Designing a Programmable Stream Editor

Spinellis, Diomidis

DOI
10.1109/MS.2025.3597697

Publication date
2025

Document Version
Final published version

Published in
IEEE Software

Citation (APA)
Spinellis, D. (2025). Designing a Programmable Stream Editor. *IEEE Software*, *42*(6), 23-27. https://doi.org/10.1109/MS.2025.3597697

Editor: Diomidis Spinellis
dds@aueb.gr

Designing a Programmable Stream Editor

Diomidis Spinellis, Athens University of Economics and Business and Delft University of Technology

THE UNIX *SED* stream editor is a programmable text processing filter, first written in C in the 1970s.[1] In the previous installment of this column, I described the tool and how I reimplemented it in Rust with some help from generative AI.[2] Here, I describe the new implementation's design to show how we can create a simple programmable tool. In the next "Adventures in Code" column, I'll present optimizations that substantially increased the tool's throughput.

What's behind a programmable tool, such as Python, *sed*, or SQLite? It turns out that the key component is the data structures used to represent the code. As a student, I came across the equation "Algorithms + Data Structures = Programs."[3] I can still remember that at the time I knew what algorithms were, but data structures were to me a fuzzy concept of academic only interest. "Why would anyone need something more than the numeric and string arrays supported by the BASIC language?" I thought. Over the years, I've come to appreciate the value and significance of how we organize our data. Now, I believe that data structures are far more important than algorithms. First, in modern systems, bespoke sophisticated algorithms play only a minor, if any, role; data structures rule! Second, in many cases the algorithm's choice is a clear-cut decision, whereas deciding on the data structure to use requires a deep understanding of context and a careful weighing of tradeoffs. Third, type systems have matured to offer us powerful practical aid in dealing with data; corresponding formal models for algorithms less so.

Consequently, the design of many systems should often start by considering how data will be structured: a database's schema, key classes and their fields, types and operations on them.

Script Compilation

For a *sed* script, the central representation is each individual command. This is defined in file command.rs (see Figure 1) as follows. (I've removed the comments to save space. You can find the fully commented code on the command's repository: https://github.com/uutils/sed.)

```
pub struct Command {
    pub code: char,
    pub addr1: Option<Address>,
    pub addr2: Option<Address>,
    pub non_select: bool,
    pub data: CommandData,
    pub next: Option<Rc<RefCell<Command>>>,
    pub location: ScriptLocation,
}
```

Each command is defined through the single character code that represents it (e.g., "b" or "t" for a branch or conditional command), the input's addresses to which it applies (e.g., from line 5, until a line containing END — "5,/END/"), whether the line application is negated, command-specific data (e.g., the branch target), the command to execute next (as a reference counted—Rc—mutable reference—RefCell— to it), and the location of the command in the supplied script in order to report runtime errors.

The command-specific data can be of various types, depending on the command, and are therefore represented as a variant (sum) type through a Rust enum.

```
pub enum CommandData {
    None,
    BranchTarget(Option<Rc<RefCell<Command>>>),
    Label(Option<String>),
    Path(PathBuf),
    NamedWriter(Rc<RefCell<NamedWriter>>),
    Number(usize),
    Substitution(Box<Substitution>),
    Text(Rc<str>),
    Transliteration(Box<Transliteration>),
}
```

Many commands can share the same syntax and type of additional data. For example, the append ("a"), insert ("i"), and change ("c") commands

Digital Object Identifier 10.1109/MS.2025.3597697
Date of current version: 10 October 2025

store their corresponding string as a text element. On the other hand, the substitution ("s") command has its own particular syntax (s/*search-expression*/*replacement-text*/*options*) and is uniquely associated with the following data element, which stores the regular expression to match, the text to replace it with, and the options affecting the replacement.

```
pub struct Substitution {
    pub regex: Option<Regex>,
    pub replacement: ReplacementTemplate,
    pub occurrence: usize,
    pub print_flag: bool,
    pub ignore_case: bool,
    pub write_file: Option<Rc<RefCell<Named
        Writer>>>,
}
```

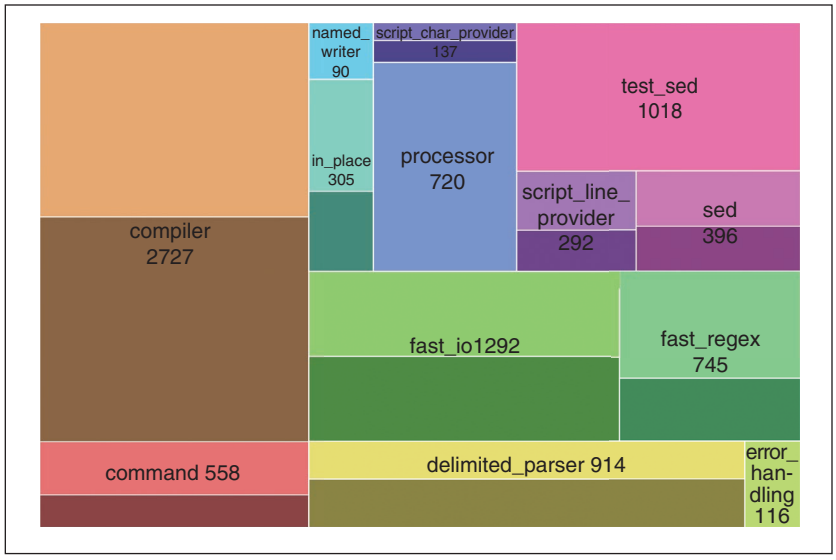

FIGURE 1. Script compilation. Files comprising *sed* and their size in lines of code. Dark portions represent unit tests. The files'.rs suffix is omitted.

When *sed* is run, it processes the script passed to it, converting each individual command into a corresponding command struct and linking it to the next one. This process converts the script into an abstract syntax tree (AST). At the top level of *sed's* script compiler (file compiler.rs), this is done by 13 functions that correspond to the syntax variations of *sed*'s 24 commands. Commands also differ in the number of addresses that specify the input to which they apply. For example, and ":" (branch label) command does not take an address prefix, the append ("a") and insert ("i") commands apply to a single address, whereas the delete ("d") and change ("c") commands can apply to a range of lines specified through two addresses. By defining all 13 functions with the same arguments (as a CommandHandler type), a Rust match statement can return each command's specification (CommandSpec) containing the number of addresses and the handler function associated with each command.

```
type CommandHandler = fn(
    lines: &mut ScriptLineProvider,
    line: &mut ScriptCharProvider,
    cmd: &mut Command,
    context: &mut ProcessingContext,
) -> UResult<CommandHandling>;

struct CommandSpec {
    n_addr: usize,
    handler: CommandHandler,
}
```

The following is an excerpt of the function performing the corresponding matches. It takes as an argument the command's character code (cmd_code) and then returns the expected number of addresses and the function that handles its syntax.

```
fn get_cmd_spec(
    lines: &ScriptLineProvider,
    line: &ScriptCharProvider,
    cmd_code: char,

) -> UResult<CommandSpec> {
    match cmd_code {
        ':' => Ok(CommandSpec {
            n_addr: 0,
            handler: compile_label_command,
        }),
        'a' | 'i' => Ok(CommandSpec {
            n_addr: 1,
            handler: compile_text_command,
        }),
        'c' => Ok(CommandSpec {
            n_addr: 2,
            handler: compile_text_command,
        }),
        'd' | 'D' | 'g' | 'G' | [...] => Ok(CommandSpec {
            n_addr: 2,
            handler: compile_empty_command,
        }),
        [...]
        _ => compilation_error(lines, line, format!
            ("invalid command code '{cmd_code}'")),
```

Thus, the get_cmd_spec function multiplexes the compilation requirements of the 24 *sed* commands into 13 functions and three address options.

For the parsed structure to become executable, some commands

require further processing. Specifically, the branch targets, which are initially stored as a string representing the label, must be converted to the target's address, while the opening part of command blocks ("{ "and "}"), which enclose conditionally executed command groups, must point to the block's end. These changes are performed by three functions traversing the AST: populate_label_map, resolve_branch_targets, and patch_block_endings. Traversing the whole AST isn't trivial because commands are recursively defined: a command can contain CommandData, which can be a BranchTarget pointing to a Command. To handle this complexity, as is common with recursively defined data types, the processing is also implemented recursively: the three functions processing the commands recursively invoke themselves when they encounter an inner element of the same type as the one passed to them.

In interpreter implementations, to speed up the subsequent execution, as much work as possible is done when constructing the AST. For example, the addresses to which a command applies are stored as a tuple containing the type of each address (lines matching a regular expression, an absolute line number, a relative line number, or the last line), and an enum containing the integer line number or the precompiled regular expression.

```
pub struct Address {
    pub atype: AddressType,
    pub value: AddressValue,
}

pub enum AddressValue {
    LineNumber(usize),
    Regex(Option<Regex>),
}

pub enum AddressType {
    Re,
    Line,
    RelLine,
    Last,
}
```

This design amortizes, over the possibly millions of times an address will be interpreted, the cost of converting the line number string into an integer or, more importantly, compiling the regular expression into the internal finite automaton representation.

The neat division of responsibilities among the command-handling functions hides several details. To keep the compilation code's size manageable, several low-level support functions reside in separate files. Functions in the file delimited_parser.rs are mainly responsible for handling delimited string sequences, such as the elements of the substitute and transliterate commands, address regular expressions, character classes (e.g., [A-Z0-9]), and C-style character escapes (e.g., "\n" or "\0x1e").

Then, there are two files that support reading the command script. One (script_line_provider.rs) provides methods to iterate over the lines of the specified script (the script can be given through command-line arguments, files, or the standard input.) The other, (script_char_provider.rs), provides methods for reading each line, character by character. Both offer methods for obtaining the precise location of each script's character in order to facilitate accurate error reporting.

Script Execution

Once *sed* converts the script into an AST, processing its input data as specified by the AST (file processor.rs) is relatively easy. A big factor contributing to the complexity of the AST generation is the script's verification. Checking for user errors must be comprehensive, and the reporting must be clear and understandable. This adds complexity, which is reflected in the relative size of the code associated with the two phases visible in Figure 1.

The core implementation of the script's execution is a loop over its AST, applying the appropriate commands to each input line (the so-called pattern space). The following are representative code examples for the processing of some commands. The design mirrors the design I adopted for the version of *sed* I originally wrote in C in the 1980s, demonstrating the value of data structures that have withstood the test of time.

```
while let Some(command_rc) = current.clone() {
    let mut command = command_rc.borrow_mut();
    if! applies(&mut command, &mut pattern,
        context)? {
        //Advance to next command
        current = command.next.clone();
        continue;
    }

    match command.code {
        '{' => {
            //Block begin; start processing the enclosed
                ones.
            let body = extract_variant!(command,
                BranchTarget);
            current = body.clone();
            continue;
        }
        'd' => {
            //Delete the pattern space and start the
                next cycle.
            pattern.clear();
            break;
        }
        'p' => {
```

```
        //Write the pattern space to standard
            output.
        write_chunk(output, context, &pattern)?;
    }
```

Unsurprisingly, this implementation style is named an *AST interpreter*. Other methods are possible. A *bytecode interpreter* converts each high-level command into a low-level byte representation, stores these bytes sequentially, and executes the script by successively processing the byte sequence elements. A *threaded interpreter* employs a cleverer method. Each instruction is represented as a pointer to a routine that performs its function. The interpreter reads each pointer and then calls the associated routine. I did not choose the alternative representations because the corresponding code would be less readable, and I reasoned that due to Rust's heavy optimizations of the AST interpreter code, any additional performance benefits would be minimal.

The open sourcing of the original AT&T code written by L. E. McMahon in the 1970s and the parallel development of GNU *sed* allowed me to compare my design against others. Interestingly, both compile commands into sequential bytecode rather than an AST. Within the bytecode stream both employ an indirection to other *sed* instructions only to implement branches. For this, McMahon's implementation uses C pointers, while the GNU one uses an integer index. Cleverly, both implement the execution control of brace-delimited command blocks with branch commands, something that the Rust implementation does with bespoke handling code.

Here again, several processing details are tucked away into separate files: the in-place editing of an input file (in_place.rs) and the output to files specified in the script (named_writer.rs). Regular expression handling and input/output are also handled in separate files (fast_regex.rs, fast_io.rs). As we shall see in the column's next installment, this was done to optimize *sed*'s performance rather than hide complexity.

Error Handling

Detailed, accurate, and consistent error reporting can help *sed*'s users to correct problems in their scripts. *Sed*'s design facilitates this as follows. The two structs whose methods provide the script's lines and the lines' characters, also have methods to pinpoint accurately the input's location. Error-reporting functions for compilation and runtime errors (implemented in the file error_handling.rs) use this location information to construct UResult<T> error enums with a detailed and consistent message in the standard convention:

file-name:line-number:column-number: error-message.

The UResult-constructing functions also incorporate in the created UResult the Unix exit code associated with the error class (compilation or runtime error). When an error is also associated with input data, for example a UTF-8 conversion error raised by an attempt to perform case-insensitive string matching, then the error is expanded to also include the input file's coordinates. (UTF-8 is the commonly used 8-bit Unicode transformation format. Not all byte sequences represent valid Unicode characters, so conversion errors can occur.)

All functions where an error can occur return a UResult type, which wraps Rust's result type. This forces their callers to propagate the error by employing the same return type through the application of Rust's "?" operator, thereby ensuring that all errors are consistently reported and handled.

Testing

As one can see in Figure 1, about half of the code in each source code file is devoted to unit tests. The execution engine (processor.rs) does not have unit tests associated with it because it can be more productively tested with end-to-end integration tests. In these, the command and execution context can be easily specified through *sed*'s command-line arguments (these are processed in file sed.rs).

Integration testing (file test_sed.rs) is based on file fixtures that specify input (in most cases numbered lines) and expected output. Most tests are specified through a macro named check_output!, which takes as arguments the test's name and the arguments with which to invoke *sed*. The test name also doubles as the file name of the expected output. After *sed's* execution, it compares the output to the expected one. The following is an example of the macro's invocation, which tests that the specified print command ("4p") will output the input file's (LINES1) fourth file.

```
check_output!(addr_one_line, ["-n", "-e", "4p",
LINES1]);
```

I derived many of the test cases from my original BSD Unix *sed* implementation by copying the expected output verbatim into a test fixture and by rewriting the *sed* execution specified through a shell function in the original test's script into a Rust macro invocation.

ABOUT THE AUTHOR

DIOMIDIS SPINELLIS is a professor in the Department of Management Science and Technology at the Athens University of Economics and Business, 104 34 Athens, Greece, and a Professor of Software Analytics in the Department of Software Technology at the Delft University of Technology, Delft, 2600 AA, The Netherlands. Contact him at dds@aueb.gr.

In a nutshell, a programmable tool's design involves parsing the code into an appropriate data structure with paranoid attention to its detailed verification and then interpreting the data structure as efficiently as possible. A generous mix of unit and integration testing can help ensure the tool's correctness.

References

1. L. E. McMahon, "SED—A non-interactive text editor," in *UNIX Programmer's Manual (Supplementary Documents)*, vol. 2, 7th ed. Murray Hill, NJ, USA: Bell Telephone Lab., 1979.
2. D. Spinellis, "Rewriting the Unix stream editor in Rust," *IEEE Softw.*, vol. 42, no. 5, pp. 21–25, Sep./Oct. 2025, doi: 10.1109/MS.2025.3579008.
3. N. Wirth, *Algorithms + Data Structures = Programs*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1976.