Specification

An *alphabet* is a set; a *letter* is an element of the alphabet. A *language* is a set of strings comprised of letters. A string is allowed to be empty (i.e., it has length zero); the empty string is commonly denoted by $\emptyset$.

The generality of the definitions above makes them widely applicable. For example, the command-line inputs to a C program as communicated by `argv` can be interpreted as an element of some language in the following way:

- The array `argv` is regarded as a string.
- The letters comprising `argv` are `argv[1] ... argv[argc-1]`.
- The alphabet is a set of strings (containing the `argv[i]` above).

While perhaps unexpected, the abstract definitions allow the alphabet to contain strings and allow the language to consist of strings which are themselves comprised of strings! Moreover, nothing prevents the interpretation of `argv` as a string whose first letter is `argv[1]` (rather than `argv[0]`), and therefore `argv` could be the empty string.

We can apply the language concepts recursively to obtain a more detailed description for the alphabet above. This will be illustrated using the source code `top-down.c`. An informal specification for the intended behavior is:

> If `argv` $= \emptyset$, then the code echos `stdin` to `stdout` except that non-ascii characters are omitted. Command-line flags are `-i`, `-o`, `-l`, and `-r`; each — except the last — should be followed by a file name; `-i <input file>`, `-o <output file>`, `-l <log file>`. The following data will be written to the log file (if a log file is specified):
>
> input file name
> output file name
> date
> line number: character-position non-ascii (in hex format)
> ⋮
>
> When present, the `-r` flag indicates that non-ascii characters are to be replaced (rather than omitted) with the character whose description follows the `-r` flag; three formats are allowed. The first is a character (to be used as the replacement). The second is the ascii code (of the character to be used as the replacement) as a decimal in one of the forms $d_0.$, $d_1d_0.$, or $d_2d_1d_0.$ where the d_i are decimal digits and the integer provided is in the range $[0, 255]$. The third is the ascii code (of the character to be used as the replacement) as a hexadecimal in one of the forms $0xd_0$, $0Xd_0$, $0xd_0d_1$, $0Xd_0d_1$ where the d_i are hexadecimal digits and the integer provided is in the range $[0, 255]$.
>
> Deviation from the command line arguments as specified above is an error and should cause a usage message to be printed to stderr and a return value of 1 (otherwise 0 is returned).

Before proceeding to obtain a description of the alphabet for `argv`, a high-level specification is provided for the language $\mathcal{L}$ from which `argv` should come. Consider first the command line options.

$$\begin{aligned} \mathit{In} &:= \{-i\}\ \mathit{file\_name} \\ \mathit{Out} &:= \{-o\}\ \mathit{file\_name} \\ \mathit{Log} &:= \{-l\}\ \mathit{file\_name} \\ \mathit{Replacement} &:= \{-r\}\ \mathit{replacement\_character} \end{aligned}$$

The left-hand sides above are names for the languages on the (corresponding) right-hand sides; the languages (denoted by) *file_name* and *replacement_character* have yet to be specified. The juxtaposition of languages — as in {-i} *file_name* — indicates language product, defined abstractly by $AB = \{\alpha\beta\ :\ \alpha \in A, \beta \in B\}$, where $\alpha\beta$ denotes the concatenation of α with β.
The alphabet $\mathcal{A}$ for $\mathcal{L}$ is

$$\mathcal{A} := \{-i, -o, -l, -r\} \cup \mathit{file\_name} \cup \mathit{replacement\_character}$$

The language for `argv` is $\mathcal{L} = \mathcal{L}_0 \cup \mathcal{L}_1 \cup \mathcal{L}_2 \cup \mathcal{L}_3 \cup \mathcal{L}_4$ where

$$\begin{aligned} \mathcal{L}_0 &:= \{\emptyset\} \\ \mathcal{L}_1 &:= \mathit{In} \cup \mathit{Out} \cup \mathit{Log} \cup \mathit{Replacement} \\ \mathcal{L}_2 &:= \mathit{In\ Out} \cup \mathit{Out\ In} \cup \mathit{In\ Log} \cup \mathit{Log\ In} \cup \mathit{In\ Replacement} \cup \mathit{Replacement\ In} \cup \\ &\quad \mathit{Out\ Log} \cup \mathit{Log\ Out} \cup \mathit{Out\ Replacement} \cup \mathit{Replacement\ Out} \cup \mathit{Log\ Replacement} \cup \\ &\quad \mathit{Replacement\ Log} \\ &\vdots \end{aligned}$$

More precisely, let $\mathcal{O} = \{\mathit{In}, \mathit{Out}, \mathit{Log}, \mathit{Replacement}\}$, and let π_n denote the set of all injective functions of type $f\ :\ \{1, \ldots, n\} \to \{1, \ldots, n\}$. If $n > 0$, then

$$\mathcal{L}_n = \bigcup_{\{L_1, \ldots, L_n\} \subset \mathcal{O}} \ \bigcup_{f \in \pi_n} L_{f(1)} \cdots L_{f(n)}$$

Language $\mathcal{L}_n$ corresponds to (any distinct) n command line options from $\mathcal{O}$ being provided (in any order).

Although elements of *file_name* are letters in $\mathcal{A}$ (the alphabet for `argv`), they are strings in a language with alphabet A (letters which comprise filenames). In GNU-Linux, file names can contain any byte with the exception of the forward slash and the null byte. Identifying bytes with their values as unsigned characters (in the definition of A below),

$$\begin{aligned} A &:= \{1, \ldots, 255\} \setminus \{47\} \\ \mathit{file\_name} &:= A^+ \end{aligned}$$

where A^+ denotes *positive closure* (all positive length strings consisting of letters from A). The fact that file names have a maximum length need not impact the specification; the environment (operating system, shell, ...) will enforce some maximum length.

The language *replacement_character* is a union of three languages (one for each allowed format), and its alphabet can be taken to be the set C of all characters (bytes are identified with their values as unsigned characters in the definition of C below).

$$\begin{aligned} C &:= \{0, \ldots, 255\} \\ \textit{replacement\_character} &:= C \cup \textit{decimal}\,\{.\} \cup \{0\}\ \{\text{x, X}\}\ \textit{hex} \end{aligned}$$

Languages:

character:

$$C := \{0, \ldots, 255\}$$

decimal:

$$\begin{aligned} d_0 &:= \{0,1,2,3,4,5,6,7,8,9\} \\ d_1 &:= \{0,1,2,3,4,5,6,7,8,9\} \\ d_2 &:= \{0,1\} \cup \{2 \mid d_1 \leq 5 \wedge d_0 \leq 5\} \\ \textit{decimal} &:= \{d_0,\ d_1d_0,\ d_2d_1d_0\} \end{aligned}$$

hex: *(below, D and C represent the characters and not elements of a language)*

$$\begin{aligned} h_0 &:= \{0,1,2,3,4,5,6,7,8,9\} \cup \{\text{a, A, b, B, c, C, d, D, e, E, f, F}\} \\ h_1 &:= \{0,1,2,3,4,5,6,7,8,9\} \cup \{\text{a, A, b, B, c, C, d, D, e, E, f, F}\} \\ \textit{hex} &:= \{h_0,\ h_1h_0\} \end{aligned}$$

replacement_character:

$$\textit{replacement\_character} := C \cup \textit{decimal}\,\{.\} \cup \{0\}\ \{\text{x, X}\}\ \textit{hex}$$

The language $\mathcal{L}$ may be criticized as only appropriate for the non-error case; if `argv` $\notin \mathcal{L}$ then a usage message should result (that is essentially a vacuous complaint, because $\mathcal{L}^c$ is determined by $\mathcal{L}$).

The *Trace Assertion Method* is a means by which the behavior of a program can be specified in terms of the inputs it accepts (a time-ordered sequence of events is a trace). Ideas often considered part of the method include the systematic consideration of every possible input trace, and the grouping of traces into equivalence classes based upon indistinguishable corresponding output behavior (present and future). Applied to the current context, that involves defining an appropriate equivalence relation on $\mathcal{L} \cup \mathcal{L}^c$ and associating with each equivalence class a corresponding behavior. The result should form a complete and consistent specification for the programs behavior (corresponding to every input trace is a uniquely specified behavior).

Program behavior is complicated by input/output; operations may fail due to nonexistent, insufficient, or busy resources, there may be inappropriate permissions, etc. The typical approach to deal with such intricacies is to condition program behavior on the outcome (return value) of the system calls used for i/o. Such issues will be postponed (consider first the case where system calls succeed).

The most obvious equivalence class is $\mathcal{L}^c$, and the corresponding behavior is to print the following message to `stderr` and return 1.

```
Usage: ascii-only [-r replacement] [-l log_file ] [-i input_file] [-o output_file]

  Missing input/output files default to stdin/stdout.
    Replacement is one of the following:
    character
    hex constant ( i.e., 0x3F )
    integer from 0 to 255 with decimal point ( i.e., 132. )
```

Additional equivalence classes follow from considering which $\mathcal{L}_n$ contains `argv` (this is a natural consequence of how the $\mathcal{L}_n$ are defined). In every such case the return value is 0.

If $n = 0$, then `argv` $= \emptyset$ and the corresponding behavior is to echo `stdin` to `stdout`, except that non-ascii characters are omitted.

Case n = 1.
Subcase `argv` $\in In$. The corresponding behavior is to echo the contents of file `argv[2]` to `stdout`, except that non-ascii characters are omitted. If `fopen(`*In*`)` fails, `perror("infile")` is called, printing to `stderr`, and the program returns unsuccessfully.

Subcase `argv` $\in Out$. The corresponding behavior is to echo `stdin` to file `argv[2]`, except that non-ascii characters are omitted. If `fopen(`*Out*`)` fails, `perror("outfile")` is called, printing to `stderr`, and the program returns unsuccessfully.

Subcase `argv` $\in Log$. The corresponding behavior is to echo `stdin` to `stdout`, except that non-ascii characters are omitted. Moreover, the following data will be written to `argv[2]`:

`(null)`
`(null)`
date
line-number: character-position non-ascii (in hex format)
⋮

where line-number and character-position (counting from zero) identify the position of each non-ascii character encountered (character-position counts from the beginning of a line). If `fopen(`*Log*`)` fails, `perror("logfile")` is called, printing to `stderr`, and the program returns unsuccessfully.

Subcase `argv` $\in$ *Replacement*. The corresponding behavior is to echo `stdin` to `stdout`, except that non-ascii characters are replaced by the character designated by `argv[2]`. If `argv`[2] $\in$ `C`, the character is `argv[2][0]`. If `argv`[2] $\in$ *decimal*{.}, the character is `(unsigned char) strtol(argv[2], NULL, 10)`. Otherwise, the replacement character is `(unsigned char) strtol(argv[2]+2, NULL, 16)`.

Case $n \geq 2$. There are 12 subcases for $n = 2$, corresponding to the 12 sets whose union defines $\mathcal{L}_2$. In the absence of helpful notation to streamline the consideration of those cases, the specification is tedious at best, and there is more drudgery ahead for $n = 3$ and $n = 4$.
However, these cases can be handled based on the observation that

$$\texttt{argv} \in \mathcal{L}_n \implies \exists\{\mathcal{L}_1, \ldots, \mathcal{L}_n\} \subset \mathcal{O} \,.\, \exists f \in \pi_n \,.\, \texttt{argv} \in L_{f(1)} \ldots L_{f(n)}$$

If $In \in \{L_1, \ldots, L_n\}$, *input* is the file `argv[k+1]` where `argv`[k] $\in \{-$`i`$\}$; otherwise *input* is `stdin`.
If $Out \in \{L_1, \ldots, L_n\}$, *output* is the file `argv[j+1]` where `argv`[j] $\in \{-$`o`$\}$; otherwise *output* is `stdout`.
If $Log \in \{L_1, \ldots, L_n\}$, *logfile* is the file `argv[i+1]` where `argv`[i] $\in \{-$`l`$\}$; otherwise *logfile* is undefined.
If *Replacement* $\in \{L_1, \ldots, L_n\}$, *character* is as determined by the last subcase for Case n = 1 above after `argv += h-1` where `argv`[h] $\in \{-$`r`$\}$; otherwise *character* is undefined.

The corresponding behavior is to echo *input* to *output* except that non-ascii characters are:
Omitted if *character* is undefined and replaced by *character* otherwise. Moreover, if *logfile* is defined, then the data described by the third subcase for Case $n = 1$ above is written to *logfile* except that the first occurrence of `(null)` is replaced with the *input* file name if input is not `stdin`, and the second occurrence of `(null)` is replaced with the *output* file name if *output* is not `stdout`.