MINISHELL

by bhagenlo

written for version 6

```
Wohoo, you've arrived at minishell. Well done!
Tokens, the environment and builtins await. Come in, come in :)
```

You'll be creating a working and day-to-day usable shell. It will be able to execute arbitrary commands, have its own environment, write to/read from files, and so on.

> Disliking minishell?
>
> I don't think the issue's the project, but rather how we handle it. Read on here, if you're interested.

For all of the following, there's one thing to keep in mind:

```
The bigger the project, the more it pays dividends to
know understand what you want before you actually
start to write code. Minishell is such a big project.
```

That being said, let's go.

PREREQUISITES

Let us start by doing that: What exactly we're trying to do here?

We are trying to implement our own programming language.

That's a big task, and can make one a little bit anxious, but it's also a big opportunity for us. People have been doing that since the early 50s. There's a big body of knowledge to learn from. Condensing that:

You probably want to structure your minishell like that:

Input `-lexer->` tokens
tokens `-parser->` AST
AST `-evaluator->` evaluated AST
evaluated AST `-executor->` output

1. Find out what a *lexer* is. What should it do in the case of minishell? What would be the resulting tokens?
2. Find out what a *parser* is. What could/should it do?
3. What is an AST (*abstract syntax tree*)?
   What could it look like in our case? Draft programs (+ their AST) on paper.
   Can you think of an easier way than a tree? If so, make a draft of that, too.

▾ DURING

Apart from sticking to the proposed general structure, another thing that buys you a lot of time is to *write tests from the beginning*.

Say, you start out with writing your lexer (for that, it might be helpful to stick to `bash`'s tokenization rules). During that whole time, already think of test cases where your lexer might break. Collect them, and transfer them into tests.

Same for your parser (You can find the bash grammar here).
As for the parser's output, in case you're not doing the bonus, the way to go might be to use a *Command Table* as shown here.

For both of those: Beware of the Heredoc. (Look HERE.)

If those both work, you're all set to implement the inner workings of a shell. I'll leave it at that.

Except for those regularly occurring questions/issues:

- Yes, it makes sense to *have your own environment (functions)*. How do you get it? -> `extern char **environ`.
- Before going on, research what every allowed function in the subject description does.

CLEANING UP

Oof. I'm not really sure what's to clean up here. I hope you tested your shell thoroughly (probably even used it for a bit), and you found it to be generally usable.

It does make sense to ask a peer that did minishell already for a test evaluation.

And, since you don't know who'll evaluate you, probably also go through an Eval sheet once or twice :)

AFTERCARE

Well, after minishell, you should know about:

- Writing a lexer
- Writing a parser

- Creating pipes, and handling IO in general
- How hard (-> *time-consuming*) it is to write reliable, good-working big(ger) systems in C.

In case you don't feel like you do, try to think it through again. I consider parsers and lexers to be generally very useful tools, and `microshell` awaits :)

Concerning the last one, in case you're not convinced:
It's quite probable a large part of all your errors were bugs related to the manual allocation of memory. And it's not only you.
Fortunately, it's 2023, and at least since May 15, 2015, there is only very little reason to still introduce them to our codebases. The two key terms here are Garbage Collection and Ownership.

So, no further work writing a parser by hand for you today. Instead, you might want to take a look on how to do it differently. Have fun and enjoy exploring!

- Q: I like C. How would I write a parser in a C codebase?
  A: As far as I know, you'd use `yacc`.
- Q: I want to have speed and complete control. Which language should I pick?
  A: How about Rust?
- Q: I want to have the most advanced language building technology. Where should I look?
  A: For that, take a look at the self-proclaimed *Programming Language* Programming Language: Racket.

```
Huh, what a project. Nasty sometimes, but full of learning. See
you at the next one.
```

```
And may your grammars always be consistent.
```

POINTERS

- bash's tokenization rules
- The bash grammar.
- For the Command Table.
- Heredocs
- Harm Smits' guide.

EVALUATING MINISHELL

> I've yet to meet someone (who finished the project) and hasn't been complaining about the project/all the annoying edge cases they needed to "fix".
>
> (Sure, people like to complain, I know that, too. With minishell, it's at a whole other level.)
>
> It goes like the following:
> They all start out implementing their own shell, and then end up implementing every peculiarity of `bash`.
> *Taking something as a reference* **is not the same as** *copying its behaviour to the tiniest detail*.
>
> I don't think that this is the point of the subject, but more importantly, it *should* not be. We're here to learn, remember? You can learn a lot from this subject. Heck, you're writing your own little programming language! But somehow, the joy this brings gets lost in all the edge case-fixing I've encountered. (Or, at least,

> it only rarely gets mentioned by the people who've completed minishell.)
>
> `Want to do something about this? Read on.`

I think this project can be different, and it's up to us to make this happen. It can be more about understanding, and less about tedious replication. How can we achieve that?

I think it all comes down to how minishell gets evaluated.

Right now, the eval sheet is a practical joke. While the others also don't ask for looking at the code, they at least leave time for actually doing that. Minishell's doesn't. But how to make it better?

I propose the following (/something along those lines):
(exchange the respective paragraphs from the subject with interrogations in a similar fashion.)

- `echo` should print the text it's given to the screen.
  Specifically: Check whether both `echo test` and `echo -n test lala` work the intended way
- `unset` should unset variables from the environment.
  Ask the evaluatees how they did do it. What does their environment look like? How do they mutate it?
  Then, check something like that:

  ```
  $ export H=42
  $ env
  ...
  H=42
  $ unset H
  ```

```
$ env
(no H anymore)
```

```
$ unset PATH
$ ls # or any other command in PATH
ls: command not found
```

The point here is: In contrast to the normal eval sheet, the evaluator checks for *understanding*, and then tests whether the code is working *in general*.

They *do not* try to break the shell's (or its builtins') parser at all costs, checking for every test case they know. Neither do they check only for the specific test cases of the eval sheet, which one can easily guard against.

`I'd love to see you join me on this.`

`See you around. :)`