Concepts of Program Design — Project Proposal

Lars Folkersma (5543800) — Paul van Grol (5528909)
Mark Leenen (6294464) — Matej Milinkovic (6324401)
Ivo Muusse (6632602) — Tom Smeding (6588956)

December 2018

Contents

1 Background

We chose to base our language on the language for *sed* scripts. We make a few slight modifications and leave a few things out (this will be explained in more detail in the "Methodology" section). *sed* scripts define a text transformation, and thus text transformation will be the domain of our language. *sed* only makes one pass over the input and is often used in a pipeline on UNIX systems.

Our motivation to use this language is a combination of a few factors. Firstly we think that it should be feasible to implement this language in Haskell in the form of a function which takes a *sed* script, and returns a function that does the same transformation that the *sed* program does, except for the few modifications we make and things we leave out. *sed* scripts are just a list of various rather simple commands, so it shouldn't be that hard to implement. On the other hand, *sed* is Turing complete, and allows for all kinds of complex transformations, so it's not too limited either. *sed* also uses regular expression matching, which we can take as a primitive in our formalization, but can also define in detail. This gives us some flexibility: if it turns out that our choice of language was too simple then we can expand on it in this way.

We intend to use Haskell as our host language because of its power in expressing algebraic properties of data structures and powerful abstractions. This will prove useful for creating an implementation that stays structurally close to the formalisation.

2 Methodology

Since our goal is to model the existing *sed* programming language, we use the POSIX 2017 *sed* standard [1] as the guiding document for our formalisation. In this section, we will give a short description of the core *sed* language features that we will formalise and implement, and the language features that we aim to cover only if time allows.

2.1 Core Language Features

Programming Model *sed* is a line-oriented streaming text editor. The runtime environment of the scripting language is the *sed* utility, which reads text from the standard input stream and writes text to the standard output stream according to the given script. The memory available to the script is organised in two strings, called the *pattern space* and the *hold space*. Most commands deal only with the pattern space, and commands that touch the hold space are only concerned with managing the pattern and hold spaces.

Execution of a *sed* script is organised in *cycles*. A cycle is started at program startup, and at the end of each cycle a new one is started if more input lines remain (and the `q` command has not been issued). At the start of a cycle, a line is read from the input (if the cycle was not started due to the `D` command being executed on a pattern space containing a newline; see below) and stored in the pattern space, after which the script starts executing. If execution reaches the end of the script, the pattern space is written to the output if the `-n` flag has not been specified to the *sed* utility. The contents of the hold space are kept for the next cycle (the pattern space will be overwritten).

The script consists of *commands* to be executed. These commands can read new lines from input, write extra lines to output, manage the pattern and hold spaces, perform transformations on the pattern space, and manage control flow. The first four of these five command sets are illustrated in Fig. 1; all sets are described in more detail below.

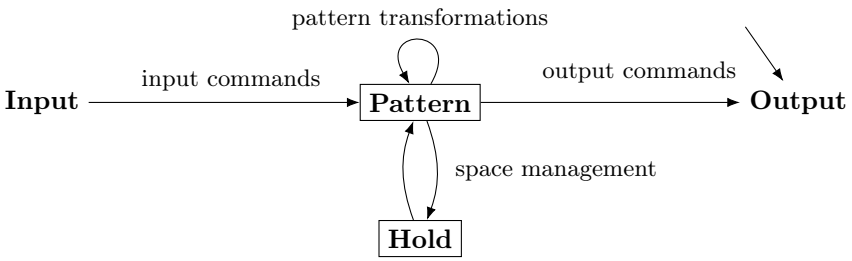

Figure 1: Dataflow commands in a *sed* script; note that control flow commands are not depicted here.

Most commands can optionally have an *address specifier* added to them, which determines whether a command is enabled in any specific situation. The address specifier can place restrictions on the current *line number*, i.e. the number of lines read from the input up until now, and the current pattern space. If the address specifier of a command is satisfied when execution reaches it, the command is executed; otherwise, it is skipped.

An address specifier consists of either one or two simple addresses; a simple address is either a line number (enabling the command only on that line), a dollar sign (enabling the command only when the last line has been read), or a regular expression (enabling the command only when the expression matches the current pattern space). An address consisting of two simple addresses indicates a range: if the first simple address matches when execution reaches this command, the command is executed, and each subsequent time it will execute again until the second simple address matches. Thereafter, the command will be executed again only if the first simple address matches again, etc. If both simple addresses are a line number or a dollar sign and the second is less than the first, the command will be executed once, when the first line number matches.

If a command has an address specifier, the specifier can be followed by a `!` to indicate that the address needs to be *inverted*: whenever the address would match according to the above description, the command is skipped; otherwise the command is executed.

Core Commands The *sed* commands can generally be divided in five categories as follows:

- (**I**) The input commands read a new line of input immediately and put it in, or append it to, the pattern space. Execution of the script continues as usual, without starting a new cycle.
- (**P**) Pattern transformation commands apply certain transformations to the pattern space.
- (**H**) Space management commands copy or append the pattern or hold space to the other space, or exchange these two spaces. These are the only commands that touch the hold space at all.
- (**C**) Control flow commands regulate the execution flow of the script, by (conditionally) jumping to different parts of the script.
- (**O**) Output commands write extra text to the output, either immediately or after the next line is printed. This text can either be specified literally in the script or come from the pattern space, depending on the command.

The core command set is listed in Table 1, where commands are divided in one or more of the above categories in the last five columns. Note that descriptions may be fully or partially taken from the POSIX *sed* specification [1]. Not all commands can take address ranges, and some commands can take no address specifiers at all. If a command supports a full address specifier, it is marked with "[*2addr*]"; if it only supports a simple address, it is marked with "[*1addr*]"; and if no addresses are supported, it is marked with "[*0addr*]".

A number of commands included in the POSIX *sed* specification are omitted in Table 1, because we considered them uninteresting compared to the other commands (e.g. `l`, for writing the pattern space to

How does the static sem. of your language compare to sed? You should have some added value - e.g. more checks, stronger static assurances

the output in a "visually unambiguous form") or because they would make the formalisation unwieldy without a significant benefit (e.g. the file reading and writing commands).

Command	Description	I	P	H	C	O
[*2addr*] `{` *cmds* ... `}`	Executes a list of *sed* commands. This allows enabling a list of commands with a single address specifier.				X	
[*1addr*] `a` *text*	Schedules text so that it will be written to standard output just before the next attempt to fetch a line of input when executing the `N` or `n` commands, or when reaching the end of the script.					X
[*2addr*] `b` [*label*]	Branch to the specified label; if not specified, branch to end of script.				X	
[*2addr*] `c` *text*	Delete the pattern space. With a 0 or 1 address or at the end of a 2-address range, place text on the output and start the next cycle.		X		X	X
[*2addr*] `d`	Delete the pattern space and start the next cycle.		X		X	
[*2addr*] `D`	If the pattern space contains no newline, delete the pattern space and start a normal new cycle as if the `d` command was issued. Otherwise, delete the initial segment of the pattern space through the first newline, and start the next cycle with the resultant pattern space and without reading any new input.		X		X	
[*2addr*] `g`	Copy the hold space over the pattern space.			X		
[*2addr*] `G`	Append a newline and the contents of the hold space to the pattern space.			X		
[*2addr*] `h`	Copy the pattern space over the hold space.			X		
[*2addr*] `H`	Append a newline and the contents of the pattern space to the hold space.			X		
[*1addr*] `i` *text*	Write text to output stream.					X
[*2addr*] `n`	Write the pattern space to standard output if the default output has not been suppressed, and replace the pattern space with the next line of the input without its terminating newline. If no next line of input is available, branch to the end of the script and quit without starting a new cycle.	X			X	
[*2addr*] `N`	Append a newline and the next line of input (without its terminating newline) to the pattern space. If no next line of input is available, branch to the end of the script and quit without starting a new cycle or copying the pattern space to standard output.	X			X	
[*2addr*] `p`	Write the pattern space to standard output.					X
[*2addr*] `P`	Write the pattern space, up to the first newline, to standard output.					X
[*1addr*] `q`	Branch to the end of the script and quit without starting a new cycle.				X	

[*2addr*] `s` *BRE repl flags*	Substitute the replacement string for instances of the BRE in the pattern space. If *flags* includes `g`, replace all instances instead of just the first one; if *flags* includes a number n, replace only the n'th instance (invalid together with `g`); if *flags* includes `p`, write the pattern space to the output if a replacement was made.		X			X
[*2addr*] `t` [*label*]	Branch to the label (or end of script if unspecified) if any substitutions have been made since the last input line was read or the last execution of a `t`.				X	
[*2addr*] `x`	Exchange the pattern and hold spaces.			X		
[*2addr*] `y` *str1 str2*	Replace all occurrences of characters in *str1* with the corresponding characters in *str2*.		X			
[*0addr*] `:` *label*	Do nothing. This command bears a label to which the `b` and `t` commands branch.				X	
[*1addr*] `=`	Write the line number of the last read line to the output.					X

Table 1: List of core commands. Some descriptions fully or partiall taken from the POSIX *sed* specification [1].

2.2 Regular expressions

Arguably the most powerful feature in the *sed* language is the ability to match regular expressions to strings and perform replacements based on those matches. However, the functionality of regular expressions is largely separate from the rest of the language, and initially we will focus on the core language as described above, considering regular expression matching and substitution as black-box operations. If that is successful, we will expand the formalisation and implementation to also include regular expressions as white-box behaviour, making our coverage almost complete.

3 Planning

The planning we have created for this project remains rather simple, since we do not yet know all the intricacies of it. We have set internal deadlines about two days ahead of the deadlines for the intermediate and final report, which gives us time to proofread the result and make changes as necessary. Therefore we have internal deadlines at December 20th and January 23rd.

Going further in the project, we have decided to, for now, split the team up in two groups to work on the formalisation and implementation of the language. The group to work on the implementation will consist of Lars, Paul and Tom, since they are already familiar with Haskell. Ivo, Mark and Matej will work on the formalisation of the language. This division is subject to change as we notice that either the formalisation or implementation requires more work.

With the short period between the proposal and intermediate report, we have decided not to set further internal deadlines. Depending on how the project unfolds, we might add additional internal deadlines beyond those set before the final deadline, for example to ensure that there is enough formalisation to work on the implementation of the language.

References

[1] The Open Group. *POSIX.1-2017 (IEEE Std 1003.1-2017): sed – stream editor*. 2018. URL: `https://pubs.opengroup.org/onlinepubs/9699919799/utilities/sed.html` (visited on 12/07/2018).