# The runner package: interpreters for various languages*

Bruno Le Floch

2013/04/20

## Contents



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

# 1 runner documentation

This package aims to provide interpreters for various languages, with code written for all commonly used modern TEX engines (pdfTEX, XETEX, and LuaTEX). 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 $2_\varepsilon$ 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 TEX, 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
  +++++ +++++
    [ >+++++++ >++++++++++ >+++ >+ < < < < -]
  >++.
  >+.
  +++++++.
  .
  +++.
  >++.
```

[1] More precisely, the size of the array is limited to the range $[-2^{31}+1, 2^{31}-1]$, like other TEX 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 | $\geq 0$ |

### 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>

```
\RequirePackage{expl3}[2012/12/07]
\RequirePackage{l3str}[2012/12/07]
\RequirePackage{xparse}[2012/12/07]
\ProvidesExplPackage
  {runner} {2013/04/20} {0.0a} {Interpreters for various languages}
```

⟨@@=runner⟩

## A.1 Variables and constants

\g_runner_languages_clist

The list of known languages.

```
\clist_new:N \g_runner_languages_clist
\clist_gset:Nn \g_runner_languages_clist
  {
    brainfuck,
    forth,
  }
```

(*End definition for* `\g_runner_languages_clist`. *This variable is documented on page* 8.)

\l__runner_input_str
\l__runner_output_str
\l__runner_program_str

Input and output common to all the programming languages. This assumes that there is no need to allow cross-language mixing.

```
\tl_new:N \l__runner_input_str
\tl_new:N \l__runner_output_str
\tl_new:N \l__runner_program_str
```

(*End definition for* `\l__runner_input_str`, `\l__runner_output_str`, *and* `\l__runner_program_str`.)

\l__runner_tmpa_tl
\l__runner_tmpb_tl
\l__runner_tmpc_tl
\l__runner_tmpd_tl
\l__runner_tmpa_fp
\l__runner_tmpa_int
\l__runner_tmpb_int
\l__runner_tmpc_int
\l__runner_tmpd_int
\l__runner_tmpa_bool

Scratch variables used by various languages.

```
\tl_new:N \l__runner_tmpa_tl
\tl_new:N \l__runner_tmpb_tl
\tl_new:N \l__runner_tmpc_tl
\tl_new:N \l__runner_tmpd_tl
\fp_new:N \l__runner_tmpa_fp
\int_new:N \l__runner_tmpa_int
\int_new:N \l__runner_tmpb_int
\int_new:N \l__runner_tmpc_int
\int_new:N \l__runner_tmpd_int
\bool_new:N \l__runner_tmpa_bool
```

(*End definition for* `\l__runner_tmpa_tl` *and others.*)

__runner_tmp:w

Short-term auxiliary, defined on the fly.

```
\cs_new_eq:NN \__runner_tmp:w ?
```

(*End definition for* `\__runner_tmp:w`.)

\l__runner_output_key_tl

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.

```
\tl_new:N \l__runner_output_key_tl
```

(*End definition for* `\l__runner_output_key_tl`.)

\g__runner_array_font_size_int

The size of the last font that was used as an array.

```
\int_new:N \g__runner_array_font_size_int
```

(*End definition for* `\g__runner_array_font_size_int`.)

## 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 TEX's `\tenrm` at a tiny size which no one would use, and zero out the 7 first parameters, to which TEX 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 TEX errors.

```
\cs_new_protected:Npn \__runner_array_new:Nn #1#2
  {
    \cs_new_eq:NN #1 \tex_nullfont:D
    \int_compare:nNnTF {#2} > \c_zero
      {
        \int_gincr:N \g__runner_array_font_size_int
        \tex_global:D \tex_font:D #1 = cmr10~at~
          \g__runner_array_font_size_int sp \scan_stop:
        \int_step_inline:nnnn { 1 } { 1 } { 7 }
          { \tex_fontdimen:D ##1 #1 = \c_zero_dim }
        \tex_fontdimen:D \int_eval:n {#2} #1 = \c_zero_dim
      }
      {
        \msg_error:nnxx { runner } { negative-array-size }
          { \token_to_str:N #1 } { \int_eval:n {#2} }
      }
  }
```

(*End definition for* `\__runner_array_new:Nn`.)

`\__runner_array_item: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.

```
\cs_new:Npn \__runner_array_item:Nn #1#2
  { \tex_number:D \tex_fontdimen:D \int_eval:n {#2} #1 }
\cs_new:Npn \__runner_array_item:NN #1#2
  { \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.

```
\cs_new_protected:Npn \__runner_array_gset:Nnn #1#2#3
  {
    \tex_fontdimen:D \int_eval:n {#2} #1
      = \int_eval:n {#3} sp \scan_stop:
  }
```

(*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`

```
\cs_new_protected:Npn \__runner_output:n #1
  { \str_put_right:Nn \l__runner_output_str {#1} }
\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.

```
\group_begin:
  \char_set_catcode_other:N \^^@
  \cs_new_protected:Npn \__runner_output_char:n #1
    {
      \group_begin:
      \char_set_lccode:nn {0} {#1}
      \tl_to_lowercase:n
        { \group_end: \str_put_right:Nn \l__runner_output_str { ^^@ } }
    }
\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`).

```
\cs_new:Npn \__runner_break:n #1 #2 \__runner_break_point:n #3 { #3 #1 }
\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.

```
\cs_new_protected:Npn \__runner:nnnn #1#2#3#4
  {
    \__runner_load_language:nT {#1}
      {
        \group_begin:
          \keys_set:nn { runner } {#2}
          \str_set:Nn \l__runner_program_str {#3}
          \str_set:Nn \l__runner_input_str {#4}
          \tl_clear:N \l__runner_output_str
          \use:c { __runner_#1_run: }
          \__runner_finish:w
        \group_end:
```

```
      }
  }
```

(*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.

```
\prg_new_conditional:Npnn \__runner_load_language:n #1 { T }
  {
    \cs_if_exist:cTF { __runner_#1_run: }
      { \prg_return_true: }
      {
        \file_if_exist:nTF { runner - #1 .ldf }
          {
            \group_begin:
              \ExplSyntaxOn
              \file_input:n { runner - #1 .ldf }
            \group_end:
            \cs_if_exist:cTF { __runner_#1_run: }
              { \prg_return_true: }
              {
                \msg_error:nnn { runner } { defective-ldf } {#1}
                \prg_return_false:
              }
          }
          {
            \clist_if_in:NnTF \g_runner_languages_clist {#1}
              { \msg_error:nnn { runner } { missing-ldf } {#1} }
              { \msg_error:nnn { runner } { unknown-language } {#1} }
            \prg_return_false:
          }
      }
  }
```

(*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`.

```
\cs_new_protected_nopar:Npn \__runner_finish:w \group_end:
  {
    \use:x
      {
        \group_end:
        \tl_if_empty:NTF \l__runner_output_key_tl
          { \l__runner_output_str }
          {
            \tl_set:Nn \exp_not:o \l__runner_output_key_tl
              { \l__runner_output_str }
          }
      }
```

```
  }
```

(*End definition for* `\__runner_finish:w`.)

## A.6 Keys

```
\keys_define:nn { runner }
  {
    output .tl_set:N = \l__runner_output_key_tl
  }
```

## 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.

```
\cs_new_protected:Npn \__runner_process_lang_name:n #1
  {
    \str_set:Nn \l__runner_tmpa_tl {#1}
    \tl_remove_all:Nn \l__runner_tmpa_tl { ~ }
    \tl_set:Nx \ProcessedArgument
      { \exp_args:NV \tl_expandable_lowercase:n \l__runner_tmpa_tl }
  }
```

(*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`.

```
\NewDocumentCommand {\runner}
  { > { \__runner_process_lang_name:n } m O{} +v +v }
  { \__runner:nnnn {#1} {#2} {#3} {#4} }
```

(*End definition for* `\runner`. *This function is documented on page* 3.)

## A.8 Messages

```
\msg_new:nnnn { runner } { unknown-language }
  { The~programming~language~'#1'~is~not~known~to~'runner'. }
  {
    The~code\\\\
    \iow_indent:n {#2}\\\\
    could~not~be~run,~because~there~is~not~support~for~'#1',~yet.
  }
\msg_new:nnnn { runner } { defective-ldf }
  { The~file~'runner-#1.ldf'~does~not~define~a~programming~language. }
  {
    The~language~definition~file~'runner-#1.ldf'~must~define~
    '\iow_char:N \\__runner_#1_run:'~to~enable~the~runner~package~
    to~run~programs~written~in~'#1'.
  }
\msg_new:nnnn { runner } { missing-ldf }
```

```
  { The~file~'runner-#1.ldf'~cannot~be~found. }
  {
    The~'#1'~programming~language~should~be~supported~by~the~
    'runner'~package,~but~this~requires~the~language~definition~
    file~'runner-#1.ldf',~which~is~nowhere~to~be~found.
  }
\msg_new:nnnn { runner } { negative-array-size }
  { The~array~'#1'~cannot~be~initialized~at~size~'#2'. }
  {
    This~is~probably~an~internal~error~in~the~runner~package.~
    Please~report~it.
  }
```

# B `brainfuck` implementation

## B.1 Variables

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

```
\int_new:N \l__runner_brainfuck_cell_int
\int_new:N \l__runner_brainfuck_instr_int
```

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

```
\prop_new:N \l__runner_brainfuck_match_prop
\seq_new:N \l__runner_brainfuck_match_seq
```

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

```
\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.

```
\cs_new_nopar:Npn \__runner_brainfuck_current_name:
  { 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.

```
\cs_new_protected_nopar:Npn \__runner_brainfuck_current_provide:
  {
    \cs_if_exist:cF { \__runner_brainfuck_current_name: }
      { \int_new:c { \__runner_brainfuck_current_name: } }
  }
```

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

```
\cs_new_nopar:Npn \__runner_brainfuck_current_value:
  {
    \cs_if_exist:cTF { \__runner_brainfuck_current_name: }
```

```
      { \int_use:c { \__runner_brainfuck_current_name: } }
      { 0 }
  }
```

## B.3 Structure to run the code

__runner_brainfuck_run:

```
\cs_new_protected_nopar:Npn \__runner_brainfuck_run:
  {
    \__runner_brainfuck_clean:
    \__runner_brainfuck_matches:
    \__runner_brainfuck_execute:
  }
```

__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.

```
\cs_new_protected_nopar:Npn \__runner_brainfuck_clean:
  {
    \tl_set:Nx \l__runner_program_str
      {
        \tl_map_function:NN
          \l__runner_program_str
          \__runner_brainfuck_clean_aux:n
      }
    \int_set:Nn \l__runner_brainfuck_length_int
      { \tl_count:N \l__runner_program_str }
  }
\cs_new:Npn \__runner_brainfuck_clean_aux:n #1
  { \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.

```
\cs_new_protected_nopar:Npn \__runner_brainfuck_matches:
  {
    \int_set:Nn \l__runner_brainfuck_instr_int { 1 }
    \tl_map_inline:Nn \l__runner_program_str
      {
        \cs_if_exist_use:c { __runner_brainfuck_matches_##1: }
        \int_incr:N \l__runner_brainfuck_instr_int
      }
  }
\cs_new_protected_nopar:cpn { __runner_brainfuck_matches_[: }
  {
    \seq_push:No \l__runner_brainfuck_match_seq
      { \int_use:N \l__runner_brainfuck_instr_int }
  }
\cs_new_protected_nopar:cpn { __runner_brainfuck_matches_]: }
  {
    \seq_pop:NN \l__runner_brainfuck_match_seq \l__runner_tmpa_tl
    \prop_put:NVV
```

```
      \l__runner_brainfuck_match_prop
      \l__runner_tmpa_tl
      \l__runner_brainfuck_instr_int
    \prop_put:NVV
      \l__runner_brainfuck_match_prop
      \l__runner_brainfuck_instr_int
      \l__runner_tmpa_tl
  }
```

`\__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`.

```
\cs_new_protected_nopar:Npn \__runner_brainfuck_execute:
  {
    \int_set:Nn \l__runner_brainfuck_instr_int { 1 }
    \int_until_do:nn
      { \l__runner_brainfuck_instr_int > \l__runner_brainfuck_length_int }
      {
        \exp_args:Nf \tl_map_inline:nn
          {
            \str_range:Nnn \l__runner_program_str
              { \l__runner_brainfuck_instr_int }
              { -1 }
          }
          {
            \use:c { __runner_brainfuck_##1: }
            \int_incr:N \l__runner_brainfuck_instr_int
          }
      }
  }
```

## B.4 The eight language commands

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

```
\cs_new_protected_nopar:cpn { __runner_brainfuck_>: }
  { \int_incr:N \l__runner_brainfuck_cell_int }
\cs_new_protected_nopar:cpn { __runner_brainfuck_<: }
  { \int_decr:N \l__runner_brainfuck_cell_int }
```

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

```
\cs_new_protected_nopar:cpn { __runner_brainfuck_+: }
  {
```

```
    \__runner_brainfuck_current_provide:
    \int_incr:c { \__runner_brainfuck_current_name: }
  }
\cs_new_protected_nopar:cpn { __runner_brainfuck_-: }
  {
    \__runner_brainfuck_current_provide:
    \int_decr:c { \__runner_brainfuck_current_name: }
  }
```

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

```
\cs_new_protected_nopar:cpn { __runner_brainfuck_.: }
  { \__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.

```
\cs_new_protected_nopar:cpn { __runner_brainfuck_,: }
  {
    \__runner_brainfuck_current_provide:
    \tl_if_empty:NTF \l__runner_input_str
      { \int_set_eq:cN { \__runner_brainfuck_current_name: } \c_minus_one }
      {
        \tl_set:Nx \l__runner_tmpa_tl
          { \str_head:N \l__runner_input_str }
        \tl_set:Nx \l__runner_input_str
          { \str_tail:N \l__runner_input_str }
        \int_set:cn { \__runner_brainfuck_current_name: }
          { \exp_after:wN ‘ \l__runner_tmpa_tl }
      }
  }
```

`\__runner_brainfuck_[:` `\__runner_brainfuck_]:` `\__runner_brainfuck_jump:N` 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.

```
\cs_new_protected_nopar:cpn { __runner_brainfuck_[: }
  {
    \int_compare:nNnT \__runner_brainfuck_current_value: = \c_zero
      { \__runner_brainfuck_jump: }
  }
\cs_new_protected_nopar:cpn { __runner_brainfuck_]: }
  {
    \int_compare:nNnF \__runner_brainfuck_current_value: = \c_zero
      { \__runner_brainfuck_jump: }
  }
\cs_new_protected_nopar:Npn \__runner_brainfuck_jump:
  {
    \prop_get:NVN
      \l__runner_brainfuck_match_prop
      \l__runner_brainfuck_instr_int
```

```
      \l__runner_tmpa_tl
    \int_set:Nn \l__runner_brainfuck_instr_int { \l__runner_tmpa_tl }
    \tl_map_break:
  }
```

# 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.

```
\cs_generate_variant:Nn \prop_gput:Nnn { Nx }
\cs_generate_variant:Nn \prop_get:NnNTF { Nx }
```

## C.1 Memory, dictionary and stacks

The data space is stored in TEX'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 TEX is not fast; and we would need to chose an encoding for Unicode (this is still needed in pdfTEX, 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 TEX 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 intepretation semantics, and the compilation semantics follows in the next `\toks` register.

### C.1.1 Data space and data stack

`\g__runner_forth_data_array`
`\g__runner_forth_data_size_int`
`\l__runner_forth_data_here_int`
`\l__runner_forth_data_top_int`

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` $\leq$ `data_top` must always hold.

```
\int_new:N \g__runner_forth_data_size_int
```

```
\int_gset:Nn \g__runner_forth_data_size_int { 65536 }
\int_new:N \l__runner_forth_data_here_int
\int_new:N \l__runner_forth_data_top_int
\__runner_array_new:Nn \g__runner_forth_data_array
  \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.

```
\cs_new_protected:Npn \__runner_forth_put_here:n #1
  {
    \__runner_array_gset:Nnn \g__runner_forth_data_array
      { \l__runner_forth_data_here_int } {#1}
    \int_incr:N \l__runner_forth_data_here_int
  }
```

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

```
\cs_new_protected:Npn \__runner_forth_pop_int:N #1
  {
    \int_compare:nNnTF
      \l__runner_forth_data_top_int < \g__runner_forth_data_size_int
      {
        \int_set:Nn #1
          {
            \__runner_array_item:NN \g__runner_forth_data_array
              \l__runner_forth_data_top_int
          }
        \int_incr:N \l__runner_forth_data_top_int
      }
      { \msg_error:nn { runner/forth } { empty-stack } }
  }
\cs_new_protected:Npn \__runner_forth_pop_int:NN #1#2
  {
    \__runner_forth_pop_int:N #2
    \__runner_forth_pop_int:N #1
  }
\cs_new_protected:Npn \__runner_forth_pop_int:NNN #1#2#3
  {
    \__runner_forth_pop_int:N #3
    \__runner_forth_pop_int:N #2
    \__runner_forth_pop_int:N #1
  }
\cs_new_protected:Npn \__runner_forth_pop_int:NNNN #1#2#3#4
  {
    \__runner_forth_pop_int:N #4
    \__runner_forth_pop_int:N #3
    \__runner_forth_pop_int:N #2
    \__runner_forth_pop_int:N #1
  }
```

__runner_forth_push:n
__runner_forth_push:nn
__runner_forth_push:nnn
__runner_forth_push:nnnn
__runner_forth_push:nnnnn
__runner_forth_push:nnnnnn

Push data on the top of the data stack, making sure that we do not step onto the data space.

```
\cs_new_protected:Npn \__runner_forth_push:n #1
  {
    \int_compare:nNnTF
      \l__runner_forth_data_top_int > \l__runner_forth_data_here_int
      {
        \int_decr:N \l__runner_forth_data_top_int
        \__runner_array_gset:Nnn \g__runner_forth_data_array
          \l__runner_forth_data_top_int
          {#1}
      }
      { \msg_error:nn { runner/forth } { out-of-memory } }
  }
\cs_new_protected:Npn \__runner_forth_push:nn #1#2
  {
    \__runner_forth_push:n {#1}
    \__runner_forth_push:n {#2}
  }
\cs_new_protected:Npn \__runner_forth_push:nnn #1#2#3
  {
    \__runner_forth_push:n {#1}
    \__runner_forth_push:n {#2}
    \__runner_forth_push:n {#3}
  }
\cs_new_protected:Npn \__runner_forth_push:nnnn #1#2#3#4
  {
    \__runner_forth_push:n {#1}
    \__runner_forth_push:n {#2}
    \__runner_forth_push:n {#3}
    \__runner_forth_push:n {#4}
  }
\cs_new_protected:Npn \__runner_forth_push:nnnnn #1#2#3#4#5
  {
    \__runner_forth_push:n {#1}
    \__runner_forth_push:n {#2}
    \__runner_forth_push:n {#3}
    \__runner_forth_push:n {#4}
    \__runner_forth_push:n {#5}
  }
\cs_new_protected:Npn \__runner_forth_push:nnnnnn #1#2#3#4#5#6
  {
    \__runner_forth_push:n {#1}
    \__runner_forth_push:n {#2}
    \__runner_forth_push:n {#3}
    \__runner_forth_push:n {#4}
    \__runner_forth_push:n {#5}
    \__runner_forth_push:n {#6}
  }
```

\l__runner_forth_base_address_int
__runner_forth_base:

The address at which the base of the number system is stored, and a function to retreive the base.

```
\int_new:N \l__runner_forth_base_address_int
\cs_new:Npn \__runner_forth_base:
  {
    \__runner_array_item:NN \g__runner_forth_data_array
      \l__runner_forth_base_address_int
  }
```

### C.1.2 Other stacks

\l__runner_forth_flow_seq
\l__runner_forth_return_seq

The remaining stacks.

```
\seq_new:N \l__runner_forth_flow_seq
\seq_new:N \l__runner_forth_return_seq
```

### C.1.3 Dictionary

\l__runner_forth_words_prop
\g__runner_forth_core_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 $d = 3$.

```
\prop_new:N \l__runner_forth_words_prop
\prop_new:N \g__runner_forth_core_words_prop
```

\l__runner_forth_toks_int
\g__runner_forth_core_toks_int

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

```
\int_new:N \l__runner_forth_toks_int
\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.

```
\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

```
\cs_new_protected:Npn \__runner_forth_new_core:nn #1#2
  {
    \tl_gput_right:Nx \g__runner_forth_init_tl
      {
        \tex_toks:D \int_use:N \g__runner_forth_core_toks_int
          { \exp_not:n {#2} }
      }
    \prop_gput:Nnx \g__runner_forth_core_words_prop {#1}
```

```
      { \int_eval:n { 100000 + \g__runner_forth_core_toks_int } }
    \int_gincr:N \g__runner_forth_core_toks_int
  }
\cs_new_protected:Npn \__runner_forth_new_immediate_core:nn #1#2
  {
    \tl_gput_right:Nx \g__runner_forth_init_tl
      {
        \tex_toks:D \int_use:N \g__runner_forth_core_toks_int
          { \exp_not:n {#2} }
      }
    \prop_gput:Nnx \g__runner_forth_core_words_prop {#1}
      { \int_eval:n { 200000 + \g__runner_forth_core_toks_int } }
    \int_gincr:N \g__runner_forth_core_toks_int
  }
\cs_new_protected:Npn \__runner_forth_new_special_core:nnn #1#2#3
  {
    \tl_gput_right:Nx \g__runner_forth_init_tl
      {
        \tex_toks:D \int_use:N \g__runner_forth_core_toks_int
          { \exp_not:n {#2} }
        \tex_toks:D \int_eval:n { \g__runner_forth_core_toks_int + 1 }
          { \exp_not:n {#3} }
      }
    \prop_gput:Nnx \g__runner_forth_core_words_prop {#1}
      { \int_eval:n { 300000 + \g__runner_forth_core_toks_int } }
    \int_gadd:Nn \g__runner_forth_core_toks_int \c_two
  }
\cs_new_protected:Npn \__runner_forth_new_compilation_core:nn #1#2
  {
    \__runner_forth_new_special_core:nnn {#1}
      { \msg_error:nnn { runner/forth } { no-interpretation } {#1} }
      {#2}
  }
```

`\__runner_forth_core_alias:nn`

```
\cs_new_protected:Npn \__runner_forth_core_alias:nn #1#2
  {
    \prop_get:NnN \g__runner_forth_core_words_prop {#2} \l__runner_tmpa_tl
    \prop_gput:NnV \g__runner_forth_core_words_prop {#1} \l__runner_tmpa_tl
  }
```

`\__runner_forth_new_word:nn`
`\__runner_forth_new_word:nx`

```
\cs_new_protected:Npn \__runner_forth_new_word:nn #1#2
  {
    \tex_toks:D \l__runner_forth_toks_int {#2}
    \prop_put:Nnx \l__runner_forth_words_prop {#1}
      { \int_eval:n { 100000 + \l__runner_forth_toks_int } }
    \int_incr:N \l__runner_forth_toks_int
  }
\cs_generate_variant:Nn \__runner_forth_new_word:nn { nx }
```

## C.2 Helpers

### C.2.1 Input

__runner_forth_input_spaces:
__runner_forth_input_spaces_aux:w

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

```
\cs_new_protected_nopar:Npn \__runner_forth_input_spaces:
  {
    \exp_after:wN \use:nn
    \exp_after:wN \__runner_forth_input_spaces_aux:w
    \l__runner_forth_input_str \q_stop
  }
\cs_new_protected_nopar:Npn \__runner_forth_input_spaces_aux:w #1 \q_stop
  { \tl_set:Nn \l__runner_forth_input_str {#1} }
```

__runner_forth_input_until:nn
__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.

```
\cs_new_protected:Npn \__runner_forth_input_until:nn #1#2
  {
    \tl_if_in:NnF \l__runner_forth_input_str { #1 }
      { \tl_put_right:Nn \l__runner_forth_input_str { #1 } }
    \cs_set:Npn \__runner_tmp:w ##1 #1 ##2 \q_stop
      {
        \str_set:Nn \l__runner_forth_input_str {##2}
        #2
      }
    \exp_after:wN \__runner_tmp:w \l__runner_forth_input_str \q_stop
  }
\cs_new_protected:Npn \__runner_forth_input_until:nN #1#2
  { \__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.

```
\cs_new_protected:Npn \__runner_forth_input_discard:N #1
  {
    \exp_args:No \tl_if_head_eq_charcode:nNT
      \l__runner_forth_input_str #1
      {
        \str_set:Nx \l__runner_forth_input_str
          { \str_tail:N \l__runner_forth_input_str }
        \__runner_forth_input_discard:N #1
      }
  }
```

### C.2.2 Numbers

\c__runner_forth_mod_int Cells can take up to $2^{24}$ values.

```
\int_const:Nn \c__runner_forth_mod_int { 16777216 }
```

\l__runner_forth_push_int
\l__runner_forth_push_fp When turning a signed number to an unsigned value for pushing onto the stack, one cannot use \l__runner_tmpa_int, as it may be used by the caller. Similarly for outputting floating points modulo some number.

```
\int_new:N \l__runner_forth_push_int
\fp_new:N \l__runner_forth_push_fp
```

__runner_forth_signed:N Turn the unsigned #1 into a signed number, by subtracting $2^{24}$ if it is not less than $2^{23}$.

```
\cs_new_protected:Npn \__runner_forth_signed:N #1
  {
    \int_compare:nNnF #1 < { \c__runner_forth_mod_int / \c_two }
      { \int_sub:Nn #1 \c__runner_forth_mod_int }
  }
```

__runner_forth_push_signed:n 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.

```
\cs_new_protected:Npn \__runner_forth_push_signed:n #1
  {
    \int_set:Nn \l__runner_forth_push_int {#1}
    \__runner_forth_push:n
      {
        \l__runner_forth_push_int
        \int_compare:nNnF \l__runner_forth_push_int > \c_minus_one
          { + \c__runner_forth_mod_int }
      }
  }
```

__runner_forth_push_mod:n Given a non-negative integer, push onto the stack its residue modulo $2^{24}$.

```
\cs_new_protected:Npn \__runner_forth_push_mod:n #1
  {
    \__runner_forth_push:n
      { \int_mod:nn {#1} { \c__runner_forth_mod_int } }
  }
```

__runner_forth_get_number:nN*TF*

```
\prg_new_protected_conditional:Npnn \__runner_forth_get_number:nN #1#2 { TF }
  {
    \int_set:Nn \l__runner_tmpb_int { \__runner_forth_base: }
    \int_set:Nn \l__runner_tmpa_int
      { \int_min:nn { `9 } { `0 + \l__runner_tmpb_int - 1 } }
    \int_set:Nn \l__runner_tmpb_int
      { `A + \l__runner_tmpb_int - 11 }
    \bool_set_true:N \l__runner_tmpa_bool
    \tl_map_inline:nn {#1}
```

```
      {
        \int_compare:nF { `0 <= `##1 <= \l__runner_tmpa_int }
          {
            \int_compare:nF { `A <= `##1 <= \l__runner_tmpb_int }
              {
                \bool_set_false:N \l__runner_tmpa_bool
                \tl_map_break:
              }
          }
      }
    \bool_if:NTF \l__runner_tmpa_bool
      {
        \tl_set:Nx #2 { \int_from_base:nn {#1} { \__runner_forth_base: } }
        \prg_return_true:
      }
      { \prg_return_false: }
  }
```

`\__runner_forth_push_fp:n`
`\__runner_forth_push_fp_mod:n`

```
\cs_new_protected:Npn \__runner_forth_push_fp:n #1
  { \__runner_forth_push:n { \fp_to_int:n {#1} } }
\cs_new_protected:Npn \__runner_forth_push_fp_mod:n #1
  {
    \fp_set:Nn \l__runner_forth_push_fp { round0(#1) }
    \__runner_forth_push_fp:n
      {
        \l__runner_forth_push_fp - \c__runner_forth_mod_int
            * round- ( \l__runner_forth_push_fp / \c__runner_forth_mod_int )
      }
  }
```

### 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.

```
\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.

```
\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.

```
\tl_new:N \l__runner_forth_def_tl
```

__runner_forth_def_put_right:n
__runner_forth_def_put_right:x
__runner_forth_def_put_right_x:n

Helpers to add material to the definition, with various types of expansion.

```
\cs_new_protected:Npn \__runner_forth_def_put_right:n #1
  { \tl_put_right:Nn \l__runner_forth_def_tl { \exp_not:n {#1} } }
\cs_generate_variant:Nn \__runner_forth_def_put_right:n { x }
\cs_new_protected:Npn \__runner_forth_def_put_right_x:n #1
  { \tl_put_right:Nn \l__runner_forth_def_tl {#1} }
```

\l__runner_forth_def_nesting_int

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.

```
\int_new:N \l__runner_forth_def_nesting_int
```

\l__runner_forth_def_name_str

The last word that was defined: this is set just after the word's meaning is changed, after the definition is fully read.

```
\tl_new:N \l__runner_forth_def_name_str
```

CONSTANT

```
\__runner_forth_new_core:nn { CONSTANT }
  {
    \__runner_forth_pop_int:N \l__runner_tmpa_int
    \__runner_forth_input_spaces:
    \__runner_forth_input_until:nn { ~ }
      {
        \__runner_forth_new_word:nx {#1}
          { \__runner_forth_push:n { \int_use:N \l__runner_tmpa_int } }
      }
  }
```

VARIABLE

```
\__runner_forth_new_core:nn { VARIABLE }
  {
    \__runner_forth_input_spaces:
    \__runner_forth_input_until:nn { ~ }
      {
        \__runner_forth_new_word:nx {#1}
          {
            \__runner_forth_push:n
              { \int_use:N \l__runner_forth_data_here_int }
          }
        \int_incr:N \l__runner_forth_data_here_int
      }
  }
```

:

Read a word, and feed it to `\__runner_forth_def:n`, responsible for all the work.

```
\__runner_forth_new_core:nn { : }
  {
    \__runner_forth_input_spaces:
    \__runner_forth_input_until:nN { ~ } \__runner_forth_def:n
  }
```

`\__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.

```
\cs_new_protected:Npn \__runner_forth_def:n #1
  {
    \seq_push:NV \l__runner_forth_flow_seq \l__runner_forth_def_tl
    \tl_clear:N \l__runner_forth_def_tl
    \bool_until_do:Nn \l__runner_forth_stop_bool
      {
        \__runner_forth_input_spaces:
        \__runner_forth_input_until:nN { ~ } \__runner_forth_compile:n
      }
    \bool_set_false:N \l__runner_forth_stop_bool
    \int_compare:nNnF \l__runner_forth_def_nesting_int = \c_zero
      {
        \msg_error:nn { runner/forth } { too-many-ifs }
        \prg_replicate:nn \l__runner_forth_def_nesting_int
          { \__runner_forth_def_put_right_x:n { \if_false: { \fi: } } }
      }
    \__runner_forth_new_word:nx {#1} { \l__runner_forth_def_tl }
    \str_set:Nn \l__runner_forth_def_name_str {#1}
    \seq_pop:NN \l__runner_forth_flow_seq \l__runner_forth_def_tl
  }
```

`\__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.

```
\cs_new_protected:Npn \__runner_forth_compile:n #1
  {
    \prop_get:NnNTF \l__runner_forth_words_prop {#1} \l__runner_tmpa_tl
      { \exp_after:wN \__runner_forth_compile:Nw \l__runner_tmpa_tl \q_stop }
      {
        \__runner_forth_get_number:nNTF {#1} \l__runner_tmpa_tl
          {
            \__runner_forth_def_put_right:x
              { \__runner_forth_push:n { \l__runner_tmpa_tl } }
          }
          { \msg_error:nnn { runner/forth } { unknown-word } {#1} }
      }
  }
\cs_new_protected:Npn \__runner_forth_compile:Nw #1#2 \q_stop
  {
    \int_case:nnn {#1}
      {
```

```
        { 1 }
          {
            \__runner_forth_def_put_right:n
              { \tex_the:D \tex_toks:D #2 \scan_stop: }
          }
        { 2 } { \tex_the:D \tex_toks:D #2 \scan_stop: }
      }
      { \tex_the:D \tex_toks:D \int_eval:n { #2 + 1 } \scan_stop: }
  }
```

; 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.

```
\__runner_forth_new_compilation_core:nn { ; }
  { \bool_set_true:N \l__runner_forth_stop_bool }
```

IMMEDIATE

```
\__runner_forth_new_core:nn { IMMEDIATE }
  {
    \prop_get:NVNTF
      \l__runner_forth_words_prop
      \l__runner_forth_def_name_str
      \l__runner_tmpa_tl
      {
        \tl_set:Nx \l__runner_tmpa_tl
          { \int_eval:n { 100000 + \l__runner_tmpa_tl } }
        \prop_put:NVV
          \l__runner_forth_words_prop
          \l__runner_forth_def_name_str
          \l__runner_tmpa_tl
      }
      {
        \msg_error:nnx { runner/forth } { no-def-immediate }
          { \l__runner_forth_def_name_str }
      }
  }
```

LITERAL

```
\__runner_forth_new_compilation_core:nn { LITERAL }
  {
    \__runner_forth_pop_int:N \l__runner_tmpa_int
    \__runner_forth_def_put_right:x
      { \__runner_forth_push:n { \int_use:N \l__runner_tmpa_int } }
  }
```

POSTPONE

__runner_forth_postpone:Nw

```
\__runner_forth_new_compilation_core:nn { POSTPONE }
  {
    \__runner_forth_input_spaces:
    \__runner_forth_input_until:nn { ~ }
```

```
      {
        \prop_get:NnN \l__runner_forth_words_prop {#1} \l__runner_tmpa_tl
        \__runner_forth_def_put_right:x
          {
            \exp_after:wN \__runner_forth_postpone:Nw
              \l__runner_tmpa_tl \q_stop
          }
      }
  }
\cs_new:Npn \__runner_forth_postpone:Nw #1#2 \q_stop
  {
    \int_case:nnn {#1}
      {
        { 1 }
          {
            \__runner_forth_def_put_right:n
              { \exp_not:N \tex_the:D \tex_toks:D #2 \scan_stop: }
          }
        { 2 } { \exp_not:N \tex_the:D \tex_toks:D #2 \scan_stop: }
      }
      {
        \exp_not:N \tex_the:D \tex_toks:D
          \int_eval:n { #2 + 1 } \scan_stop:
      }
  }
```

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.

```
\__runner_forth_new_compilation_core:nn { RECURSE }
  {
    \__runner_forth_def_put_right_x:n
      {
        \exp_not:N \tex_the:D \tex_toks:D
        \int_use:N \l__runner_forth_toks_int \scan_stop:
      }
  }
```

### 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.

```
\__runner_forth_new_core:nn { ' }
  {
    \__runner_forth_input_spaces:
    \__runner_forth_input_until:nn { ~ }
      {
        \prop_get:NnN \l__runner_forth_words_prop {#1} \l__runner_tmpa_tl
        \__runner_forth_push:n
```

```
            { \exp_after:wN \use_none:n \l__runner_tmpa_tl }
        }
    }
```

`[']` 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.

```
\__runner_forth_new_compilation_core:nn { ['] }
  {
    \__runner_forth_input_spaces:
    \__runner_forth_input_until:nn { ~ }
      {
        \prop_get:NnN \l__runner_forth_words_prop {#1} \l__runner_tmpa_tl
        \__runner_forth_def_put_right:x
          {
            \__runner_forth_push:n
              { \exp_after:wN \use_none:n \l__runner_tmpa_tl }
          }
      }
  }
```

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

```
\__runner_forth_new_core:nn { EXECUTE }
  {
    \__runner_forth_pop_int:N \l__runner_tmpa_int
    \tex_the:D \tex_toks:D \l__runner_tmpa_int
  }
```

### 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.

```
\__runner_forth_new_core:nn { DEPTH }
  {
    \__runner_forth_push:n
      {
        \g__runner_forth_data_size_int
        - \l__runner_forth_data_top_int - \c_one
      }
  }
```

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

```
\__runner_forth_new_compilation_core:nn { >R }
  {
    \__runner_forth_pop_int:N \l__runner_tmpa_int
    \seq_push:NV \l__runner_forth_return_seq \l__runner_tmpa_int
  }
```

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

```
\__runner_forth_new_core:nn { R> }
  {
    \seq_pop:NN \l__runner_forth_return_seq \l__runner_tmpa_tl
    \__runner_forth_push:n { \l__runner_tmpa_tl }
  }
```

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

```
\__runner_forth_new_core:nn { R@ }
  {
    \seq_get:NN \l__runner_forth_return_seq \l__runner_tmpa_tl
    \__runner_forth_push:n { \l__runner_tmpa_tl }
  }
```

`?DUP` Duplicate top of stack if non-zero.

```
\__runner_forth_new_core:nn { ?DUP }
  {
    \__runner_forth_pop_int:N \l__runner_tmpa_int
    \int_compare:nNnTF \l__runner_tmpa_int = \c_zero
      { \__runner_forth_push:n { \c_zero } }
      {
        \__runner_forth_push:nn
          { \l__runner_tmpa_int } { \l__runner_tmpa_int }
      }
  }
```

`DROP` `2DROP` Pop stack, once or twice.

```
\__runner_forth_new_core:nn { DROP }
  { \__runner_forth_pop_int:N \l__runner_tmpa_int }
\__runner_forth_new_core:nn { 2DROP }
  { \__runner_forth_pop_int:NN \l__runner_tmpa_int \l__runner_tmpb_int }
```

`DUP` `2DUP` Turn $x$ to $x, x$, or $x_1, x_2$ to $x_1, x_2, x_1, x_2$.

```
\__runner_forth_new_core:nn { DUP }
  {
    \__runner_forth_pop_int:N \l__runner_tmpa_int
    \__runner_forth_push:nn { \l__runner_tmpa_int } { \l__runner_tmpa_int }
  }
\__runner_forth_new_core:nn { 2DUP }
  {
    \__runner_forth_pop_int:NN \l__runner_tmpa_int \l__runner_tmpb_int
    \__runner_forth_push:nnnn
      { \l__runner_tmpa_int } { \l__runner_tmpb_int }
      { \l__runner_tmpa_int } { \l__runner_tmpb_int }
  }
```

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

```
\__runner_forth_new_core:nn { OVER }
  {
    \__runner_forth_pop_int:NN \l__runner_tmpa_int \l__runner_tmpb_int
    \__runner_forth_push:nnn
      { \l__runner_tmpa_int } { \l__runner_tmpb_int } { \l__runner_tmpa_int }
  }
\__runner_forth_new_core:nn { 2OVER }
  {
    \__runner_forth_pop_int:NNNN
      \l__runner_tmpa_int \l__runner_tmpb_int
      \l__runner_tmpc_int \l__runner_tmpd_int
    \__runner_forth_push:nnnnnn
      { \l__runner_tmpa_int } { \l__runner_tmpb_int }
      { \l__runner_tmpc_int } { \l__runner_tmpd_int }
      { \l__runner_tmpa_int } { \l__runner_tmpb_int }
  }
```

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

```
\__runner_forth_new_core:nn { ROT }
  {
    \__runner_forth_pop_int:NNN
      \l__runner_tmpa_int \l__runner_tmpb_int \l__runner_tmpc_int
    \__runner_forth_push:nnn
      { \l__runner_tmpb_int } { \l__runner_tmpc_int } { \l__runner_tmpa_int }
  }
```

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

```
\__runner_forth_new_core:nn { SWAP }
  {
    \__runner_forth_pop_int:NN \l__runner_tmpa_int \l__runner_tmpb_int
    \__runner_forth_push:nn { \l__runner_tmpb_int } { \l__runner_tmpa_int }
  }
\__runner_forth_new_core:nn { 2SWAP }
  {
    \__runner_forth_pop_int:NNNN
      \l__runner_tmpa_int \l__runner_tmpb_int
      \l__runner_tmpc_int \l__runner_tmpd_int
    \__runner_forth_push:nn
      { \l__runner_tmpc_int } { \l__runner_tmpd_int }
      { \l__runner_tmpa_int } { \l__runner_tmpb_int }
  }
```

### C.3.4 Comparison words

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

```
\__runner_forth_new_core:nn { 0< }
  {
    \__runner_forth_pop_int:N \l__runner_tmpa_int
```

```
    \int_compare:nNnTF \l__runner_tmpa_int < \c_zero
      { \__runner_forth_push:n \c_minus_one }
      { \__runner_forth_push:n \c_zero }
  }
```

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

```
\__runner_forth_new_core:nn { 0= }
  {
    \__runner_forth_pop_int:N \l__runner_tmpa_int
    \int_compare:nNnTF \l__runner_tmpa_int = \c_zero
      { \__runner_forth_push:n \c_minus_one }
      { \__runner_forth_push:n \c_zero }
  }
```

`<` `=` `>` Comparisons. Push $-1$ if true, 0 if false.

```
\__runner_forth_new_core:nn { < }
  {
    \__runner_forth_pop_int:NN \l__runner_tmpa_int \l__runner_tmpb_int
    \int_compare:nNnTF \l__runner_tmpa_int < \l__runner_tmpb_int
      { \__runner_forth_push:n \c_minus_one }
      { \__runner_forth_push:n \c_zero }
  }
\__runner_forth_new_core:nn { = }
  {
    \__runner_forth_pop_int:NN \l__runner_tmpa_int \l__runner_tmpb_int
    \int_compare:nNnTF \l__runner_tmpa_int = \l__runner_tmpb_int
      { \__runner_forth_push:n \c_minus_one }
      { \__runner_forth_push:n \c_zero }
  }
\__runner_forth_new_core:nn { > }
  {
    \__runner_forth_pop_int:NN \l__runner_tmpa_int \l__runner_tmpb_int
    \int_compare:nNnTF \l__runner_tmpa_int > \l__runner_tmpb_int
      { \__runner_forth_push:n \c_minus_one }
      { \__runner_forth_push:n \c_zero }
  }
```

### C.3.5 Arithmetic words

`1+` `1-` Increment or decrement the top of the stack.

```
\__runner_forth_new_core:nn { 1+ }
  {
    \__runner_forth_pop_int:N \l__runner_tmpa_int
    \__runner_forth_push:n { \l__runner_tmpa_int + 1 }
  }
\__runner_forth_new_core:nn { 1- }
  {
    \__runner_forth_pop_int:N \l__runner_tmpa_int
    \__runner_forth_push:n { \l__runner_tmpa_int - 1 }
  }
```

`2*` `2/` Multiply or divide the top of the stack by 2. Neither TEX's rounding division nor LATEX3's truncating division do the right thing there, as we want floored division ($(-1)/2 = -1$), so we distinguish the even and odd cases.

```
\__runner_forth_new_core:nn { 2* }
  {
    \__runner_forth_pop_int:N \l__runner_tmpa_int
    \__runner_forth_push:n { \l__runner_tmpa_int * 2 }
  }
\__runner_forth_new_core:nn { 2/ }
  {
    \__runner_forth_pop_int:N \l__runner_tmpa_int
    \int_if_even:nTF { \l__runner_tmpa_int }
      { \__runner_forth_push:n { \l__runner_tmpa_int / 2 } }
      { \__runner_forth_push:n { (\l__runner_tmpa_int - 1) / 2 } }
  }
```

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

```
\__runner_forth_new_core:nn { ABS }
  {
    \__runner_forth_pop_int:N \l__runner_tmpa_int
    \__runner_forth_signed:N \l__runner_tmpa_int
    \__runner_forth_push:n { \int_abs:n { \l__runner_tmpa_int } }
  }
```

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

```
\__runner_forth_new_core:nn { MAX }
  {
    \__runner_forth_pop_int:NN \l__runner_tmpa_int \l__runner_tmpb_int
    \__runner_forth_signed:N \l__runner_tmpa_int
    \__runner_forth_signed:N \l__runner_tmpb_int
    \__runner_forth_push_signed:n
      { \int_max:nn { \l__runner_tmpa_int } { \l__runner_tmpb_int } }
  }
\__runner_forth_new_core:nn { MIN }
  {
    \__runner_forth_pop_int:NN \l__runner_tmpa_int \l__runner_tmpb_int
    \__runner_forth_signed:N \l__runner_tmpa_int
    \__runner_forth_signed:N \l__runner_tmpb_int
    \__runner_forth_push_signed:n
      { \int_min:nn { \l__runner_tmpa_int } { \l__runner_tmpb_int } }
  }
```

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

```
\__runner_forth_new_core:nn { ABS }
  {
    \__runner_forth_pop_int:N \l__runner_tmpa_int
    \__runner_forth_signed:N \l__runner_tmpa_int
    \__runner_forth_push_signed:n { - \l__runner_tmpa_int }
```

```
}
```

`-` `+` 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.

```
\__runner_forth_new_core:nn { - }
  {
    \__runner_forth_pop_int:NN \l__runner_tmpa_int \l__runner_tmpb_int
    \__runner_forth_push_signed:n
      { \l__runner_tmpa_int - \l__runner_tmpb_int }
  }
\__runner_forth_new_core:nn { + }
  {
    \__runner_forth_pop_int:NN \l__runner_tmpa_int \l__runner_tmpb_int
    \__runner_forth_push_signed:n
      {
        \l__runner_tmpa_int + \l__runner_tmpb_int
        - \c__runner_forth_mod_int
      }
  }
```

`*` 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 $a \times b - 2^{24} \times \lfloor a \times b / 2^{24} \rfloor$.

```
\__runner_forth_new_core:nn { * }
  {
    \__runner_forth_pop_int:NN \l__runner_tmpa_int \l__runner_tmpb_int
    \__runner_forth_push_fp_mod:n
      { \l__runner_tmpa_int * \l__runner_tmpb_int }
  }
```

`/` 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.

```
\__runner_forth_new_core:nn { / }
  {
    \__runner_forth_pop_int:NN \l__runner_tmpa_int \l__runner_tmpb_int
    \__runner_forth_signed:N \l__runner_tmpa_int
    \__runner_forth_signed:N \l__runner_tmpb_int
    \__runner_forth_push_signed:n
      {
        \int_div_truncate:nn
          { \l__runner_tmpa_int } { \l__runner_tmpb_int }
      }
  }
```

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

```
\__runner_forth_new_core:nn { MOD }
  {
    \__runner_forth_pop_int:NN \l__runner_tmpa_int \l__runner_tmpb_int
    \__runner_forth_signed:N \l__runner_tmpa_int
    \__runner_forth_signed:N \l__runner_tmpb_int
    \__runner_forth_push_signed:n
      { \int_mod:nn { \l__runner_tmpa_int } { \l__runner_tmpb_int } }
  }
```

`/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.

```
\__runner_forth_new_core:nn { /MOD }
  {
    \__runner_forth_pop_int:NN \l__runner_tmpa_int \l__runner_tmpb_int
    \__runner_forth_signed:N \l__runner_tmpa_int
    \__runner_forth_signed:N \l__runner_tmpb_int
    \__runner_forth_push_signed:n
      { \int_mod:nn { \l__runner_tmpa_int } { \l__runner_tmpb_int } }
    \__runner_forth_push_signed:n
      {
        \int_div_truncate:nn
          { \l__runner_tmpa_int } { \l__runner_tmpb_int }
      }
  }
```

`*/` `*/MOD` 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.

```
\__runner_forth_new_core:nn { */ }
  {
    \__runner_forth_pop_int:NNN
      \l__runner_tmpa_int \l__runner_tmpb_int \l__runner_tmpc_int
    \__runner_forth_signed:N \l__runner_tmpa_int
    \__runner_forth_signed:N \l__runner_tmpb_int
    \__runner_forth_signed:N \l__runner_tmpc_int
    \__runner_forth_push_fp_mod:n
      { \l__runner_tmpa_int * \l__runner_tmpb_int / \l__runner_tmpc_int }
  }
\__runner_forth_new_core:nn { */MOD }
  {
    \__runner_forth_pop_int:NNN
      \l__runner_tmpa_int \l__runner_tmpb_int \l__runner_tmpc_int
    \__runner_forth_signed:N \l__runner_tmpa_int
    \__runner_forth_signed:N \l__runner_tmpb_int
```

```
    \__runner_forth_signed:N \l__runner_tmpc_int
    \fp_set:Nn \l__runner_tmpa_fp
      {
        round0 ( \l__runner_tmpa_int * \l__runner_tmpb_int
        / \l__runner_tmpc_int )
      }
    \__runner_forth_push_fp_mod:n
      {
        \l__runner_tmpa_int * \l__runner_tmpb_int
        - \l__runner_tmpc_int * \l__runner_tmpa_fp
      }
    \__runner_forth_push_fp_mod:n { \l__runner_tmpa_fp }
  }
```

### C.3.6 Storage words

ALIGN
ALIGNED

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

```
\__runner_forth_new_core:nn { ALIGN } { }
\__runner_forth_new_core:nn { ALIGNED }
  {
    \__runner_forth_pop_int:N \l__runner_tmpa_int
    \__runner_forth_push:n \l__runner_tmpa_int
  }
```

ALLOT

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

```
\__runner_forth_new_core:nn { ALLOT }
  {
    \__runner_forth_pop_int:N \l__runner_tmpa_int
    \int_add:Nn \l__runner_forth_data_here_int \l__runner_tmpa_int
    \int_compare:nNnTF \l__runner_forth_data_here_int < \c_zero
      {
        \msg_error:nnx { runner/forth } { out-of-bounds }
          { \int_use:N \l__runner_forth_data_here_int }
        \int_zero:N \l__runner_forth_data_here_int
      }
      {
        \int_compare:nNnTF
          \l__runner_forth_data_here_int > \l__runner_forth_data_top_int
          {
            \msg_error:nn { runner/forth } { out-of-memory }
            \int_set_eq:NN \l__runner_forth_data_here_int
              \l__runner_forth_data_top_int
          }
      }
  }
```

FILL

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

```
\__runner_forth_new_core:nn { FILL }
```

```
  {
    \__runner_forth_pop_int:NNN
      \l__runner_tmpa_int \l__runner_tmpb_int \l__runner_tmpc_int
    \int_step_inline:nnnn
      { \l__runner_tmpa_int }
      { 1 }
      { \l__runner_tmpa_int + \l__runner_tmpb_int - 1 }
      {
        \__runner_array_gset:Nnn \g__runner_forth_data_array {#1}
          { \l__runner_tmpc_int }
      }
  }
```

`HERE`

```
\__runner_forth_new_core:nn { HERE }
  { \__runner_forth_push:n { \l__runner_forth_data_here_int } }
```

`!` Store value at a given address.

```
\__runner_forth_new_core:nn { ! }
  {
    \__runner_forth_pop_int:NN \l__runner_tmpa_int \l__runner_tmpb_int
    \__runner_array_gset:Nnn \g__runner_forth_data_array
      { \l__runner_tmpb_int } { \l__runner_tmpa_int }
  }
```

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

```
\__runner_forth_new_core:nn { 2! }
  {
    \__runner_forth_pop_int:NNN
      \l__runner_tmpa_int \l__runner_tmpb_int \l__runner_tmpc_int
    \__runner_array_gset:Nnn \g__runner_forth_data_array
      { \l__runner_tmpc_int } { \l__runner_tmpb_int }
    \__runner_array_gset:Nnn \g__runner_forth_data_array
      { \l__runner_tmpc_int + 1 } { \l__runner_tmpa_int }
  }
```

`+!` Sum $a$ and the value at $b$ into the integer `\l__runner_tmpa_int`, then store that into the address at $b$.

```
\__runner_forth_new_core:nn { +! }
  {
    \__runner_forth_pop_int:NN \l__runner_tmpa_int \l__runner_tmpb_int
    \int_add:Nn \l__runner_tmpa_int
      {
        \__runner_array_item:Nn \g__runner_forth_data_array
          \l__runner_tmpb_int
      }
    \int_compare:nNnF \l__runner_tmpa_int < \c__runner_forth_mod_int
      { \int_sub:Nn \l__runner_tmpa_int \c__runner_forth_mod_int }
    \__runner_array_gset:Nnn \g__runner_forth_data_array
```

```
      { \l__runner_tmpb_int } { \l__runner_tmpa_int }
  }
```

Store value in a newly allocated cell, given by the data-space pointer "`here`".

```
\__runner_forth_new_core:nn { , }
  {
    \__runner_forth_pop_int:N \l__runner_tmpa_int
    \__runner_forth_put_here:n { \l__runner_tmpa_int }
  }
```

`@` Pop an address from the stack, and push the value $x$ of the corresponding memory cell.

```
\__runner_forth_new_core:nn { @ }
  {
    \__runner_forth_pop_int:N \l__runner_tmpa_int
    \__runner_forth_push:n
      {
        \__runner_array_item:Nn \g__runner_forth_data_array
          { \l__runner_tmpa_int }
      }
  }
```

`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.

```
\__runner_forth_new_core:nn { 2@ }
  {
    \__runner_forth_pop_int:N \l__runner_tmpa_int
    \__runner_forth_push:nn
      {
        \__runner_array_item:Nn \g__runner_forth_data_array
          { \l__runner_tmpa_int + 1 }
      }
      {
        \__runner_array_item:Nn \g__runner_forth_data_array
          { \l__runner_tmpa_int }
      }
  }
```

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

```
\__runner_forth_core_alias:nn { C! } { ! }
\__runner_forth_core_alias:nn { C, } { , }
\__runner_forth_core_alias:nn { C@ } { @ }
```

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

```
\__runner_forth_core_alias:nn { CELL+ } { 1+ }
\__runner_forth_core_alias:nn { CHAR+ } { 1+ }
\__runner_forth_core_alias:nn { CELLS } { ALIGNED }
\__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.

```
\__runner_forth_new_core:nn { . }
  {
    \__runner_forth_pop_int:N \l__runner_tmpa_int
    \__runner_output:x
      { \int_to_base:nn { \l__runner_tmpa_int } { \__runner_forth_base: } ~ }
  }
```

." 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.

```
\__runner_forth_new_compilation_core:nn { ." }
  {
    \__runner_forth_input_until:nn "
      { \__runner_forth_def_put_right:n { \__runner_output:n {#1} } }
  }
```

BASE

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

BL

```
\__runner_forth_new_core:nn { BL }
  { \__runner_forth_push:n { `\  } }
```

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

```
\__runner_forth_new_core:nn { CHAR }
  {
    \__runner_forth_input_spaces:
    \__runner_forth_input_until:nn { ~ }
      {
        \tl_set:Nx \l__runner_tmpa_tl { \str_head:n {#1} }
        \__runner_forth_push:n { \exp_after:wN ` \l__runner_tmpa_tl }
      }
  }
```

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

```
\__runner_forth_new_compilation_core:nn { [CHAR] }
  {
    \__runner_forth_input_spaces:
    \__runner_forth_input_until:nn { ~ }
      {
        \tl_set:Nx \l__runner_tmpa_tl { \str_head:n {#1} }
        \__runner_forth_def_put_right:x
          { \__runner_forth_push:n { \exp_after:wN ` \l__runner_tmpa_tl } }
```

```
      }
  }
```

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.

```
\__runner_forth_new_core:nn { COUNT }
  {
    \__runner_forth_pop_int:N \l__runner_tmpa_int
    \__runner_forth_push:nn
      { \l__runner_tmpa_int + 1 }
      {
        \__runner_array_item:Nn \g__runner_forth_data_array
          { \l__runner_tmpa_int }
      }
  }
```

CR

```
\__runner_forth_new_core:nn { CR }
  { \__runner_output:x { \iow_newline: } }
```

DECIMAL

```
\__runner_forth_new_core:nn { DECIMAL }
  {
    \__runner_array_gset:Nnn \g__runner_forth_data_array
      \l__runner_forth_base_address_int \c_ten
  }
```

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

```
\__runner_forth_new_core:nn { EMIT }
  {
    \__runner_forth_pop_int:N \l__runner_tmpa_int
    \__runner_output_char:n { \l__runner_tmpa_int }
  }
```

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.

```
\__runner_forth_new_compilation_core:nn { S" }
  {
    \__runner_forth_input_until:nn "
      { \tl_set:Nn \l__runner_tmpa_tl {#1} }
    \int_set_eq:NN \l__runner_tmpa_int \l__runner_forth_data_here_int
    \tl_replace_all:Nnn \l__runner_tmpa_tl { ~ } { { ~ } }
    \tl_map_inline:Nn \l__runner_tmpa_tl
      { \__runner_forth_put_here:n { `#1 } }
    \__runner_forth_def_put_right:x
      {
        \__runner_forth_push:n { \int_use:N \l__runner_tmpa_int }
```

```
        \__runner_forth_push:n
          {
            \int_eval:n
              { \l__runner_forth_data_here_int - \l__runner_tmpa_int }
          }
      }
  }
```

SPACE

```
\__runner_forth_new_core:nn { SPACE }
  { \__runner_output:n { ~ } }
```

SPACES

```
\__runner_forth_new_core:nn { SPACES }
  {
    \__runner_forth_pop_int:N \l__runner_tmpa_int
    \__runner_output:x { \prg_replicate:nn { \l__runner_tmpa_int } { ~ } }
  }
```

TYPE

```
\__runner_forth_new_core:nn { TYPE }
  {
    \__runner_forth_pop_int:NN \l__runner_tmpa_int \l__runner_tmpb_int
    \int_step_inline:nnnn
      { \l__runner_tmpa_int }
      { 1 }
      { \l__runner_tmpa_int + \l__runner_tmpb_int - 1 }
      {
        \__runner_output_char:n
          { \__runner_array_item:Nn \g__runner_forth_data_array {#1} }
      }
  }
```

WORD

```
\group_begin:
  \char_set_catcode_other:n { 0 }
  \__runner_forth_new_core:nn { WORD }
    {
      \__runner_forth_pop_int:N \l__runner_tmpa_int
      \__runner_forth_push:n { \l__runner_forth_data_here_int }
      \group_begin:
      \char_set_lccode:nn { 0 } { \l__runner_tmpa_int }
      \tex_lowercase:D
        {
          \group_end:
          \tl_set:Nn \l__runner_tmpa_tl { ^^@ }
        }
      \exp_args:No \__runner_forth_input_discard:N \l__runner_tmpa_tl
      \exp_args:No \__runner_forth_input_until:nn \l__runner_tmpa_tl
```

```
        { \tl_set:Nn \l__runner_tmpa_tl {#1} }
      \__runner_forth_put_here:n { \str_count:N \l__runner_tmpa_tl }
      \tl_replace_all:Nnn \l__runner_tmpa_tl { ~ } { { ~ } }
      \tl_map_inline:Nn \l__runner_tmpa_tl
        { \__runner_forth_put_here:n { `#1 } }
    }
\group_end:
```

### C.3.8 Conditional words

IF

```
\__runner_forth_new_compilation_core:nn { IF }
  {
    \__runner_forth_def_put_right_x:n
      { \__runner_forth_compiled_if:nw { \if_false: } \fi: }
    \int_incr:N \l__runner_forth_def_nesting_int
  }
```

ELSE

```
\__runner_forth_new_compilation_core:nn { ELSE }
  {
    \int_compare:nNnTF \l__runner_forth_def_nesting_int > \c_zero
      {
        \__runner_forth_def_put_right_x:n
          { \if_false: { \fi: } { \if_false: } \fi: }
      }
      { \msg_error:nnn { runner/forth } { misplaced } { ELSE } }
  }
```

THEN

```
\__runner_forth_new_compilation_core:nn { THEN }
  {
    \int_compare:nNnTF \l__runner_forth_def_nesting_int > \c_zero
      {
        \__runner_forth_def_put_right_x:n
          { \if_false: { \fi: } \__runner_forth_compiled_then: }
      }
      { \msg_error:nnn { runner/forth } { misplaced } { THEN } }
    \int_decr:N \l__runner_forth_def_nesting_int
  }
```

__runner_forth_compiled_if:nw
__runner_forth_compiled_then:

```
\cs_new_protected:Npn \__runner_forth_compiled_if:nw
    #1#2 \__runner_forth_compiled_then:
  {
    \__runner_forth_pop_int:N \l__runner_tmpa_int
    \int_compare:nNnTF \l__runner_tmpa_int = \c_zero {#2} {#1}
  }
\cs_new_protected_nopar:Npn \__runner_forth_compiled_then:
  { \msg_error:nn { runner/forth } { internal } }
```

### C.3.9 Looping words

DO

```
\__runner_forth_new_compilation_core:nn { DO }
  {
    \__runner_forth_def_put_right_x:n
      { \__runner_forth_compiled_do:n { \if_false: } \fi: }
    \int_incr:N \l__runner_forth_def_nesting_int
  }
```

LOOP

```
\__runner_forth_new_compilation_core:nn { LOOP }
  {
    \int_compare:nNnTF \l__runner_forth_def_nesting_int > \c_zero
      { \__runner_forth_def_put_right_x:n { \if_false: { \fi: } } }
      { \msg_error:nnn { runner/forth } { misplaced } { LOOP } }
    \int_decr:N \l__runner_forth_def_nesting_int
  }
```

__runner_forth_compiled_do:n

```
\cs_new_protected:Npn \__runner_forth_compiled_do:n #1
  {
    \__runner_forth_pop_int:NN \l__runner_tmpa_int \l__runner_tmpb_int
    \seq_push:Nx \l__runner_forth_return_seq
      { \int_use:N \l__runner_tmpa_int } % ^^A limit
    \seq_push:Nx \l__runner_forth_return_seq
      { \int_use:N \l__runner_tmpb_int } % ^^A index
    \int_do_until:nNnn \l__runner_tmpb_int = \l__runner_tmpa_int
      {
        #1
        \seq_pop:NN \l__runner_forth_return_seq \l__runner_tmpb_tl
        \int_set:Nn \l__runner_tmpb_int { \l__runner_tmpb_tl + 1 }
        \seq_get:NN \l__runner_forth_return_seq \l__runner_tmpa_tl
        \int_set:Nn \l__runner_tmpa_int { \l__runner_tmpa_tl }
        \seq_push:Nx \l__runner_forth_return_seq
          { \int_use:N \l__runner_tmpb_int }
      }
    \__runner_break_point:n
      {
        \seq_pop:NN \l__runner_forth_return_seq \l__runner_tmpb_tl
        \seq_pop:NN \l__runner_forth_return_seq \l__runner_tmpa_tl
      }
  }
```

I

J

```
\__runner_forth_new_compilation_core:nn { I }
  {
    \__runner_forth_def_put_right:n
      {
```

```
        \__runner_forth_push:n
          { \seq_item:Nn \l__runner_forth_return_seq { 1 } }
      }
  }
\__runner_forth_new_compilation_core:nn { J }
  {
    \__runner_forth_def_put_right_x:n
      {
        \__runner_forth_push:n
          { \seq_item:Nn \l__runner_forth_return_seq { 3 } }
      }
  }
```

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

```
\__runner_forth_new_compilation_core:nn { LEAVE }
  { \__runner_forth_def_put_right:n { \__runner_break:n { } } }
```

UNLOOP Drop the loop control parameters from the return stack.

```
\__runner_forth_new_compilation_core:nn { UNLOOP }
  {
    \seq_pop:NN \l__runner_forth_return_seq \l__runner_tmpa_tl
    \seq_pop:NN \l__runner_forth_return_seq \l__runner_tmpb_tl
  }
```

### 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.

```
\__runner_forth_new_immediate_core:nn { }
  { \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.

__runner_forth_environment_aux:n

```
\__runner_forth_new_core:nn { ENVIRONMENT? }
  {
    \__runner_forth_pop_int:NN \l__runner_tmpa_int \l__runner_tmpb_int
    \prop_get:NxNTF \g__runner_forth_environment_prop
      {
        \int_step_function:nnnN
          { \l__runner_tmpa_int }
          { 1 }
          { \l__runner_tmpa_int + \l__runner_tmpb_int - 1 }
          \__runner_forth_environment_aux:n
```

```
      }
      \l__runner_tmpa_tl
      {
        \l__runner_tmpa_tl
        \__runner_forth_push:n { -1 }
      }
      {
        \__runner_forth_push:n { 0 }
      }
  }
\cs_new:Npn \__runner_forth_environment_aux:n #1
  { \__runner_array_item:Nn \g__runner_forth_data_array {#1} , }
```

\g__runner_forth_environment_prop
__runner_forth_environment_def:nn

Keys are known string, in the form of comma lists of their character codes. Currently, 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.

```
\prop_new:N \g__runner_forth_environment_prop
\group_begin:
  \cs_set_protected:Npn \__runner_forth_environment_def:nn #1#2
    {
      \prop_gput:Nxn \g__runner_forth_environment_prop
        { \tl_map_function:nN {#1} \__runner_tmp:w }
        {#2}
    }
  \cs_set:Npn \__runner_tmp:w #1 { \int_eval:n { `#1 } , }
  \__runner_forth_environment_def:nn { ADDRESS-UNIT-BITS }
    { \__runner_forth_push:n { 24 } }
  \__runner_forth_environment_def:nn { CORE }
    { \__runner_forth_push:n { 0 } }
  \__runner_forth_environment_def:nn { CORE-EXT }
    { \__runner_forth_push:n { 0 } }
  \__runner_forth_environment_def:nn { FLOORED }
    { \__runner_forth_push:n { 0 } }
  \__runner_forth_environment_def:nn { MAX-D }
```

```
    { % ^^A todo: check endianness
      \__runner_forth_push:n { 8388607 }
      \__runner_forth_push:n { 16777215 }
    }
  \__runner_forth_environment_def:nn { MAX-N }
    { \__runner_forth_push:n { 8388607 } }
  \__runner_forth_environment_def:nn { MAX-U }
    { \__runner_forth_push:n { 16777215 } }
  \__runner_forth_environment_def:nn { MAX-UD }
    {
      \__runner_forth_push:n { 16777215 }
      \__runner_forth_push:n { 16777215 }
    }
\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.

```
\__runner_forth_new_immediate_core:nn (
  { \__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.

```
\cs_new_protected_nopar:Npn \__runner_forth_run:
  {
    \__runner_forth_init:
    \bool_set_false:N \l__runner_forth_stop_bool
    \bool_until_do:Nn \l__runner_forth_stop_bool
      {
        \__runner_forth_input_spaces:
        \__runner_forth_input_until:nN { ~ } \__runner_forth_interpret:n
      }
  }
```

`\__runner_forth_init:` Before starting the interpreter loop, we concatenate the program string and the input
`\__runner_forth_cleanup:n` 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.

```
\cs_new_protected_nopar:Npn \__runner_forth_init:
  {
    \tl_use:N \g__runner_forth_init_tl
    \int_set_eq:NN
      \l__runner_forth_toks_int
      \g__runner_forth_core_toks_int
    \int_set_eq:NN
```

```
      \l__runner_forth_data_top_int
      \g__runner_forth_data_size_int
    \prop_set_eq:NN
      \l__runner_forth_words_prop
      \g__runner_forth_core_words_prop
    \int_set:Nn \l__runner_forth_base_address_int { 1 }
    \__runner_forth_interpret:n { DECIMAL }
    \int_set:Nn \l__runner_forth_data_here_int { 2 }
    \tl_set:Nx \l__runner_tmpa_tl
      { \l__runner_program_str \c_space_tl \l__runner_input_str }
    \tl_replace_all:Nnn \l__runner_tmpa_tl { ~ } { { ~ } }
    \str_set:Nx \l__runner_forth_input_str
      {
        \tl_map_function:NN \l__runner_tmpa_tl
          \__runner_forth_cleanup:n
      }
  }
\cs_new:Npn \__runner_forth_cleanup:n #1
  { \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.

```
\cs_new_protected:Npn \__runner_forth_interpret:n #1
  {
    \prop_get:NnNTF \l__runner_forth_words_prop {#1} \l__runner_tmpa_tl
      {
        \tex_the:D \tex_toks:D
          \exp_after:wN \use_none:n \l__runner_tmpa_tl \scan_stop:
      }
      {
        \__runner_forth_get_number:nNTF {#1} \l__runner_tmpa_tl
          { \__runner_forth_push:n { \l__runner_tmpa_tl } }
          { \msg_error:nnn { runner/forth } { unknown-word } {#1} }
      }
  }
```

## C.5 Messages

```
\msg_new:nnn { runner/forth } { unknown-word }
  { The~word~'#1'~is~not~defined. }
\msg_new:nnn { runner/forth } { empty-stack }
  { The~data~stack~is~empty,~and~there~is~nothing~to~retrieve~there. }
\msg_new:nnnn { runner/forth } { out-of-memory }
  { The~Forth~interpreter~ran~out~of~memory. }
  {
    Summary~of~memory~use:\\
    \iow_indent:n
      {
        Data~usage:~
```

```
          \int_use:N \l__runner_forth_data_here_int
          \ cells\\
        Stack~size:~
          \int_eval:n
            {
              \g__runner_forth_data_size_int
              - \l__runner_forth_data_top_int
            }
          \ cells\\
        Total:~
          \int_use:N \g__runner_forth_data_size_int
          \ cells.
      }
  }
\msg_new:nnn { runner/forth } { out-of-bounds }
  { ALLOT~was~called~with~a~negative~argument~that~made~HERE~=~#1. }
\msg_new:nnn { runner/forth } { no-interpretation }
  { The~word~#1~can~only~be~used~in~definitions. }
\msg_new:nnn { runner/forth } { no-def-immediate }
  { Somehow~the~word~#1~cannot~be~found~in~the~dictionary }
\msg_new:nnn { runner/forth } { too-many-ifs }
  { More~IFs~than~THENs~in~this~definition! }
\msg_new:nnn { runner/forth } { misplaced }
  { Misplaced~#1. }
\msg_new:nnn { runner/forth } { internal }
  { Internal~error.~Please~report. }
```

</package>