 __runner_forth_push:n {#2} 382 __runner_forth_push:n {#3} 383 __runner_forth_push:n {#4} 384 __runner_forth_push:n {#5} 385 __runner_forth_push:n {#6} 386 } </pre>
---	---

`\l__runner_forth_base_address_int` The address at which the base of the number system is stored, and a function to retrieve the base.

```

\__runner_forth_base:
387 \int_new:N \l__runner_forth_base_address_int
388 \cs_new:Npn \__runner_forth_base:
389 {
390   \__runner_array_item:NN \g__runner_forth_data_array
391   \l__runner_forth_base_address_int
392 }
```

C.1.2 Other stacks

`\l__runner_forth_flow_seq` The remaining stacks.

```

\l__runner_forth_return_seq
393 \seq_new:N \l__runner_forth_flow_seq
394 \seq_new:N \l__runner_forth_return_seq
```

C.1.3 Dictionary

`\l__runner_forth_words_prop` Keys of the `words` property list are words known to Forth at a given time during the execution of the code. Keys of the `core_words` global property list are words from the core set, with which the `words` property list is initialized.

Values are a digit $d \in \{1, 2, 3\}$ followed by five (decimal) digits forming an execution token `xt`, which lies in the range $[0, 2^{15} - 1]$ for pdfTeX and XeTeX, and in the range $[0, 2^{16} - 1]$ for LuaTeX. The code to be run upon execution is always found in the `\toks` register given by this five-digit integer. The interpretation semantics is identical to the execution semantics unless $d = 3$, in which case the execution token `xt + 1` is used. The compilation semantics is to append `xt` to the current definition if $d = 1$, to perform `xt` if $d = 2$, and to perform `xt + 2` if $d = 3$.

```

395 \prop_new:N \l__runner_forth_words_prop
396 \prop_new:N \g__runner_forth_core_words_prop
```

`\l__runner_forth_toks_int` The next free `\toks` register, for use to store the code corresponding to execution tokens.

```

\g__runner_forth_core_toks_int
397 \int_new:N \l__runner_forth_toks_int
398 \int_new:N \g__runner_forth_core_toks_int
```

`\g__runner_forth_init_tl` Token list containing part of the initialization code, currently many assignments to `\toks` registers.

```

399 \tl_new:N \g__runner_forth_init_tl
```

`__runner_forth_new_core:nn`

```

\__runner_forth_new_immediate_core:nn
\__runner_forth_new_special_core:nnn
\__runner_forth_new_compilation_core:nn
400 \cs_new_protected:Npn \__runner_forth_new_core:nn #1#2
401 {
402   \tl_gput_right:Nx \g__runner_forth_init_tl
403   {
404     \tex_toks:D \int_use:N \g__runner_forth_core_toks_int
405     { \exp_not:n {#2} }
406   }
407   \prop_gput:Nnx \g__runner_forth_core_words_prop {#1}
```

```

408     { \int_eval:n { 100000 + \g__runner_forth_core_toks_int } }
409     \int_gincr:N \g__runner_forth_core_toks_int
410   }
411   \cs_new_protected:Npn \__runner_forth_new_immediate_core:nn #1#2
412   {
413     \tl_gput_right:Nx \g__runner_forth_init_tl
414     {
415       \tex_toks:D \int_use:N \g__runner_forth_core_toks_int
416       { \exp_not:n {#2} }
417     }
418     \prop_gput:Nnx \g__runner_forth_core_words_prop {#1}
419     { \int_eval:n { 200000 + \g__runner_forth_core_toks_int } }
420     \int_gincr:N \g__runner_forth_core_toks_int
421   }
422   \cs_new_protected:Npn \__runner_forth_new_special_core:nnn #1#2#3
423   {
424     \tl_gput_right:Nx \g__runner_forth_init_tl
425     {
426       \tex_toks:D \int_use:N \g__runner_forth_core_toks_int
427       { \exp_not:n {#2} }
428       \tex_toks:D \int_eval:n { \g__runner_forth_core_toks_int + 1 }
429       { \exp_not:n {#3} }
430     }
431     \prop_gput:Nnx \g__runner_forth_core_words_prop {#1}
432     { \int_eval:n { 300000 + \g__runner_forth_core_toks_int } }
433     \int_gadd:Nn \g__runner_forth_core_toks_int \c_two
434   }
435   \cs_new_protected:Npn \__runner_forth_new_compilation_core:nn #1#2
436   {
437     \__runner_forth_new_special_core:nnn {#1}
438     { \msg_error:nnn { runner/forth } { no-interpretation } {#1} }
439     {#2}
440   }

```

_runner_forth_core_alias:nn

```

441   \cs_new_protected:Npn \__runner_forth_core_alias:nn #1#2
442   {
443     \prop_get:NnN \g__runner_forth_core_words_prop {#2} \l__runner_tmpa_tl
444     \prop_gput:NnV \g__runner_forth_core_words_prop {#1} \l__runner_tmpa_tl
445   }

```

_runner_forth_new_word:nn

_runner_forth_new_word:nx

```

446   \cs_new_protected:Npn \__runner_forth_new_word:nn #1#2
447   {
448     \tex_toks:D \l__runner_forth_toks_int {#2}
449     \prop_put:Nnx \l__runner_forth_words_prop {#1}
450     { \int_eval:n { 100000 + \l__runner_forth_toks_int } }
451     \int_incr:N \l__runner_forth_toks_int
452   }
453   \cs_generate_variant:Nn \__runner_forth_new_word:nn { nx }

```

C.2 Helpers

C.2.1 Input

`_runner_forth_input_spaces:` Strip leading spaces from `\l__runner_forth_input_str` using `\use:nn`, then grab everything and store it back into the input string.
`_runner_forth_input_spaces_aux:w`

```
454 \cs_new_protected_nopar:Npn \_runner_forth_input_spaces:
455 {
456   \exp_after:wN \use:nn
457   \exp_after:wN \_runner_forth_input_spaces_aux:w
458   \l__runner_forth_input_str \q_stop
459 }
460 \cs_new_protected_nopar:Npn \_runner_forth_input_spaces_aux:w #1 \q_stop
461 { \tl_set:Nn \l__runner_forth_input_str {#1} }
```

`_runner_forth_input_until:nn` Assuming `\l__runner_forth_input_str` contains the trailing delimiter `#1`, we split it at the first `#1` using an auxiliary function defined on the fly. This auxiliary stores the remainder back as the input string, and runs the second argument of `_runner_forth_input_until:nn`, which gets the text found as `##1`. If the input contained no occurrence of `#1`, we simply need to append it before-hand to get back to the case where the delimiter is present.
`_runner_forth_input_until:nN`

```
462 \cs_new_protected:Npn \_runner_forth_input_until:nn #1#2
463 {
464   \tl_if_in:NnF \l__runner_forth_input_str { #1 }
465   { \tl_put_right:Nn \l__runner_forth_input_str { #1 } }
466   \cs_set:Npn \_runner_tmp:w ##1 #1 ##2 \q_stop
467   {
468     \str_set:Nn \l__runner_forth_input_str {##2}
469     #2
470   }
471   \exp_after:wN \_runner_tmp:w \l__runner_forth_input_str \q_stop
472 }
473 \cs_new_protected:Npn \_runner_forth_input_until:nN #1#2
474 { \_runner_forth_input_until:nn {#1} { #2 {##1} } }
```

`_runner_forth_input_discard:N` This loop removes all occurrences of `#1` from the start of the input string. If the first token in the input string has the same character code as `#1`, discard it and look for more. Otherwise we are done.

```
475 \cs_new_protected:Npn \_runner_forth_input_discard:N #1
476 {
477   \exp_args:No \tl_if_head_eq_charcode:nNT
478   \l__runner_forth_input_str #1
479   {
480     \str_set:Nx \l__runner_forth_input_str
481     { \str_tail:N \l__runner_forth_input_str }
482     \_runner_forth_input_discard:N #1
483   }
484 }
```


C.2.2 Numbers

<code>\c__runner_forth_mod_int</code>	Cells can take up to 2^{24} values. <pre> 485 \int_const:Nn \c__runner_forth_mod_int { 16777216 } </pre>
<code>\l__runner_forth_push_int</code> <code>\l__runner_forth_push_fp</code>	When turning a signed number to an unsigned value for pushing onto the stack, one cannot use <code>\l__runner_tmpa_int</code> , as it may be used by the caller. Similarly for outputting floating points modulo some number. <pre> 486 \int_new:N \l__runner_forth_push_int 487 \fp_new:N \l__runner_forth_push_fp </pre>
<code>__runner_forth_signed:N</code>	Turn the unsigned <code>#1</code> into a signed number, by subtracting 2^{24} if it is not less than 2^{23} . <pre> 488 \cs_new_protected:Npn __runner_forth_signed:N #1 489 { 490 \int_compare:nNf #1 < { \c__runner_forth_mod_int / \c_two } 491 { \int_sub:Nn #1 \c__runner_forth_mod_int } 492 } </pre>
<code>__runner_forth_push_signed:n</code>	Given a signed integer in the range $[-2^{24}, 2^{24} - 1]$ (note the extra large range), push the corresponding unsigned representation onto the stack. This simply requires adding 2^{24} to negative numbers. <pre> 493 \cs_new_protected:Npn __runner_forth_push_signed:n #1 494 { 495 \int_set:Nn \l__runner_forth_push_int {#1} 496 __runner_forth_push:n 497 { 498 \l__runner_forth_push_int 499 \int_compare:nNf \l__runner_forth_push_int > \c_minus_one 500 { + \c__runner_forth_mod_int } 501 } 502 } </pre>
<code>__runner_forth_push_mod:n</code>	Given a non-negative integer, push onto the stack its residue modulo 2^{24} . <pre> 503 \cs_new_protected:Npn __runner_forth_push_mod:n #1 504 { 505 __runner_forth_push:n 506 { \int_mod:nn {#1} { \c__runner_forth_mod_int } } 507 } </pre>
<code>__runner_forth_get_number:nNTF</code>	 <pre> 508 \prg_new_protected_conditional:Npnn __runner_forth_get_number:nN #1#2 { TF } 509 { 510 \int_set:Nn \l__runner_tmpb_int { __runner_forth_base: } 511 \int_set:Nn \l__runner_tmpa_int 512 { \int_min:nn { '9 } { '0 + \l__runner_tmpb_int - 1 } } 513 \int_set:Nn \l__runner_tmpb_int 514 { 'A + \l__runner_tmpb_int - 11 } 515 \bool_set_true:N \l__runner_tmpa_bool 516 \tl_map_inline:nn {#1} </pre>

```

517     {
518         \int_compare:nF { '0 <= '##1 <= \l__runner_tmpa_int }
519         {
520             \int_compare:nF { 'A <= '##1 <= \l__runner_tmpb_int }
521             {
522                 \bool_set_false:N \l__runner_tmpa_bool
523                 \tl_map_break:
524             }
525         }
526     }
527     \bool_if:NTF \l__runner_tmpa_bool
528     {
529         \tl_set:Nx #2 { \int_from_base:nn {#1} { \__runner_forth_base: } }
530         \prg_return_true:
531     }
532     { \prg_return_false: }
533 }

\__runner_forth_push_fp:n
  \__runner_forth_push_fp_mod:n
534 \cs_new_protected:Npn \__runner_forth_push_fp:n #1
535 { \__runner_forth_push:n { \fp_to_int:n {#1} } }
536 \cs_new_protected:Npn \__runner_forth_push_fp_mod:n #1
537 {
538     \fp_set:Nn \l__runner_forth_push_fp { round0(#1) }
539     \__runner_forth_push_fp:n
540     {
541         \l__runner_forth_push_fp - \c__runner_forth_mod_int
542         * round- ( \l__runner_forth_push_fp / \c__runner_forth_mod_int )
543     }
544 }

```

C.2.3 Misc

`\l__runner_forth_input_str` When running Forth, no distinction is made between the program and the input arguments to `\runner`, so those are combined into a single string.

```

545 \str_new:N \l__runner_forth_input_str

```

`\l__runner_forth_stop_bool` This boolean is set to be true by the word `;` which ends definitions: the loop that compiles words into the definition then stops, and we return to interpreting the input. This boolean is also set true by the empty word received by the interpreter when there is no input left. This occurrence marks the end of the program.

```

546 \bool_new:N \l__runner_forth_stop_bool

```

C.3 Core words

C.3.1 Definition words

`\l__runner_forth_def_tl` The definition being built.

```

547 \tl_new:N \l__runner_forth_def_tl

```

<pre> _runner_forth_def_put_right:n _runner_forth_def_put_right:x _runner_forth_def_put_right_x:n </pre>	<p>Helpers to add material to the definition, with various types of expansion.</p> <pre> 548 \cs_new_protected:Npn _runner_forth_def_put_right:n #1 549 { \tl_put_right:Nn \l__runner_forth_def_tl { \exp_not:n {#1} } } 550 \cs_generate_variant:Nn _runner_forth_def_put_right:n { x } 551 \cs_new_protected:Npn _runner_forth_def_put_right_x:n #1 552 { \tl_put_right:Nn \l__runner_forth_def_tl {#1} } </pre>
<pre> \l__runner_forth_def_nesting_int </pre>	<p>Since conditionals add unmatched braces to the definition, we must ensure that at the end of the day there is no extra open or closed brace.</p> <pre> 553 \int_new:N \l__runner_forth_def_nesting_int </pre>
<pre> \l__runner_forth_def_name_str </pre>	<p>The last word that was defined: this is set just after the word's meaning is changed, after the definition is fully read.</p> <pre> 554 \tl_new:N \l__runner_forth_def_name_str </pre>
<p>CONSTANT</p>	
<pre> 555 __runner_forth_new_core:nn { CONSTANT } 556 { 557 __runner_forth_pop_int:N \l__runner_tmpa_int 558 __runner_forth_input_spaces: 559 __runner_forth_input_until:nn { ~ } 560 { 561 __runner_forth_new_word:nx {#1} 562 { __runner_forth_push:n { \int_use:N \l__runner_tmpa_int } } 563 } 564 } </pre>	
<p>VARIABLE</p>	
<pre> 565 __runner_forth_new_core:nn { VARIABLE } 566 { 567 __runner_forth_input_spaces: 568 __runner_forth_input_until:nn { ~ } 569 { 570 __runner_forth_new_word:nx {#1} 571 { 572 __runner_forth_push:n 573 { \int_use:N \l__runner_forth_data_here_int } 574 } 575 \int_incr:N \l__runner_forth_data_here_int 576 } 577 } </pre>	
<pre> : </pre>	<p>Read a word, and feed it to __runner_forth_def:n, responsible for all the work.</p> <pre> 578 __runner_forth_new_core:nn { : } 579 { 580 __runner_forth_input_spaces: 581 __runner_forth_input_until:nN { ~ } __runner_forth_def:n 582 } </pre>

`__runner_forth_def:n` Store in the control-flow stack the definition being built, if any, and empty `\l__runner_forth_def_tl`. Then switch to compiling state, looping until something (typically, the word `;`) makes the “stop” boolean true. We then set the boolean back to be false: otherwise, the interpreter loop would end. A new non-immediate word is then defined with the name `#1` and the definition that was collected. Finally, the name `#1` is store (for IMMEDIATE) and the former value of `\l__runner_forth_def_tl` is restored.

```

583 \cs_new_protected:Npn \__runner_forth_def:n #1
584 {
585   \seq_push:NV \l__runner_forth_flow_seq \l__runner_forth_def_tl
586   \tl_clear:N \l__runner_forth_def_tl
587   \bool_until_do:Nn \l__runner_forth_stop_bool
588   {
589     \__runner_forth_input_spaces:
590     \__runner_forth_input_until:nN { ~ } \__runner_forth_compile:n
591   }
592   \bool_set_false:N \l__runner_forth_stop_bool
593   \int_compare:nNnF \l__runner_forth_def_nesting_int = \c_zero
594   {
595     \msg_error:nn { runner/forth } { too-many-ifs }
596     \prg_replicate:nn \l__runner_forth_def_nesting_int
597     { \__runner_forth_def_put_right_x:n { \if_false: { \fi: } } }
598   }
599   \__runner_forth_new_word:nx {#1} { \l__runner_forth_def_tl }
600   \str_set:Nn \l__runner_forth_def_name_str {#1}
601   \seq_pop:NN \l__runner_forth_flow_seq \l__runner_forth_def_tl
602 }

```

`__runner_forth_compile:n` This is very similar to `__runner_forth_interpret:n`. Try to find the word in the dictionary. If it is present, split its value into 1 + 5 digits: if the first is 1, simply add to the current definition; if the first is 2 perform the execution semantics, and if it is 3, the word is a special word, and we perform the compilation semantics. Otherwise, we try to interpret the word as a number.

```

603 \cs_new_protected:Npn \__runner_forth_compile:n #1
604 {
605   \prop_get:NnNTF \l__runner_forth_words_prop {#1} \l__runner_tmpa_tl
606   { \exp_after:wN \__runner_forth_compile:Nw \l__runner_tmpa_tl \q_stop }
607   {
608     \__runner_forth_get_number:nNTF {#1} \l__runner_tmpa_tl
609     {
610       \__runner_forth_def_put_right:x
611       { \__runner_forth_push:n { \l__runner_tmpa_tl } }
612     }
613     { \msg_error:nnn { runner/forth } { unknown-word } {#1} }
614   }
615 }
616 \cs_new_protected:Npn \__runner_forth_compile:Nw #1#2 \q_stop
617 {
618   \int_case:nnn {#1}
619   {

```

```

620     { 1 }
621     {
622         \__runner_forth_def_put_right:n
623         { \tex_the:D \tex_toks:D #2 \scan_stop: }
624     }
625     { 2 } { \tex_the:D \tex_toks:D #2 \scan_stop: }
626 }
627 { \tex_the:D \tex_toks:D \int_eval:n { #2 + 1 } \scan_stop: }
628 }

```

; All the finishing touches for a definition are done by the implementation of the colon word, and here we only need to stop the compile loop by setting a boolean.

```

629 \__runner_forth_new_compilation_core:nn { ; }
630 { \bool_set_true:N \l__runner_forth_stop_bool }

```

IMMEDIATE

```

631 \__runner_forth_new_core:nn { IMMEDIATE }
632 {
633     \prop_get:NVNTF
634     \l__runner_forth_words_prop
635     \l__runner_forth_def_name_str
636     \l__runner_tmpa_tl
637     {
638         \tl_set:Nx \l__runner_tmpa_tl
639         { \int_eval:n { 100000 + \l__runner_tmpa_tl } }
640         \prop_put:NVV
641         \l__runner_forth_words_prop
642         \l__runner_forth_def_name_str
643         \l__runner_tmpa_tl
644     }
645     {
646         \msg_error:nnx { runner/forth } { no-def-immediate }
647         { \l__runner_forth_def_name_str }
648     }
649 }

```

LITERAL

```

650 \__runner_forth_new_compilation_core:nn { LITERAL }
651 {
652     \__runner_forth_pop_int:N \l__runner_tmpa_int
653     \__runner_forth_def_put_right:x
654     { \__runner_forth_push:n { \int_use:N \l__runner_tmpa_int } }
655 }

```

POSTPONE

```

\__runner_forth_postpone:Nw
656 \__runner_forth_new_compilation_core:nn { POSTPONE }
657 {
658     \__runner_forth_input_spaces:
659     \__runner_forth_input_until:nn { ~ }

```

```

660     {
661       \prop_get:NnN \l__runner_forth_words_prop {#1} \l__runner_tmpa_tl
662       \__runner_forth_def_put_right:x
663       {
664         \exp_after:wN \__runner_forth_postpone:Nw
665         \l__runner_tmpa_tl \q_stop
666       }
667     }
668   }
669   \cs_new:Npn \__runner_forth_postpone:Nw #1#2 \q_stop
670   {
671     \int_case:nnn {#1}
672     {
673       { 1 }
674       {
675         \__runner_forth_def_put_right:n
676         { \exp_not:N \tex_the:D \tex_toks:D #2 \scan_stop: }
677       }
678       { 2 } { \exp_not:N \tex_the:D \tex_toks:D #2 \scan_stop: }
679     }
680     {
681       \exp_not:N \tex_the:D \tex_toks:D
682       \int_eval:n { #2 + 1 } \scan_stop:
683     }
684   }

```

RECURSE When the definition is expanded prior to the assignment, `\l__runner_forth_toks_int` is the value of the `\toks` register which will contain the code. Recursion involves calling this code from within itself.

```

685 \__runner_forth_new_compilation_core:nn { RECURSE }
686 {
687   \__runner_forth_def_put_right_x:n
688   {
689     \exp_not:N \tex_the:D \tex_toks:D
690     \int_use:N \l__runner_forth_toks_int \scan_stop:
691   }
692 }

```

C.3.2 Execution words

- Push onto the data stack the execution token associated to a word (read from the input). The word is read from the input after removing all leading spaces.

```

693 \__runner_forth_new_core:nn { ' }
694 {
695   \__runner_forth_input_spaces:
696   \__runner_forth_input_until:nn { ~ }
697   {
698     \prop_get:NnN \l__runner_forth_words_prop {#1} \l__runner_tmpa_tl
699     \__runner_forth_push:n

```

```

700         { \exp_after:wN \use_none:n \l__runner_tmpa_tl }
701     }
702 }

```

['] Read a word, find it in the dictionary (prop), and find its execution token (the `\use_none:n` construction). Then add to the current definition some code to place this execution token on the stack.

```

703 \__runner_forth_new_compilation_core:nn { ['] }
704 {
705     \__runner_forth_input_spaces:
706     \__runner_forth_input_until:nn { ~ }
707     {
708         \prop_get:NnN \l__runner_forth_words_prop {#1} \l__runner_tmpa_tl
709         \__runner_forth_def_put_right:x
710         {
711             \__runner_forth_push:n
712             { \exp_after:wN \use_none:n \l__runner_tmpa_tl }
713         }
714     }
715 }

```

EXECUTE Pop from the data stack an execution token. Perform it.

```

716 \__runner_forth_new_core:nn { EXECUTE }
717 {
718     \__runner_forth_pop_int:N \l__runner_tmpa_int
719     \tex_the:D \tex_toks:D \l__runner_tmpa_int
720 }

```

C.3.3 Stack words

DEPTH The `data_top` integer is already updated to its new value before the argument of `__runner_forth_push:n` is evaluated, so we need to compensate for that, to give zero for an empty stack.

```

721 \__runner_forth_new_core:nn { DEPTH }
722 {
723     \__runner_forth_push:n
724     {
725         \g__runner_forth_data_size_int
726         - \l__runner_forth_data_top_int - \c_one
727     }
728 }

```

>R Pop from the data stack onto the return stack.

```

729 \__runner_forth_new_compilation_core:nn { >R }
730 {
731     \__runner_forth_pop_int:N \l__runner_tmpa_int
732     \seq_push:NV \l__runner_forth_return_seq \l__runner_tmpa_int
733 }

```

R> Pop from the return stack onto the data stack.

```
734 \__runner_forth_new_core:nn { R> }
735 {
736   \seq_pop:NN \l__runner_forth_return_seq \l__runner_tmpa_t1
737   \__runner_forth_push:n { \l__runner_tmpa_t1 }
738 }
```

R@ Copy from the return stack onto the data stack.

```
739 \__runner_forth_new_core:nn { R@ }
740 {
741   \seq_get:NN \l__runner_forth_return_seq \l__runner_tmpa_t1
742   \__runner_forth_push:n { \l__runner_tmpa_t1 }
743 }
```

?DUP Duplicate top of stack if non-zero.

```
744 \__runner_forth_new_core:nn { ?DUP }
745 {
746   \__runner_forth_pop_int:N \l__runner_tmpa_int
747   \int_compare:nNnTF \l__runner_tmpa_int = \c_zero
748   { \__runner_forth_push:n { \c_zero } }
749   {
750     \__runner_forth_push:nn
751     { \l__runner_tmpa_int } { \l__runner_tmpa_int }
752   }
753 }
```

DROP Pop stack, once or twice.

```
2DROP 754 \__runner_forth_new_core:nn { DROP }
755 { \__runner_forth_pop_int:N \l__runner_tmpa_int }
756 \__runner_forth_new_core:nn { 2DROP }
757 { \__runner_forth_pop_int:NN \l__runner_tmpa_int \l__runner_tmpb_int }
```

DUP Turn x to x, x , or x_1, x_2 to x_1, x_2, x_1, x_2 .

```
2DUP 758 \__runner_forth_new_core:nn { DUP }
759 {
760   \__runner_forth_pop_int:N \l__runner_tmpa_int
761   \__runner_forth_push:nn { \l__runner_tmpa_int } { \l__runner_tmpa_int }
762 }
763 \__runner_forth_new_core:nn { 2DUP }
764 {
765   \__runner_forth_pop_int:NN \l__runner_tmpa_int \l__runner_tmpb_int
766   \__runner_forth_push:nnnn
767   { \l__runner_tmpa_int } { \l__runner_tmpb_int }
768   { \l__runner_tmpa_int } { \l__runner_tmpb_int }
769 }
```


OVER Turn x_1, x_2 to x_1, x_2, x_1 , or x_1, x_2, x_3, x_4 to $x_1, x_2, x_3, x_4, x_1, x_2$.

2OVER

```

770 \__runner_forth_new_core:nn { OVER }
771 {
772   \__runner_forth_pop_int:NN \l__runner_tmpa_int \l__runner_tmpb_int
773   \__runner_forth_push:nnn
774   { \l__runner_tmpa_int } { \l__runner_tmpb_int } { \l__runner_tmpa_int }
775 }
776 \__runner_forth_new_core:nn { 2OVER }
777 {
778   \__runner_forth_pop_int:NNNN
779   \l__runner_tmpa_int \l__runner_tmpb_int
780   \l__runner_tmppc_int \l__runner_tmppd_int
781   \__runner_forth_push:nnnnnn
782   { \l__runner_tmpa_int } { \l__runner_tmpb_int }
783   { \l__runner_tmppc_int } { \l__runner_tmppd_int }
784   { \l__runner_tmpa_int } { \l__runner_tmpb_int }
785 }

```

ROT Turn x_1, x_2, x_3 to x_2, x_3, x_1 .

```

786 \__runner_forth_new_core:nn { ROT }
787 {
788   \__runner_forth_pop_int:NNN
789   \l__runner_tmpa_int \l__runner_tmpb_int \l__runner_tmppc_int
790   \__runner_forth_push:nnn
791   { \l__runner_tmpb_int } { \l__runner_tmppc_int } { \l__runner_tmpa_int }
792 }

```

SWAP Turn x_1, x_2 to x_2, x_1 , or x_1, x_2, x_3, x_4 to x_3, x_4, x_1, x_2 .

2SWAP

```

793 \__runner_forth_new_core:nn { SWAP }
794 {
795   \__runner_forth_pop_int:NN \l__runner_tmpa_int \l__runner_tmpb_int
796   \__runner_forth_push:nn { \l__runner_tmpb_int } { \l__runner_tmpa_int }
797 }
798 \__runner_forth_new_core:nn { 2SWAP }
799 {
800   \__runner_forth_pop_int:NNNN
801   \l__runner_tmpa_int \l__runner_tmpb_int
802   \l__runner_tmppc_int \l__runner_tmppd_int
803   \__runner_forth_push:nn
804   { \l__runner_tmppc_int } { \l__runner_tmppd_int }
805   { \l__runner_tmpa_int } { \l__runner_tmpb_int }
806 }

```

C.3.4 Comparison words

0< If $a < 0$ push -1 (true), otherwise 0 (false).

```

807 \__runner_forth_new_core:nn { 0< }
808 {
809   \__runner_forth_pop_int:N \l__runner_tmpa_int

```

```

810 \int_compare:nNnTF \l__runner_tmpa_int < \c_zero
811 { \__runner_forth_push:n \c_minus_one }
812 { \__runner_forth_push:n \c_zero }
813 }

```

0= If a is 0, push -1 , otherwise 0.

```

814 \__runner_forth_new_core:nn { 0= }
815 {
816   \__runner_forth_pop_int:N \l__runner_tmpa_int
817   \int_compare:nNnTF \l__runner_tmpa_int = \c_zero
818   { \__runner_forth_push:n \c_minus_one }
819   { \__runner_forth_push:n \c_zero }
820 }

```

< Comparisons. Push -1 if true, 0 if false.

```

= 821 \__runner_forth_new_core:nn { < }
> 822 {
823   \__runner_forth_pop_int:NN \l__runner_tmpa_int \l__runner_tmpb_int
824   \int_compare:nNnTF \l__runner_tmpa_int < \l__runner_tmpb_int
825   { \__runner_forth_push:n \c_minus_one }
826   { \__runner_forth_push:n \c_zero }
827 }
828 \__runner_forth_new_core:nn { = }
829 {
830   \__runner_forth_pop_int:NN \l__runner_tmpa_int \l__runner_tmpb_int
831   \int_compare:nNnTF \l__runner_tmpa_int = \l__runner_tmpb_int
832   { \__runner_forth_push:n \c_minus_one }
833   { \__runner_forth_push:n \c_zero }
834 }
835 \__runner_forth_new_core:nn { > }
836 {
837   \__runner_forth_pop_int:NN \l__runner_tmpa_int \l__runner_tmpb_int
838   \int_compare:nNnTF \l__runner_tmpa_int > \l__runner_tmpb_int
839   { \__runner_forth_push:n \c_minus_one }
840   { \__runner_forth_push:n \c_zero }
841 }

```

C.3.5 Arithmetic words

1+ Increment or decrement the top of the stack.

```

1- 842 \__runner_forth_new_core:nn { 1+ }
843 {
844   \__runner_forth_pop_int:N \l__runner_tmpa_int
845   \__runner_forth_push:n { \l__runner_tmpa_int + 1 }
846 }
847 \__runner_forth_new_core:nn { 1- }
848 {
849   \__runner_forth_pop_int:N \l__runner_tmpa_int
850   \__runner_forth_push:n { \l__runner_tmpa_int - 1 }
851 }

```

- 2* Multiply or divide the top of the stack by 2. Neither \TeX 's rounding division nor \LaTeX 's
2/ truncating division do the right thing there, as we want floored division ($(-1)/2 = -1$),
so we distinguish the even and odd cases.

```

852 \__runner_forth_new_core:nn { 2* }
853 {
854   \__runner_forth_pop_int:N \l__runner_tmpa_int
855   \__runner_forth_push:n { \l__runner_tmpa_int * 2 }
856 }
857 \__runner_forth_new_core:nn { 2/ }
858 {
859   \__runner_forth_pop_int:N \l__runner_tmpa_int
860   \int_if_even:nTF { \l__runner_tmpa_int }
861     { \__runner_forth_push:n { \l__runner_tmpa_int / 2 } }
862     { \__runner_forth_push:n { (\l__runner_tmpa_int - 1) / 2 } }
863 }

```

- ABS Get a signed integer from the top of the stack, and change it to its absolute value.

```

864 \__runner_forth_new_core:nn { ABS }
865 {
866   \__runner_forth_pop_int:N \l__runner_tmpa_int
867   \__runner_forth_signed:N \l__runner_tmpa_int
868   \__runner_forth_push:n { \int_abs:n { \l__runner_tmpa_int } }
869 }

```

- MAX Get two signed integers from the stack, and push the biggest/smallest back onto the
MIN stack.

```

870 \__runner_forth_new_core:nn { MAX }
871 {
872   \__runner_forth_pop_int:NN \l__runner_tmpa_int \l__runner_tmpb_int
873   \__runner_forth_signed:N \l__runner_tmpa_int
874   \__runner_forth_signed:N \l__runner_tmpb_int
875   \__runner_forth_push_signed:n
876     { \int_max:nn { \l__runner_tmpa_int } { \l__runner_tmpb_int } }
877 }
878 \__runner_forth_new_core:nn { MIN }
879 {
880   \__runner_forth_pop_int:NN \l__runner_tmpa_int \l__runner_tmpb_int
881   \__runner_forth_signed:N \l__runner_tmpa_int
882   \__runner_forth_signed:N \l__runner_tmpb_int
883   \__runner_forth_push_signed:n
884     { \int_min:nn { \l__runner_tmpa_int } { \l__runner_tmpb_int } }
885 }

```

- NEGATE Get a signed integer from the top of the stack, and change it to its opposite.

```

886 \__runner_forth_new_core:nn { ABS }
887 {
888   \__runner_forth_pop_int:N \l__runner_tmpa_int
889   \__runner_forth_signed:N \l__runner_tmpa_int
890   \__runner_forth_push_signed:n { - \l__runner_tmpa_int }

```

```
891 }
```

- Get two integers from the data stack, sum or take the difference, and push the result
- + onto the stack. The integers can be signed or unsigned, but are given to us as unsigned values in $[0, 2^{24} - 1]$, hence there is an ambiguity by 2^{24} . The difference of two unsigned values lies in $[-2^{24} + 1, 2^{24} - 1]$, and can be brought back to $[0, 2^{24} - 1]$ by adding 2^{24} to negative numbers, as `__runner_forth_push_signed:n` does. The sum, shifted by 2^{24} , lies in $[-2^{24}, 2^{24} - 2]$, and is also appropriate input for the `push_signed` function.

```
892 \__runner_forth_new_core:nn { - }
893 {
894   \__runner_forth_pop_int:NN \l__runner_tmpa_int \l__runner_tmpb_int
895   \__runner_forth_push_signed:n
896   { \l__runner_tmpa_int - \l__runner_tmpb_int }
897 }
898 \__runner_forth_new_core:nn { + }
899 {
900   \__runner_forth_pop_int:NN \l__runner_tmpa_int \l__runner_tmpb_int
901   \__runner_forth_push_signed:n
902   {
903     \l__runner_tmpa_int + \l__runner_tmpb_int
904     - \c__runner_forth_mod_int
905   }
906 }
```

- * Multiply two integers, then push the result (modulo 2^{24}) onto the stack. To avoid TeX overflow, we manipulate the numbers as floating points, computing $a \times b - 2^{24} \times \lfloor a \times b / 2^{24} \rfloor$.

```
907 \__runner_forth_new_core:nn { * }
908 {
909   \__runner_forth_pop_int:NN \l__runner_tmpa_int \l__runner_tmpb_int
910   \__runner_forth_push_fp_mod:n
911   { \l__runner_tmpa_int * \l__runner_tmpb_int }
912 }
```

- / Pop two integers from the stack. The input is assumed signed, so we subtract 2^{24} if the integers are 2^{23} or more. Then perform the division, and push the result (converted back to being unsigned) onto the stack.

```
913 \__runner_forth_new_core:nn { / }
914 {
915   \__runner_forth_pop_int:NN \l__runner_tmpa_int \l__runner_tmpb_int
916   \__runner_forth_signed:N \l__runner_tmpa_int
917   \__runner_forth_signed:N \l__runner_tmpb_int
918   \__runner_forth_push_signed:n
919   {
920     \int_div_truncate:nn
921     { \l__runner_tmpa_int } { \l__runner_tmpb_int }
922   }
923 }
```

MOD Pop two signed integers from the stack. Push the remainder of a divided by b onto the stack.

```

924 \__runner_forth_new_core:nn { MOD }
925 {
926   \__runner_forth_pop_int:NN \l__runner_tmpa_int \l__runner_tmpb_int
927   \__runner_forth_signed:N \l__runner_tmpa_int
928   \__runner_forth_signed:N \l__runner_tmpb_int
929   \__runner_forth_push_signed:n
930   { \int_mod:nn { \l__runner_tmpa_int } { \l__runner_tmpb_int } }
931 }

```

/MOD Get two signed integers from the stack, then perform the division, and put the remainder, then the quotient, on the stack. Both quotient and remainder remain in the range $[-2^{23}, 2^{23}]$ (the upper bound happens when computing $(-2^{23})/(-1)$), and are brought back to an unsigned form before pushing onto the stack.

```

932 \__runner_forth_new_core:nn { /MOD }
933 {
934   \__runner_forth_pop_int:NN \l__runner_tmpa_int \l__runner_tmpb_int
935   \__runner_forth_signed:N \l__runner_tmpa_int
936   \__runner_forth_signed:N \l__runner_tmpb_int
937   \__runner_forth_push_signed:n
938   { \int_mod:nn { \l__runner_tmpa_int } { \l__runner_tmpb_int } }
939   \__runner_forth_push_signed:n
940   {
941     \int_div_truncate:nn
942     { \l__runner_tmpa_int } { \l__runner_tmpb_int }
943   }
944 }

```

*/ Those two words have a lot of code in common, and start by popping three signed integers from the stack. Then compute $a \times b/c$ using floating points, rounding it towards zero. For */ this is the end, and the result is output (after suitably reducing its range). For */MOD, use that quotient to compute the remainder.

```

945 \__runner_forth_new_core:nn { */ }
946 {
947   \__runner_forth_pop_int:NNN
948   \l__runner_tmpa_int \l__runner_tmpb_int \l__runner_tmpe_int
949   \__runner_forth_signed:N \l__runner_tmpa_int
950   \__runner_forth_signed:N \l__runner_tmpb_int
951   \__runner_forth_signed:N \l__runner_tmpe_int
952   \__runner_forth_push_fp_mod:n
953   { \l__runner_tmpa_int * \l__runner_tmpb_int / \l__runner_tmpe_int }
954 }
955 \__runner_forth_new_core:nn { */MOD }
956 {
957   \__runner_forth_pop_int:NNN
958   \l__runner_tmpa_int \l__runner_tmpb_int \l__runner_tmpe_int
959   \__runner_forth_signed:N \l__runner_tmpa_int
960   \__runner_forth_signed:N \l__runner_tmpb_int

```

```

961  \__runner_forth_signed:N \l__runner_tmpc_int
962  \fp_set:Nn \l__runner_tmpa_fp
963  {
964      round0 ( \l__runner_tmpa_int * \l__runner_tmpb_int
965              / \l__runner_tmpc_int )
966  }
967  \__runner_forth_push_fp_mod:n
968  {
969      \l__runner_tmpa_int * \l__runner_tmpb_int
970      - \l__runner_tmpc_int * \l__runner_tmpa_fp
971  }
972  \__runner_forth_push_fp_mod:n { \l__runner_tmpa_fp }
973  }

```

C.3.6 Storage words

ALIGN Since all addresses are aligned, **ALIGN** does nothing, **ALIGNED** simply pops and pushes back the same value.

```

974  \__runner_forth_new_core:nn { ALIGN } { }
975  \__runner_forth_new_core:nn { ALIGNED }
976  {
977      \__runner_forth_pop_int:N \l__runner_tmpa_int
978      \__runner_forth_push:n \l__runner_tmpa_int
979  }

```

ALLOT Shift the data pointer by the value at the top of the stack

```

980  \__runner_forth_new_core:nn { ALLOT }
981  {
982      \__runner_forth_pop_int:N \l__runner_tmpa_int
983      \int_add:Nn \l__runner_forth_data_here_int \l__runner_tmpa_int
984      \int_compare:nNnTF \l__runner_forth_data_here_int < \c_zero
985      {
986          \msg_error:nnx { runner/forth } { out-of-bounds }
987          { \int_use:N \l__runner_forth_data_here_int }
988          \int_zero:N \l__runner_forth_data_here_int
989      }
990      {
991          \int_compare:nNnTF
992          \l__runner_forth_data_here_int > \l__runner_forth_data_top_int
993          {
994              \msg_error:nn { runner/forth } { out-of-memory }
995              \int_set_eq:NN \l__runner_forth_data_here_int
996              \l__runner_forth_data_top_int
997          }
998      }
999  }

```

FILL Fill the addresses from a to $a + b - 1$ with the value c .

```

1000 \__runner_forth_new_core:nn { FILL }

```

```

1001 {
1002   \__runner_forth_pop_int:NNN
1003   \l__runner_tmpa_int \l__runner_tmpb_int \l__runner_tmpc_int
1004   \int_step_inline:nnnn
1005   { \l__runner_tmpa_int }
1006   { 1 }
1007   { \l__runner_tmpa_int + \l__runner_tmpb_int - 1 }
1008   {
1009     \__runner_array_gset:Nnn \g__runner_forth_data_array {#1}
1010     { \l__runner_tmpc_int }
1011   }
1012 }

```

HERE

```

1013 \__runner_forth_new_core:nn { HERE }
1014 { \__runner_forth_push:n { \l__runner_forth_data_here_int } }

```

! Store value at a given address.

```

1015 \__runner_forth_new_core:nn { ! }
1016 {
1017   \__runner_forth_pop_int:NN \l__runner_tmpa_int \l__runner_tmpb_int
1018   \__runner_array_gset:Nnn \g__runner_forth_data_array
1019   { \l__runner_tmpb_int } { \l__runner_tmpa_int }
1020 }

```

2! Store values at a given address and at the next cell.

```

1021 \__runner_forth_new_core:nn { 2! }
1022 {
1023   \__runner_forth_pop_int:NNN
1024   \l__runner_tmpa_int \l__runner_tmpb_int \l__runner_tmpc_int
1025   \__runner_array_gset:Nnn \g__runner_forth_data_array
1026   { \l__runner_tmpc_int } { \l__runner_tmpb_int }
1027   \__runner_array_gset:Nnn \g__runner_forth_data_array
1028   { \l__runner_tmpc_int + 1 } { \l__runner_tmpa_int }
1029 }

```

+! Sum a and the value at b into the integer $\backslash\text{l_runner_tmpa_int}$, then store that into the address at b .

```

1030 \__runner_forth_new_core:nn { +! }
1031 {
1032   \__runner_forth_pop_int:NN \l__runner_tmpa_int \l__runner_tmpb_int
1033   \int_add:Nn \l__runner_tmpa_int
1034   {
1035     \__runner_array_item:Nn \g__runner_forth_data_array
1036     \l__runner_tmpb_int
1037   }
1038   \int_compare:nNf \l__runner_tmpa_int < \c__runner_forth_mod_int
1039   { \int_sub:Nn \l__runner_tmpa_int \c__runner_forth_mod_int }
1040   \__runner_array_gset:Nnn \g__runner_forth_data_array

```

```

1041     { \l__runner_tmpb_int } { \l__runner_tmpa_int }
1042 }
    Store value in a newly allocated cell, given by the data-space pointer “here”.
1043 \__runner_forth_new_core:nn { , }
1044 {
1045     \__runner_forth_pop_int:N \l__runner_tmpa_int
1046     \__runner_forth_put_here:n { \l__runner_tmpa_int }
1047 }

```

- Ⓒ Pop an address from the stack, and push the value x of the corresponding memory cell.

```

1048 \__runner_forth_new_core:nn { Ⓒ }
1049 {
1050     \__runner_forth_pop_int:N \l__runner_tmpa_int
1051     \__runner_forth_push:n
1052     {
1053         \__runner_array_item:Nn \g__runner_forth_data_array
1054         { \l__runner_tmpa_int }
1055     }
1056 }

```

- 2Ⓒ Find an address on the stack, and fetch the value x_2 of the corresponding memory cell, and the next, x_1 . The value x_2 (at the given address) ends up at the top of the stack.

```

1057 \__runner_forth_new_core:nn { 2Ⓒ }
1058 {
1059     \__runner_forth_pop_int:N \l__runner_tmpa_int
1060     \__runner_forth_push:nn
1061     {
1062         \__runner_array_item:Nn \g__runner_forth_data_array
1063         { \l__runner_tmpa_int + 1 }
1064     }
1065     {
1066         \__runner_array_item:Nn \g__runner_forth_data_array
1067         { \l__runner_tmpa_int }
1068     }
1069 }

```

- C! Since cells are one character wide, storing at a character-aligned address, or at an aligned C@ address, is the same.

```

C 1070 \__runner_forth_core_alias:nn { C! } { ! }
    1071 \__runner_forth_core_alias:nn { C, } { , }
    1072 \__runner_forth_core_alias:nn { C@ } { @ }

```

CELL+ Since the cell size, the character size and the address sizes are identical, CELL+ and CHAR+ simply add 1, like 1+, and CELLS and CHARS do nothing to the top of the stack (which must be present), just like ALIGNED.

```

CHARS 1073 \__runner_forth_core_alias:nn { CELL+ } { 1+ }
    1074 \__runner_forth_core_alias:nn { CHAR+ } { 1+ }
    1075 \__runner_forth_core_alias:nn { CELLS } { ALIGNED }
    1076 \__runner_forth_core_alias:nn { CHARS } { ALIGNED }

```


C.3.7 Words for display, input, and strings

- . Get a value from the data stack, convert it to the base, which can be altered through the Forth keyword BASE. Trailing space.

```
1077 \__runner_forth_new_core:nn { . }
1078 {
1079   \__runner_forth_pop_int:N \l__runner_tmpa_int
1080   \__runner_output:x
1081   { \int_to_base:nn { \l__runner_tmpa_int } { \__runner_forth_base: } ~ }
1082 }
```

- ." The dot-quote word reads the input until a double quote. It has no interpretation semantics. Its compilation semantics is to add code to the current definition that display what it read.

```
1083 \__runner_forth_new_compilation_core:nn { ." }
1084 {
1085   \__runner_forth_input_until:nn "
1086   { \__runner_forth_def_put_right:n { \__runner_output:n {#1} } }
1087 }
```

BASE

```
1088 \__runner_forth_new_core:nn { BASE }
1089 { \__runner_forth_push:n \l__runner_forth_base_address_int }
```

BL

```
1090 \__runner_forth_new_core:nn { BL }
1091 { \__runner_forth_push:n { '\ } }
```

CHAR Read a word, leave its first character on the stack.

```
1092 \__runner_forth_new_core:nn { CHAR }
1093 {
1094   \__runner_forth_input_spaces:
1095   \__runner_forth_input_until:nn { ~ }
1096   {
1097     \tl_set:Nx \l__runner_tmpa_tl { \str_head:n {#1} }
1098     \__runner_forth_push:n { \exp_after:wN '\l__runner_tmpa_tl }
1099   }
1100 }
```

[CHAR] Read a word, add code to the current definition that leaves its first character on the stack.

```
1101 \__runner_forth_new_compilation_core:nn { [CHAR] }
1102 {
1103   \__runner_forth_input_spaces:
1104   \__runner_forth_input_until:nn { ~ }
1105   {
1106     \tl_set:Nx \l__runner_tmpa_tl { \str_head:n {#1} }
1107     \__runner_forth_def_put_right:x
1108     { \__runner_forth_push:n { \exp_after:wN '\l__runner_tmpa_tl } }
```

```

1109     }
1110 }

```

COUNT Convert the address of a counted string to its length (on top of the stack), and the address of the first character. This could be defined with : COUNT DUP 1+ SWAP @ ; but this is faster.

```

1111 \__runner_forth_new_core:nn { COUNT }
1112 {
1113   \__runner_forth_pop_int:N \l__runner_tmpa_int
1114   \__runner_forth_push:nn
1115   { \l__runner_tmpa_int + 1 }
1116   {
1117     \__runner_array_item:Nn \g__runner_forth_data_array
1118     { \l__runner_tmpa_int }
1119   }
1120 }

```

CR

```

1121 \__runner_forth_new_core:nn { CR }
1122 { \__runner_output:x { \iow_newline: } }

```

DECIMAL

```

1123 \__runner_forth_new_core:nn { DECIMAL }
1124 {
1125   \__runner_array_gset:Nnn \g__runner_forth_data_array
1126   \l__runner_forth_base_address_int \c_ten
1127 }

```

EMIT Output a character with the character code found by popping the data stack.

```

1128 \__runner_forth_new_core:nn { EMIT }
1129 {
1130   \__runner_forth_pop_int:N \l__runner_tmpa_int
1131   \__runner_output_char:n { \l__runner_tmpa_int }
1132 }

```

S" Read a double-quoted string, then go through its characters one by one, storing them into the data array (starting at the data-space pointer). Then append to the current definition some code which pushes the address and length of the string to the stack.

```

1133 \__runner_forth_new_compilation_core:nn { S" }
1134 {
1135   \__runner_forth_input_until:nn "
1136   { \tl_set:Nn \l__runner_tmpa_tl {#1} }
1137   \int_set_eq:NN \l__runner_tmpa_int \l__runner_forth_data_here_int
1138   \tl_replace_all:Nnn \l__runner_tmpa_tl { ~ } { { ~ } }
1139   \tl_map_inline:Nn \l__runner_tmpa_tl
1140   { \__runner_forth_put_here:n { '#1' } }
1141   \__runner_forth_def_put_right:x
1142   {
1143     \__runner_forth_push:n { \int_use:N \l__runner_tmpa_int }

```

```

1144     \__runner_forth_push:n
1145     {
1146         \int_eval:n
1147         { \l__runner_forth_data_here_int - \l__runner_tmpa_int }
1148     }
1149 }
1150 }

```

SPACE

```

1151 \__runner_forth_new_core:nn { SPACE }
1152 { \__runner_output:n { ~ } }

```

SPACES

```

1153 \__runner_forth_new_core:nn { SPACES }
1154 {
1155     \__runner_forth_pop_int:N \l__runner_tmpa_int
1156     \__runner_output:x { \prg_replicate:nn { \l__runner_tmpa_int } { ~ } }
1157 }

```

TYPE

```

1158 \__runner_forth_new_core:nn { TYPE }
1159 {
1160     \__runner_forth_pop_int:NN \l__runner_tmpa_int \l__runner_tmpb_int
1161     \int_step_inline:nnnn
1162     { \l__runner_tmpa_int }
1163     { 1 }
1164     { \l__runner_tmpa_int + \l__runner_tmpb_int - 1 }
1165     {
1166         \__runner_output_char:n
1167         { \__runner_array_item:Nn \g__runner_forth_data_array {#1} }
1168     }
1169 }

```

WORD

```

1170 \group_begin:
1171 \char_set_catcode_other:n { 0 }
1172 \__runner_forth_new_core:nn { WORD }
1173 {
1174     \__runner_forth_pop_int:N \l__runner_tmpa_int
1175     \__runner_forth_push:n { \l__runner_forth_data_here_int }
1176     \group_begin:
1177     \char_set_lccode:nn { 0 } { \l__runner_tmpa_int }
1178     \tex_lowercase:D
1179     {
1180         \group_end:
1181         \tl_set:Nn \l__runner_tmpa_tl { ^^@ }
1182     }
1183     \exp_args:No \__runner_forth_input_discard:N \l__runner_tmpa_tl
1184     \exp_args:No \__runner_forth_input_until:nn \l__runner_tmpa_tl

```

```

1185         { \tl_set:Nn \l__runner_tmpa_tl {#1} }
1186         \__runner_forth_put_here:n { \str_count:N \l__runner_tmpa_tl }
1187         \tl_replace_all:Nnn \l__runner_tmpa_tl { ~ } { { ~ } }
1188         \tl_map_inline:Nn \l__runner_tmpa_tl
1189         { \__runner_forth_put_here:n { '#1' } }
1190     }
1191 \group_end:

```

C.3.8 Conditional words

IF

```

1192 \__runner_forth_new_compilation_core:nn { IF }
1193 {
1194     \__runner_forth_def_put_right_x:n
1195     { \__runner_forth_compiled_if:nw { \if_false: } \fi: }
1196     \int_incr:N \l__runner_forth_def_nesting_int
1197 }

```

ELSE

```

1198 \__runner_forth_new_compilation_core:nn { ELSE }
1199 {
1200     \int_compare:nNnTF \l__runner_forth_def_nesting_int > \c_zero
1201     {
1202         \__runner_forth_def_put_right_x:n
1203         { \if_false: { \fi: } { \if_false: } \fi: }
1204     }
1205     { \msg_error:nnn { runner/forth } { misplaced } { ELSE } }
1206 }

```

THEN

```

1207 \__runner_forth_new_compilation_core:nn { THEN }
1208 {
1209     \int_compare:nNnTF \l__runner_forth_def_nesting_int > \c_zero
1210     {
1211         \__runner_forth_def_put_right_x:n
1212         { \if_false: { \fi: } \__runner_forth_compiled_then: }
1213     }
1214     { \msg_error:nnn { runner/forth } { misplaced } { THEN } }
1215     \int_decr:N \l__runner_forth_def_nesting_int
1216 }

```

_runner_forth_compiled_if:nw

_runner_forth_compiled_then:

```

1217 \cs_new_protected:Npn \__runner_forth_compiled_if:nw
1218     #1#2 \__runner_forth_compiled_then:
1219     {
1220         \__runner_forth_pop_int:N \l__runner_tmpa_int
1221         \int_compare:nNnTF \l__runner_tmpa_int = \c_zero {#2} {#1}
1222     }
1223 \cs_new_protected_nopar:Npn \__runner_forth_compiled_then:
1224     { \msg_error:nn { runner/forth } { internal } }

```

C.3.9 Looping words

DO

```
1225 \__runner_forth_new_compilation_core:nn { DO }
1226 {
1227   \__runner_forth_def_put_right_x:n
1228   { \__runner_forth_compiled_do:n { \if_false: } \fi: }
1229   \int_incr:N \l__runner_forth_def_nesting_int
1230 }
```

LOOP

```
1231 \__runner_forth_new_compilation_core:nn { LOOP }
1232 {
1233   \int_compare:nNnTF \l__runner_forth_def_nesting_int > \c_zero
1234   { \__runner_forth_def_put_right_x:n { \if_false: { \fi: } } }
1235   { \msg_error:nnn { runner/forth } { misplaced } { LOOP } }
1236   \int_decr:N \l__runner_forth_def_nesting_int
1237 }
```

__runner_forth_compiled_do:n

```
1238 \cs_new_protected:Npn \__runner_forth_compiled_do:n #1
1239 {
1240   \__runner_forth_pop_int:NN \l__runner_tmpa_int \l__runner_tmpb_int
1241   \seq_push:Nx \l__runner_forth_return_seq
1242   { \int_use:N \l__runner_tmpa_int } % ^^A limit
1243   \seq_push:Nx \l__runner_forth_return_seq
1244   { \int_use:N \l__runner_tmpb_int } % ^^A index
1245   \int_do_until:nNnn \l__runner_tmpb_int = \l__runner_tmpa_int
1246   {
1247     #1
1248     \seq_pop:NN \l__runner_forth_return_seq \l__runner_tmpb_tl
1249     \int_set:Nn \l__runner_tmpb_int { \l__runner_tmpb_tl + 1 }
1250     \seq_get:NN \l__runner_forth_return_seq \l__runner_tmpa_tl
1251     \int_set:Nn \l__runner_tmpa_int { \l__runner_tmpa_tl }
1252     \seq_push:Nx \l__runner_forth_return_seq
1253     { \int_use:N \l__runner_tmpb_int }
1254   }
1255   \__runner_break_point:n
1256   {
1257     \seq_pop:NN \l__runner_forth_return_seq \l__runner_tmpb_tl
1258     \seq_pop:NN \l__runner_forth_return_seq \l__runner_tmpa_tl
1259   }
1260 }
```

I

J

```
1261 \__runner_forth_new_compilation_core:nn { I }
1262 {
1263   \__runner_forth_def_put_right:n
1264   {
```

```

1265     \__runner_forth_push:n
1266     { \seq_item:Nn \l__runner_forth_return_seq { 1 } }
1267   }
1268 }
1269 \__runner_forth_new_compilation_core:nn { J }
1270 {
1271   \__runner_forth_def_put_right_x:n
1272   {
1273     \__runner_forth_push:n
1274     { \seq_item:Nn \l__runner_forth_return_seq { 3 } }
1275   }
1276 }

```

LEAVE Break the current loop, which takes care of removing the loop control parameters from the return stack.

```

1277 \__runner_forth_new_compilation_core:nn { LEAVE }
1278 { \__runner_forth_def_put_right:n { \__runner_break:n { } } }

```

UNLOOP Drop the loop control parameters from the return stack.

```

1279 \__runner_forth_new_compilation_core:nn { UNLOOP }
1280 {
1281   \seq_pop:NN \l__runner_forth_return_seq \l__runner_tmpa_tl
1282   \seq_pop:NN \l__runner_forth_return_seq \l__runner_tmpb_tl
1283 }

```

C.3.10 Ending words

When there is no more input, the interpreter sees an empty word. Its execution semantics is to stop the interpreter by setting a boolean. This also works to stop when in compilation mode.

```

1284 \__runner_forth_new_immediate_core:nn { }
1285 { \bool_set_true:N \l__runner_forth_stop_bool }

```

C.3.11 Environmental queries

ENVIRONMENT? Pop an address a and a length b from the stack, and extract a comma-list of the values from a to $a + b - 1$ inclusive (the query string). If that string matches any known one, return the result and a true flag, otherwise return a false flag.

```

1286 \__runner_forth_new_core:nn { ENVIRONMENT? }
1287 {
1288   \__runner_forth_pop_int:NN \l__runner_tmpa_int \l__runner_tmpb_int
1289   \prop_get:NxNTF \g__runner_forth_environment_prop
1290   {
1291     \int_step_function:nnnN
1292     { \l__runner_tmpa_int }
1293     { 1 }
1294     { \l__runner_tmpa_int + \l__runner_tmpb_int - 1 }
1295     \__runner_forth_environment_aux:n

```

```

1296     }
1297     \l__runner_tmpa_t1
1298     {
1299         \l__runner_tmpa_t1
1300         \__runner_forth_push:n { -1 }
1301     }
1302     {
1303         \__runner_forth_push:n { 0 }
1304     }
1305 }
1306 \cs_new:Npn \__runner_forth_environment_aux:n #1
1307 { \__runner_array_item:Nn \g__runner_forth_data_array {#1} , }

```

\g__runner_forth_environment_prop Keys are known string, in the form of comma lists of their character codes. Currently,
 __runner_forth_environment_def:nn the following exist:

- ADDRESS-UNIT-BITS: 24
- CORE: false
- CORE-EXT: false
- FLOORED: false
- MAX-D: 8388607 and 16777215
- MAX-N: 8388607
- MAX-U: 16777215
- MAX-UD: 16777215 and 16777215.

Values are the code that pushes the appropriate values to the stack.

```

1308 \prop_new:N \g__runner_forth_environment_prop
1309 \group_begin:
1310 \cs_set_protected:Npn \__runner_forth_environment_def:nn #1#2
1311 {
1312     \prop_gput:Nxn \g__runner_forth_environment_prop
1313     { \tl_map_function:nN {#1} \__runner_tmp:w }
1314     {#2}
1315 }
1316 \cs_set:Npn \__runner_tmp:w #1 { \int_eval:n { '#1 } , }
1317 \__runner_forth_environment_def:nn { ADDRESS-UNIT-BITS }
1318 { \__runner_forth_push:n { 24 } }
1319 \__runner_forth_environment_def:nn { CORE }
1320 { \__runner_forth_push:n { 0 } }
1321 \__runner_forth_environment_def:nn { CORE-EXT }
1322 { \__runner_forth_push:n { 0 } }
1323 \__runner_forth_environment_def:nn { FLOORED }
1324 { \__runner_forth_push:n { 0 } }
1325 \__runner_forth_environment_def:nn { MAX-D }

```

```

1326 { % ^^A todo: check endianness
1327   \__runner_forth_push:n { 8388607 }
1328   \__runner_forth_push:n { 16777215 }
1329 }
1330 \__runner_forth_environment_def:nn { MAX-N }
1331 { \__runner_forth_push:n { 8388607 } }
1332 \__runner_forth_environment_def:nn { MAX-U }
1333 { \__runner_forth_push:n { 16777215 } }
1334 \__runner_forth_environment_def:nn { MAX-UD }
1335 {
1336   \__runner_forth_push:n { 16777215 }
1337   \__runner_forth_push:n { 16777215 }
1338 }
1339 \group_end:

```

C.3.12 Misc words

- (Comments are implemented by grabbing a piece of input delimited by a right parenthesis, and discarding it with `\use_none:n`. This is an immediate word.

```

1340 \__runner_forth_new_immediate_core:nn (
1341   { \__runner_forth_input_until:nN ) \use_none:n }

```

C.4 Running the interpreter

`__runner_forth_run:` The interpreter is an infinite loop retrieving a word at each iteration. When there is nothing left, the word fed to `__runner_forth_interpret:n` is empty, and this word is defined to set the boolean `\l__runner_forth_stop_bool`, so that the loop stops.

```

1342 \cs_new_protected_nopar:Npn \__runner_forth_run:
1343 {
1344   \__runner_forth_init:
1345   \bool_set_false:N \l__runner_forth_stop_bool
1346   \bool_until_do:Nn \l__runner_forth_stop_bool
1347   {
1348     \__runner_forth_input_spaces:
1349     \__runner_forth_input_until:nN { ~ } \__runner_forth_interpret:n
1350   }
1351 }

```

`__runner_forth_init:` Before starting the interpreter loop, we concatenate the program string and the input string, with a space in between, then replace all non-graphical characters by spaces (some care is needed to avoid losing spaces), then set some of the initial values.

```

1352 \cs_new_protected_nopar:Npn \__runner_forth_init:
1353 {
1354   \tl_use:N \g__runner_forth_init_tl
1355   \int_set_eq:NN
1356     \l__runner_forth_toks_int
1357     \g__runner_forth_core_toks_int
1358   \int_set_eq:NN

```



```

1359     \l__runner_forth_data_top_int
1360     \g__runner_forth_data_size_int
1361     \prop_set_eq:NN
1362     \l__runner_forth_words_prop
1363     \g__runner_forth_core_words_prop
1364     \int_set:Nn \l__runner_forth_base_address_int { 1 }
1365     \__runner_forth_interpret:n { DECIMAL }
1366     \int_set:Nn \l__runner_forth_data_here_int { 2 }
1367     \tl_set:Nx \l__runner_tmpa_tl
1368     { \l__runner_program_str \c_space_tl \l__runner_input_str }
1369     \tl_replace_all:Nnn \l__runner_tmpa_tl { ~ } { { ~ } }
1370     \str_set:Nx \l__runner_forth_input_str
1371     {
1372         \tl_map_function:NN \l__runner_tmpa_tl
1373         \__runner_forth_cleanup:n
1374     }
1375 }
1376 \cs_new:Npn \__runner_forth_cleanup:n #1
1377 { \int_compare:nTF { 31 < '#1 < 128 } {#1} { ~ } }

```

`__runner_forth_interpret:n` Use the interpretation semantics of #1. If this fails, we hope to have a number, which should be parsed as such.

```

1378 \cs_new_protected:Npn \__runner_forth_interpret:n #1
1379 {
1380     \prop_get:NnNTF \l__runner_forth_words_prop {#1} \l__runner_tmpa_tl
1381     {
1382         \tex_the:D \tex_toks:D
1383         \exp_after:wN \use_none:n \l__runner_tmpa_tl \scan_stop:
1384     }
1385     {
1386         \__runner_forth_get_number:nNTF {#1} \l__runner_tmpa_tl
1387         { \__runner_forth_push:n { \l__runner_tmpa_tl } }
1388         { \msg_error:nnn { runner/forth } { unknown-word } {#1} }
1389     }
1390 }

```

C.5 Messages

```

1391 \msg_new:nnn { runner/forth } { unknown-word }
1392 { The~word~'#1'~is~not~defined. }
1393 \msg_new:nnn { runner/forth } { empty-stack }
1394 { The~data~stack~is~empty,~and~there~is~nothing~to~retrieve~there. }
1395 \msg_new:nnnn { runner/forth } { out-of-memory }
1396 { The~Forth~interpreter~ran~out~of~memory. }
1397 {
1398     Summary~of~memory~use:\\
1399     \iow_indent:n
1400     {
1401         Data~usage:~

```

```

1402         \int_use:N \l__runner_forth_data_here_int
1403         \ cells\\
1404     Stack~size:~
1405         \int_eval:n
1406         {
1407             \g__runner_forth_data_size_int
1408             - \l__runner_forth_data_top_int
1409         }
1410         \ cells\\
1411     Total:~
1412         \int_use:N \g__runner_forth_data_size_int
1413         \ cells.
1414     }
1415 }
1416 \msg_new:nnn { runner/forth } { out-of-bounds }
1417 { ALLOT~was~called~with~a~negative~argument~that~made~HERE~=#1. }
1418 \msg_new:nnn { runner/forth } { no-interpretation }
1419 { The~word~#1~can~only~be~used~in~definitions. }
1420 \msg_new:nnn { runner/forth } { no-def-immediate }
1421 { Somehow~the~word~#1~cannot~be~found~in~the~dictionary }
1422 \msg_new:nnn { runner/forth } { too-many-ifs }
1423 { More~IFs~than~THENs~in~this~definition! }
1424 \msg_new:nnn { runner/forth } { misplaced }
1425 { Misplaced~#1. }
1426 \msg_new:nnn { runner/forth } { internal }
1427 { Internal~error.~Please~report. }

</package>

```