How to code with morsecco

morsecco is a minimalistic, but mighty programming language using **morse c**ode for **co**ding (hence the sparkling name). You need only a dot for the short beep, a stroke (dash or slash) for the long beep and a whitespace for the break between letters. This already makes it the coolest coding feeling ever, because on most keyboards you can put your right thumb on space key, two other fingers on dot and stroke key and start coding, not moving your hand and leaving the left hand free for your beer or pizza. This already feels great, but even gets topped by you recognising that it doesn't take long until you can actually create complex and even useful software that way! If you know some morse codes, a little forth and are a type safety anarchist, you can almost start. Otherwise continue reading.

Data type and memory

There is only one data type called a cell, think of it as some kind of string. Whether this string is a UFT-8 string, one or more integer values, a memory address, the program counter, a custom type or executable code, depends on you – you are free to do the dirtiest tricks with that or to shoot your own foot.

There are two habitats for cells: The stack and the storage. The stack behaves like in all stacked-based language: You can push something on it, and most commands work on the top cell(s) of the stack. The storage is an addressable memory: `E`nter a cell with some value on the stack, then a cell with some address and `W`rite the cell with the value to the address. Later `E`nter the address and `R`ead the cell back on the stack (like using a variable in other languages). Or `U`se it as `F`ile handle to `R`ead and `W`rite. Or even use the address as a command to execute its contents (like calling a function in other languages).

The basics of coding

That's already the whole working principle, but before you start coding, just one paragraph about the morse code. In written morse code, typically a `.` stands for the short beep, `-` for the long beep and a whitespace for the break between two characters. So a character is defined by a certain sequence of dots and dashes between two spaces. For example, a single `.` is the letter `E`, the combination `.-` is `A`. Using the table to the right, you can try to decipher the sequence `-. .- -- .`! Yes, it's `NAME`. And what about `. -. . -- .-`? Maybe not your favourite word, but completely different as a programming code!

Code	letter
.	E
-	T
..	I
.-	A
-.	N
--	M

You can read more elsewhere, but this is not about the *morse code* about *coding* with *morsecco*, and here the combinations of dots and dashes[1] don't form only letters, but also

[1] for cooler coding on US or UK keyboards, the slash can be used instead, but we stick to the dash in this tutorial.

numbers or commands. That's why we call those things between two spaces a *token*, that's pretty neutral. The token `·` could be the letter E, the integer number zero or the **E**nter command to put something on the stack, depending on where they are used. If you did wonder about some strange blue word capitals, now you know it: To easily remember the commands, they correspond to the morse code of the first letter of their meaning. Thus, you already know the command `·--` for **W**rite and `·-·` for **R**ead and `··-` for **U**se.

Just like Mr Morse did use shorter codes for the most frequently used letters, *morsecco* uses the shortest tokens for the commands used most. And just like in Forth, you need to put something on the stack all the time. Command `·` **E**nters the next token on the stack, so our *enema* code first enters `-·` placing it on the top of the stack, then enters `--` above. Before we think about the A command, let's verify this! If you have morsecco installed properly, you can start it in the interactive mode by simply typing **`morsecco`** from you command line, then enter the token sequence `· -· · --` and press the return key. Nothing will happen, because you didn't request any output, you just placed two tokens on the stack. But how to verify that? In morse code, the code `···-·` stands for »verified«, so in *morsecco*, this is used for verification or debugging. Add this code and press the return key to see

```
$ morsecco
Thank you for entering morsecco 🍾 interactive mode. Type .... for Help. Leave with ctrl-D
> . -. . --
> ...-.
===
--
-.
:::
```

The blue part was what you did type, the interpreter's answer is written in black. Your verify request printed the stack (between `===` and `:::` as separators) with `--` on top and `-·` below , just like expected for a LIFO (last in, first out) stack. Great! Now let's try what happens to the stack if you use the **A** command `·-` followed by another `···-·`:

```
> .- ...-.
===
-.-
:::
```

If you know binary numbers, you can guess which command starting with an **A** replaced a `-·` and a `--` by `-·-`! Take the dots as zero and the dashes as one, thus `10` and `11` **A**dd to `101`. So now you now how numbers are coded in morsecco.[2]

```
  -·
+ --
 -·-
```

[2] Note that morse code has different codes for decimal digits, but they are quite clumsy. Minus one would take twelve keystrokes, while `·-` does the same with two. Of course, higher numbers may be tricky to enter as binary on the fly, so can also enter them as morse and **K**onvert them.

And before you ask: negative numbers are coded by simply placing a single dot in front of the number: `•-•-` is –5.[3]

Back to that *enema* sequence which turns out to be an addition: we **E**ntered 2 and **E**ntered 3, and the **A**dd command removed both numbers from the stack to replace them with their sum 5. You can also **O**utput the contents of the top cell with `---`. The whole code to put numbers 2 and 3 on the stack and print their sum is `• -• • -- •- ---`. [4]

Stack Transformations

The second 1–char–token is `-` to **T**ransform the stack, because this is also needed frequently in a stack-based language. In Forth, you do a lot of dup, swap and rot; in morsecco we have `-` as a multi-**T**ool with the token after `-` as parameter:

- if the `-` command is followed by a token of only dots, you move a cell to the top of the stack. The number of dots is the number of cells it rises. Thus, `- •` takes the second cell one step up, resulting in a `swap` of the top elements. `- ••` takes the third cell two cells up (a `rot`) to the top and so on.
- if the token after `-` is a positive integer, this is the number of the stack item to copy to the top of the stack (pick in Forth). Thus, `- -` duplicates the top cell (dup in Forth), `- -•` copies the second item (over in Forth). `- ---` will copy the 7th cell of the stack to the top – this could be a hint that your coding style is weird … or you are doing a brilliant hack.
- a negative token after the `-` gives the (negative) number of a stack item to remove. `- •-` removes to top cell (`drop`), `- •-•`

Code	Stack before	Stack after	Forth
- •	-- - • -	- • -- -	swap
- •	-- - • -	- -- - •	rot
- -	- • -	-• -• -	dup
- -•	-• -	- -• -	over
- •-	-- -• -	-• -	drop
- •-	-- -• -	-- -	nip

[3] The behaviour of a number with more than one leading dot is undefined and may change in future versions, so don't rely on it.

[4] A side note on formatting: It will not take long until you can easily write such a piece of code, but you can probably imagine that it can soon get impossible to read, because you don't even see what is a command and what is a parameter or whatever kind of token. So it can be a good idea to insert additional spaces to make logical units visible: `• -•  • --  •-  ---`.
In more complex code, we can also use a line break instead of a space (both have the identical meaning of splitting tokens). And you can indent some lines with spaces or tab (again identical function) to make structures in your code visible.
The additional spaces usually don't hurt: a double space produces an empty token in between, but an empty token as a command is ignored, so it doesn't matter. Just be careful: Don't separate a command from its parameter token with a double space, as this would make the empty token the parameter (which often has a special meaning). And sometimes the code may be treated specially until the next empty token, so keep that in mind.
Finally, you can also comment your source code: `• -•  • --  •-  --- add 2 and 3` No problem, everything but dots, dashes and the three types of whitespaces are ignored. Of course, this will destroy the cool feeling of coding with just one hand. And never be tempted to use a period or dash in your comment, because they will always be interpreted!

removes the second one (`nip`) and so on.

- an empty token after the `–` (that is two whitespaces, because the spaces separate token, and between two spaces there is just an empty token) tells the interpreter to take the parameter from the top cell on the stack (like the pick command in Forth does).[5]

Jumping and looping

Not very productive until now! We want to do our work, no stack juggling, but as with every stack-based language, you need to know your basic tools. Thank you for your patience! We know how to add, so why not calculate the *sum formula*: for a number n you add 1+2+3+4 and so on, until you reach n. For n=5, the sum formula gives 15.

In other languages, you have for-loops or while-loops for such a job, like `do {sum=sum+n; n=n-1;} while (n>0)`. In morsecco, we could do a recursion, but here we do it building our own loop.

The value *n* is **E**ntered on the stack, then we enter a zero cell, where we will sum up while counting down *n*. Next, we open the loop by setting a **M**ark with `––  –`. The parameter `–` gives the relative number of the token to mark, in this case 1, which is the mark command itself. So this program position is now saved on our address stack and we can later **G**o back there and resume execution. For a loop, it's typical to mark the Mark command itself, because the **G**o command will remove the address, so we need to mark it again each time.

Inside the loop we add value *n* to our *sum* by copying it to the top of the stack with `–  –•` (because it is the second cell), then **A**dd. Now to decrement *n*, we first need to swap *n* and *sum* with `–  •` to make *n* the top cell. There is no minus command, but we can simply subtract by **ad**ding a negative value, so we **E**nter –1 and **A**dd: `•  •–  •–` . Before we close the loop with the **G**o command, we need to swap again to have the *sum* as top cell, like it was at the beginning of the loop.

Of course, our loop lacks an exit by now. We have to test whether *n* becomes zero, so we can stop adding. Let's do this after decrementing *n*, when it is still the top cell. Currently, morsecco has no if statement, although the `••` is reserved for that, but I didn't need it so far. So here we use the **Z**ero-skip, which means if the top cell contains a zero, we skip all further tokens until the one given as parameter. As parameter we choose the **G**o command `––•`, so in case of a skip, execution continues after that command. **Z**ero-skip is polite, not

[5] This does not only add flexibility like misusing the stack as an array, but also allows to do highly complex stack transformations in one step, because a cell can contain a string with spaces, leading to multiple tokens which will be executed one after another. If you want to test this case, you will want to know how to Enter multiple tokens into one cell. There are many ways, the easiest is using the `•` command followed by an empty token: Instead of putting an empty cell on the stack, you enter a cell with all tokens until the next empty token. Thus, if you want `•  –` (ET) on the stack, do `•■■•■–■■`

touching the top cell unless it is zero, so we neither need to duplicate it before not to remove the zero after the skip. We can finally **O**utput the result or first do `-·- -·` to **K**onvert it to a **N**umber to keep you from interpreting binary numbers. Here is our code, including the explanation and some equivalent code in a boring language (sometimes the order of commands if different there):

```
· -·-  Enter 5     n=5
· ·    Enter 0       sum=0
-- -   Mark here       do {
  - -· copy n
  ·-   Add               sum=sum+n
  - ·  swap
  · ·- Enter minus1
  ·-   Add                 dec(n)
  --·· --· Zeroskip            while (n!=0)
  - ·  swap
  --·  Go                    }
-·- -· Konvert Num                        %i
---    Output                     printf("  ",n)
```

And yes, this – with all that comments – is valid code! All other stuff than dot and dash is ignored. Just don't write something like `b=b-1` in your comment, because the `-` will seen as command, causing random errors!

As an environmental advice, you should clean up your stacks after a command. The data stack is clean here, but we left waste on the address stack: In the last looping we **M**ark the beginning of the loop, but we don't execute the **G**o to remove it, so we manually drop it with `-- ·`:

```
-- ·     clean Addr stack
```

The **M**ark parameter is like the ages: There was a year 1 B.C. and a year 1 A.C., but no year zero. There is also no **M**ark zero, leaving the parameter `·` for dropping the top of the address stack.[6]

Creating own commands

In a complex project, you don't want to always write all that stuff each time you need the sum formula. To have clean, understandable code, you want your sum formula be used as easy as the sum command, taking *n* from the stack and leaving the sum on it. The part of our code doing that is:

```
· ·      Enter sum
-- -     Mark here
  - -· ·-   Add n to sum
  - ·       swap
  · ·- ·-   decrement n
  --·· --· Zeroskip
  - ·       swap
  --·       Go
-- ·     clean addr stack
```

[6] You can also use `-- ··` to drop the second address or … well, you guess it.

Input of *n* and output are left away, because they are not part of the command to create. The cleaning of the address stack in included as it becomes really important here. And the code is reformatted by grouping commands that belong together in one line for better readability. If you are more used to it and get lazy, you will maybe leave away the comments and replace the line breaks by comma (the loop with double comma), so you help readers understand the code without needing the left hand for coding or moving your right hand:
`· ·,, –– –, – –· ·–, – ·, · ·– ·–, ––·· ––·, – ·, ––·,, –– ·`
That is the form we need to let magic begin. First, we have to get this code in a cell, but we learned that **E**nter takes only the next token as parameter. A token is everything between two whitespaces, so a token can't contain whitespaces. Luckily, there is a trick: If you give an empty token as a parameter by giving two spaces after the `·`, you **E**nter everything up to the next empty token (the next double whitespace). That's why we need the code without double whitespaces to enter it in one step. Here you see the code above placed between two double whitespaces that are separated by a thin red line for better visibility:

```
$ morsecco
Thank you for entering morsecco 🍾 interactive mode. Type .... for Help. Leave with ctrl-D
> .  . .,, -- -, - -. .-, - ., . .- .-, --.. --., - ., --.,, -- .  ...-.
===
. . -- - - -. .- - . . .- .- --.. --. - . --. -- .
:::
```

After the long parameter there is the `····–·` command to verify the output and indeed, the whole sequence is placed on the stack, just losing the commas, because other characters are really ignored. Now continue like this:

```
> . .....-. .-- ...-.
===
:::
.....-. : . . -- - - -. .- - . . .- .- --.. --. - . --. -- .
```

As the name of the new command, `·····–·` was chosen (that's S `···` and F `··–·` for Sum-Formula glued together), but you could choose another name, just avoid build-in commands! This name was **E**ntered and the **W**rite command `·––` was called. The verification call now shows an empty stack between `===` and `:::`. The new line after `:::` shows the storage with the format ***`name : content`***. So our code has been written to the storage cell named `·····–·`. We could **R**ead it back to the stack with command `·–·`, but we can also

```
> . --. .....-. -.- -. ---
21
```

This did **E**nter 6 (`––·`) on the stack and called the new storage cell as a command. The result on the stack is **K**onverted to **N**umber and put **O**ut as 21, the sum formula for 6. We executed our first self-defined command, taking the first step into a wide open of possibilities.

Of course, self-defined commands can also have parameters like some build-in commands have. On calling a command the current program position is automatically **M**arked on the address stack, and **R**eading from the special address –– (**M**ark) reads the next token from the position pointed to by the top of the address stack to the stack and advances the pointer. This way you can read one or more parameters following your own command. After execution the program will continue after the read token(s).

And this opens even more possibilities, if you learn that ·– (**A**) is also a special address for the **A**ddress stack. **R**eading from ·– moves the top cell form the address stack to the stack; Writing to it does the opposite. And the format of an address is simply a (non-negative) number for the position in the cell, a space and the address of the cell, so –·· –···–· points to the 4th character in the cell at –···–. So let's say you have some tokens to be parsed in a cell at address – –, then place a pointer to the start of them on the address stack with ·   · – –   · ·– ·–– and retrieve one token after another with · –– ·–·!

Recursion

If you are used to stack-based languages, you may already have thought: if we define a command anyhow, then why loop at all? You can probably skip this chapter. Who is less used to this mind set may learn to love the elegance of recursions here. Simply compare the code of our command with the loop:

```
· ·      Enter sum
-- -     Mark here
  - -· ·-  Add n to sum
  - ·      swap
  · ·- ·-  decrement n
  --·· --· Zeroskip
  - ·      swap
  --·      Go
-- ·     clean addr stack
```

with the following using recursion:

```
- -        dup n for later adding
. .- .-    decrement n
--.. .-    Zeroskip to simply leave zero on the stack
.....-. .- otherwise call itself and Add
```

That's already it! No initialisation of sum=0, no loop, no swapping, no stack cleaning. This is the literal translation of $SF_n=n+SF_{n-1}$. If you are not used to this, add a stacktrace ····–· before the **Z**eroskip and at the end of the command: You can see how the stack grows with each call of the command placing the decremented number to the stack, until zero is reached and each command call leaving the sum so far.

Files

This brilliant piece of code needs to be save for the future! So let's learn how to write and read files. First, we've got to tell a storage cell (here we take ··–· for **F**), we want to **U**se it as **F**ile handle and assign it a file name. As file name we can take our command's name •••••–• with •–– as file extension[7], if your file system prefers to have such a dot-separated extension (the –– after the dot is meant to be an **M** for morsecco … what a poor joke!):

```
• ••–•                Enter F as file handle
••– ••–• •••••–••–– Use as File with given name
```

Nothing has happened so far. The file system has not been checked, no file has been created, we just internally connected the address ••–• with the file name. But as soon as we will now Write to that address, it will not get written to the storage, but to the named file:

```
• •••••–• •–•   Read our command
• ••–• •––      Write it to the file
• ••–• ••– –•–• Close the file
```

You can check with a text editor that there is really a file with that name (note that it may be hidden because of the leading dot) containing the command. To make sure we are really reading from the file, we use a different file handle (and you can even chance the file's contents with the text editor) for **R**eading:

```
• ••–•• ••– ••–• •••••–••–– • ••–•• •–• –––
```

This should have read and printed the file contents.[8] Instead of printing it you could have written it to an address to execute it. This way you can import commands: Read them from a file and Write them to their command names. Or you do that at startup:

```
$ morsecco -r .....-..-- '. .....-. .--' -i
> . -- .....-. ---
--.
```

The **-r** option of morsecco reads the file given as argument and places its contents on the stack. The next part in ticks is executed as morsecco code.[9] You know this code: it **W**rites the cell placed by the **-r** option to the given address. The final option **-i** enters interactive mode, similar to executing morsecco without any arguments (just without the welcome

[7] If you prefer a boring readable file name, you can **E**nter it between empty tokens as morse code, **K**onvert it from •**M**orse, **K**onvert it to **T**ext and follow the **U**se **F**ile with an empty token to take the file name from the stack:
•■■•••■••–•■•–•–•–■–••••–■–••••–■■–•–■•––■–•–■–■•■••–•■••–■••–•■■

[8] You may have noticed, that we didn't »open« the file for reading or writing, like you do in many languages. As soon as you Use a File, it can be read or written, so be careful not to accidentally write to file you didn't want to! If a file does exist, Writing to it does append the cell's content to the end by default. If you want to overwrite it, you first need to Move to the Start of the file with • ••• • ••–• ••– –– (place Start and the file handle on the stack, then Use Move). Other Moves can be ••–• for the Finish of the file, a positive number for the number of characters to Move forward or a negative number to move backward.
Reading starts from the start of the file by default. Subsequent Reads continue at the current file position. A following Write will append to this file position, deleting the rest of the file.

[9] Everything which is no option or option's argument is seen as code – you could even leave away the quotes in this case, but note you cannot enter empty tokens then, because the calling shell will eat the extra spaces before morsecco can see it.

message). Without `-i`, after all scripts have been executed, morsecco would silently quit (you use that to execute scripts without user interaction). In interactive mode, `· ––` Enters a 3 on the stack, then you execute the command just stored and get the output `––·` (6, the sum formula of 3).

By default, the whole file is Read, but you can change that the Usage to reading Linewise, Token by token or a given number of CHaracters or Bytes. For example, if your file starts with hex bytes `00 09 0A`, the code

```
· ··–· ··– ··–· –·– Use address F as File dashdotdash
· ··–· ··– –···     Use F to read Bytes
· –– · ··–· ·–· ––– Read and Output 3 bytes
```

will give the output `· –··– –·–·`

A final remark about file names: In a file name, each combination of dash+space will be replaced by your file system's directory separator, while dot+space will get replaced by a single whitespace. Thus, the code

```
·  ··– ––·––– · –· · ·––  · ··–· ··– ··–·  · ––– · ··–· ·––
```

will use the file path `../--.--/ -   .--` to create a file with a weird name starting with a whitespace and later having a double space in a subdirectory of the parent directory.

More Usage magic

Another example of the Use command:

```
$ echo 'https://github.com/' | morsecco '. - .-.  . --. .--  . --. ..- ..-  . --. .-.  ---'
```

will read the address from sTdin, Write it to `––·`, change it's Usage to Url, so the next Read will not read the storage, but try to read from the given Url!

Let's introduce two more Use modes for commands: `··– –··–` sets the Use mode to eXecutable. This may seem superfluous, since you can execute a command without that, but what if you dislike the behaviour of a build-in command? Just writing code to the register will still execute the original build-in command … unless you set the Usage to eXecutable! This way you can even overwrite the Add command. The address `·–` has a special meaning for accessing the address stack, but after you change its Usage to eXecutable with `· ·– ··– –··–`, you can Write code to the address and the next `·–` will execute this code.

The other command Usage introduces some kind of name spaces. It's so simple to create new commands in morsecco, that it is a good programming style to define a new command for any command sequence with more than a couple of tokens. Unfortunately, this will make you soon run out of short commands in a complex project. If you build up a library of command which include other commands, each occupying some command or storage names, how to make sure nothing is used twice? That's the reason for Use Local: If your command has this Usage mode, it creates its own independent storage with all addresses unused.[10]

Creating another Usage

Morsecco would not be morsecco if it would leave you alone with the given set of Usages! Creating a Usage of your own is as simple as defining a command to handle it and give the address of that command as a parameter to the Use command. Following Reads or Writes or Use commands like Close or Move on the address you connected to your own Usage will then place the calling command on the stack and hand over to your code.

Let's illustrate this by creating the Usage `···-` for a Vector of tokens: We want it to take another value from the stack for the index. In the first step, a Read with index 1 on the stack shall put the first token on the stack, index *n* shall return the *n*-th token.

When our handler is called, it will find three cells on the stack: the index to pick from the vector, the address of the vector and a token for which command called the handler (a read or a write or something else). The idea for the code is to create a pointer to the start of the address containing the vector move that address to the address stack (special address `·-`) to retrieve tokens from there doing reads from special address `--`. Looping over the index reading tokens until the right one is found leads to this handler code:

```
--·· --·-            Zeroskip for empty cell = read access
  - ·- - ·- --·-       for other access drop access and address and Quit to return
· · - ·  -·-· ··     Concatenate a pointer to position 0 of the given address
-- -                 Mark to start a loop
  · ·- ·--  · -- ·--·  move the pointer to address stack and a read token from there
  - · · ·- ·-          decrement the index
  --·· --·             Zeroskip on the index
    - ·--·               drop the unused token
    · ·- ·-· --·         move the pointer back from the address stack and Go loop
-- · -- ·            drop the pointer and the loop Mark form the address stack
```

If we write this code to address `···-`, and write the word `WORD` in address `----`:

```
·  ·-- --- ·-· -··  · ---- ·--
```

we can set the mode of `----` to `···-`

```
· ---- ··- ···-
```

and Read index 3 from our `WORD` to get the `R`:

```
· -- · ---- ·-· ---
```

This should output `·-·` (= `R`). Morsecco doesn't offer an array as data type, but this mode handler expanded the language to retrieve an array element. And the same way one could

10 Even the address under which the command itself was stored if free to use, thus a library command doesn't even need to know under which address it has been executed. No file handles or overwritten commands will bother you, only the special addresses for address stack, Morse table and stdin/stdout are shared globally (and the stacks themselves or course). Even the Error handler you'll later learn about can be set locally.

add `W`rite support or for example, `U`sage `D` to delete an item from the array (the `U`sage-build-ins will not overwrite the mode, but be passed to your mode handler).

It's your turn

This was about all to explore the unlimited possibilities by yourself. Use the command table of this tutorial or the build-in Help command of the interactive mode to find out the rest. If you are into *code golfing*, the next chapter shows some typical challenges and how to solve them. Please report bugs, improvements for the tutorial, suggestions of any kind or contributions via GitHub.

Now have fun … have a lot of fun!

Code golfing with morsecco

Code golfing is a programming challenge with the goal to write a program with a given function in as few bytes of source code as possible. Special languages for code golfing use the whole range of 256 different codes per byte for highly complex commands. Obviously, morsecco can't compete with those in terms of code length, because it is the opposite: The commands are extremely basic and each byte can only have three meanings. You could easily shorten the code size by a factor of 5 by compressing five characters with their 2^5=243 combinations into one byte, but it still could not compete with golfing languages.

But still morsecco is a great language for golfing, probably one of the most interesting, because there are so many different approaches and tricks possible. In most languages the optimisation is mainly done by a clever algorithm, while it's more or less obvious how to translate that algorithm into short code. But morsecco gives you a variety to do the same thing with different code: You can write flat code or define own commands, do classical loops or recurse, use variables or do everything on the stack, do tricks with multi-token cells, manipulate the address stack, work with self-modifying code and so on.

Quine

A *quine* is a program giving its own source as output, a basic code golfing task. To produce a quine with morsecco, you just need to know that the code of the main program is also in the storage with the empty address (not shown by `••••-•`, because it would mess up the output most of the time). So you just need to produce an empty cell and `R`ead from this empty address. An empty parameter causes `•` to `E`nter everything up to the next empty token, so if you let another empty token follow by having three whitespaces after the dot, you get your empty cell on the stack. Finally you have to print it `Out` to get your quine:

`• | | •-• ---` (11 bytes, red lines are just to indicate the spaces.)

Sometimes accessing the source code is considered cheating, so let's have a look at a way to do a quine without reading from the empty address: `E`nter a command, write it to an address and execute it. If this command writes its own code along with the code before and after it, it's a quine (94 bytes):

```
.  . . . -. .-. -.-. ... . . -.-. ... . -. -.-. .. . .-- -.-. .. . -. -.-. .. ---  . -. .-- -.
```

The **C**oncatening of the four commands costs many bytes. How to golf this further? By adding the trailing code to the command, **C**ut it and **C**oncatenate it twice (88 bytes):

```
.  . . . -. .-. -.-. ... -.-. .--.. - - -.-. .. -.-. . --- --.- . -. .-- -.  . -. .-- -.
```

So that's it? Not yet. Instead of **W**riting a command to an address and executing it as a command, we can directly e**X**ecute it from the stack (73 bytes):

```
.  . . - . -.-. ... . - -.-. ... . - -.-. .. . -..- -.-. .. ---  - - -..-
```

Hello, world!

A program that outputs the string »Hello, world!« is often to first task a tutorial starts with. Not here, because it is less obvious what has to be done when letters, comma and exclamation mark are ignored in the source. There are many ways to do that, the (probably) shortest of them seen here with 90 bytes:

```
.  .... ---- . .-.. .-.. --- --..-- -..... .-- --- .-. .-.. -.. -.-.--  -.- .--- -.- - ---
E  H    CH   e l    l    o   ,      space  w   o   r   l    d   !       Kov frM Kov T Out
```

The string is **E**ntered as morse code between the double spaces, then **K**onverted from **M**orse (`-·-  ·---`), **K**onverted to **T**ext (`-·-  -`) and printed **O**ut. Morse code in not case sensitive, you have no way to produce lower case letters, so morsecco misuses `----`: This is defined as the letter combination `CH`, but morsecco takes this as the **C**ase **H**ack to toggle between uppercase and lowercase. Other symbols not available are coded by their UTF-8 value, in this case the whitespace written as `-·····`. Considering the factor 5 lost by using only three symbols, this is like 18 bytes in other language, so not too bad for a 13-byte message.

Reverse the input

Special file `-` is the standard input and output, so can can **R**ead from there and loop, always cutting the first char, appending it to the result:

```
·       Enter empty cell as result string
· - ·-· Read from stdin
-- -    Mark here
- - ·-··  Length of input
--·· --·  leave if length is zero
- ·-      drop length
-·-· -    Cut first char
- ··      rot result to top
-·-· ·    Concatenate result and char
- · --·   swap and Go to loop
- ·- ---drop empty input and Output result
```

Or, as the golfed version with 75 bytes:

```
·   · - ·-· -- - - - ·-·· --·· --· - ·- -·-· - - ·· -·-· · - · --· - ·- ---
```

Output a single whitespace

This does sound like a boring task, but in a language using the whitespace a separator for tokens it's not trivial to place a whitespace on the stack, because **E**nter takes a token, but a whitespace can't be a token. **E**nter with an empty token can take spaces between tokens, but neither at the beginning nor at the end, so the shortest possibility would be to **E**nter the whitespace between two short tokens and remove the tokens with two **C**ut commands:
`·  · ·  −·−· − − · −·−· − −−−` (29 bytes)
Really bad! It's shorter to create two empty cells and **C**oncatenate them with a whitespace:
`·   ·   −·−· ·· −−−` (19 bytes)
Can't we avoid those 8 bytes for creating two empty cells? Creating one and duplicating it
`·   − − −·−· ·· −−−` (19 bytes)
does not help, but it illustrates how morsecco offers different approaches to a task. Even stranger solutions like creating an empty error handler by writing an empty cell to special address `·`. That allows me to call **C**oncatenate with an empty stack without producing an error. The empty stack will then return empty cells, which can be **C**oncatenated with a whitespace. Unfortunately, creating the empty error handler takes 12 bytes:
`·   · · ·−− −·−· ·· −−−` (23 bytes)
You'd need to use the **Q**uiet mode with option `-q` to suppress error messages (which does also set an empty error handler):
`morsecco -q '−·−· ·· −−−'` (11 bytes)
Let's try another weird trick with **M**arking the code position: This will place position zero, a white space and an empty token in a cell on the address stack. If we **R**ead it from there and **C**ut away the leading dot, we get a whitespace, too:
`−− − · ·− ·−· −·−· − − · −−−` (28 bytes)
Cutting and swapping were too expensive! The shortest solution (besides from option `-q`) is probably to **E**nter the unicode 32 of whitespace and **K**onvert to **T**ext:
`· −····· −·− − −−−` (18 bytes)

Command table

The following table explains the commands and their variations. The stack effect is shown in the form: (stack before → stack after) with stack elements shown from the bottom (left to the top of the stack (right).

Code	Command	Subtoken & stack effect	
·	Enter	· ▢*token* (→ token) · ▢▢t_1▢…▢t_n▢▢ (→ ‹t_1▢…▢t_n›)	Put the next token on the stack Put everything between the empty tokens on the stack

Code	Command	Subtoken & stack effect	
−	Transform stack	`−▪+i` (c_n … c_1 → c_n … c_1 c_i) `−▪−i` (c_n … c_1 → c_n … c_{i+1} c_{i+1} … c_n) `−▪`*`dots`* `−▪▪` (*list* →)	copy the i-th stack cell to the top drop the i-th stack cell move the cell given by number of dots up execute stack commands from *list*
·−	Add	(x y → x+y) (‹x_1… x_n› ‹y_1 … y_n y_{n+1} … y_m› → ‹x_1+y_1 … x_n+y_n y_{n+1} … y_m›)	add single values add multiple values pairwise; trailing items of the longer cell remain untouched

−−	Mark	`−−▪−`
		`−−▪`*`n`*
		`−−▪·`*`n`* (negative number)
		`−−▪▪`
		
		`−−▪·`
		`−−▪`*`dots`*
			Mark current code position on address stack
			Mark position of *n*-th token (`−−` itself is 1)
			Mark position *n* tokens before the `−−`
			Move the position on the top of the address stack to the search pattern in top cell
			drop top item from address stack
			drop *dots*-th item from address stack

−−·	Go	addr:(‹*pos*▪*cell*› →)	continue execution at *pos* in storage *cell* taken from the top of the address stack.
−··−	eXecute	(‹*code* …› →) (·− →) (−··− →)	eXecute *code* eXecute Again the code of the last −··− eXecute current code recursively
−−··	Zero-skip	`−−··▪`*`token`* (*n* → *n*) `−−··▪`*`token`* (· →)	do nothing for non-empty non-zero top cell continue execution after next occurrence of *token*
−·−	Konvert	`−·−▪−` (−·····− → A) `−·−▪·−` (A → −·····−) `−·−▪−·` (−·····− → 65) `−·−▪·−·` (65 → −·····−) `−·−▪−−` (−·····− → ·−) `−·−▪·−−` (·− → −·····−)	Konvert to Text Konvert from Text Konvert to Number Konvert from Number Konvert to Morse code Konvert from Morse code
··−	Use	`··−▪··−·▪`*`name`* (*handle* →) `··−▪··−·▪▪` (*name handle* →) `··−▪−·−·` (*handle* →) `··−▪−··` (*address* →) `··−▪−−` (*n handle* →) `··−▪−−` (··· *handle* →) `··−▪−−` (··−· *handle* →) `··−▪·` (*handle* →) `··−▪·−··` (*handle* →) `··−▪−` (*handle* →) `··−▪··−` (*handle* →) `··−▪−···` (*handle* →)	Use *handle* as File with given *name* with empty name take name from stack Close file Delete file or cell at *address* Move file position by n (negative = back) Move file position to Start Move file position to Finish Switch the Read mode: to read Everything = whole file (default) to read Linewise to read Token by token to read given number of Unicode chars to read given number of Bytes
·−·	Read	(*address* → ‹cell(*address*)›) (*handle* → ‹file contents›) (*handle* → ‹next line of file›) (*handle* → ‹next token of file›) (*i handle* → ‹*i* unicode chars›) (*i handle* → ‹*i* Bytes as numbers›) (− → ‹read from stdin›) (as defined by custom command) (·− → ‹pos›) addr:(‹pos› →) (−− → ‹*token*›) addr: (a → ‹a+x›)	Normal storage cell file in · mode file in ·−·· mode file in − mode file in ··− mode file in −··· mode handle = − : stdin in given mode execute the *mode* cell with empty top cell handle = .− : from Address stack read one token from Marked position and advance the address pointer

Code	Command	Subtoken & stack effect	
•––	Write	(*cell address* →) (*cell* •– →) addr:(→ ‹*cell*›) (‹*code x y z*› –– →) (*cell handle* →) (as defined by custom command)	Write the *cell* to the given storage *address* push *cell* to address stack update Morse table to konvert *code* to *x y z* Write the contents of *cell* to file *handle* execute the *mode* cell with top cell •––
•–••	Length	(`cell` → `length`)	replaces a cell by the Length of it's content
–•••	Binary op	–•••▪•– (x y → x&y) –•••▪––– (x y → x|y) –•••▪–••– (x y → x^y) –•••▪–•• (x y → diff) –•••▪–•• (x x → '')	Binary And Binary Or Binary Xor Binary Diff gives • for each match, – for each diff, empty for equal
–•–•	Concatenate Cut	–•–•▪• (c1 c2 → <c1c2>) –•–•▪•• (c1 c2 → <c1 c2>) –•–•▪*dots* (c1 c2 → <c1 c2>) –•–•▪*n* (c → c1 c2)	Concatenate cells Concatenate cells with a whitespace use one space less than number of dots Cut cell after the n-th character for n<0 the n-th character from the end
–––	Output	––– (text →)	write cell's contents with newline to stdout
––•–	Quit		Jump to the address on top of the address stack (if any) or quit the script.
	Help	••••▪▪ ••••▪*topic*	general help with empty token help on *topic*
••••–•	Debug output		
••	I (reserved)		
–•	N (reserved)		
•••	S (reserved)		

Special addresses

	(empty)	The empty address contains the main code. Be very careful when writing to it!
•	Error	if this cell exists, its code will get executed on error instead of aborting
–	sTdout	Write cell contents to stdout
–	sTdin	Read from stdin and place the read content in a cell on the stack
•–	Address stack	move a cell between stack and address stack
––	Morse table	Write a code and some unicodes separated by a space to change the morse table
––	Marker token	Read a token from the position marked by the top cell of the address stack