

# fountain-parser README

version 0.1.0.0

## Synopsis

`fountain-parser` is a small parser library for the [FOUNTAIN](#) screenplay format, fully supporting 1.1 version [syntax](#) and producing a simple, easy to grok AST.

`fountain-parser` is written in [HASKELL](#) and it uses the [MEGAPARSEC](#) library for parsing.

## Disclaimer

Currently, this is *pre-alpha* software, not yet usable in productive form.

This software is distributed *as-is* under the terms of the BSD THREE-CLAUSE LICENSE. See the [LICENSE](#) file for more details.

## Motivation

The [Developers section](#) of the Fountain site provides a link to a [parsing library](#) in OBJECTIVE C. This presents a portability issue: while there *are* projects that make it possible to bridge OBJECTIVE C and HASKELL, they're platform- or framework-specific. That library informs this project in matching the different Fountain entities even as it uses different parsing methods.

## Prospective Related Projects

`fountain-parser` aims to power a series of command-line utilities for conversion from FOUNTAIN to a series of convenient formats (like `.tex`) without intervention from thirds.

## My software already supports Fountain

The [Apps section](#) of the FOUNTAIN site lists software that also imports or exports the format. There's a caveat: most are *cloud-based* and/or *proprietary*. By favoring (mostly) open formats, *fountain-parse* allows integration into many FLOSS tools, enabling entirely non-proprietary workflows and helping the creation of compound documents such as production bibles.

## Implementation Specifics

In general, the library parser is rather lenient, allowing liberal spacing and recognizing UNICODE codepoints. Languages without uppercase/lowercase distinction must resort to *power-user characters* for case-dependent items such as transitions ('>') and character names ('@').

- As per the [syntax guide](#):
  - This library expects FOUNTAIN text to be encoded in UTF-8.

- Tabs are converted into **four** spaces.
- Your line spacing is respected.
- Initial spaces are ignored everywhere except in action lines.
- A line with two spaces doesn't count as an empty line.
- All parsing functions expect Text inputs. File I/O is left to the application or framework.
- Varying-width UNICODE spaces are either converted into regular spaces or suppressed if they're hairline- or zero-width.
- Vertical tabs and form-feed characters are interpreted as line changes. For vertical spacing, use multiple blank lines and/or the FOUNTAIN form feed character sequence ("===") instead.
- The parser keeps everything: notes, boneyards, sections and synopses. Some possible conversion targets have analogues to those, so it might be desirable to preserve them.

## Tentative Grammar

The following is an attempt to formalize the syntax in [ABNF](#), drawing from the [syntax guide](#) and [OBJECTIVE C implementation](#). Note that parsing actually occurs at the line level so the grammar should be considered a guide.

```
;; The grammar is ambiguous, requiring lots of lookahead, or backtracking.

;; ABNF is used here, but no BNF variant suits the grammar perfectly. Some characters will
;; be described as regular expressions inside prose values (i.e., <regex:...>) as it's
;; more concise than enumerating multiple character ranges; \p{...} and \P{...}
;; (having/not-having Unicode property) and [:defined-set:] notations will be
;; used, as well as <lookahead:...>, which is self-explanatory.

; A screenplay is defined as an optional cover page (a list of cover entries) followed by
; an also optional script (script elements.)
fountain-screenplay = *cover-entry *script-element

cover-entry = cover-key ":" *space cover-value newline

; For the cover-key, the values:
; ("TITLE" / "CREDIT" / "AUTHOR" / "SOURCE" / "DRAFT" 1*SPACE "DATE" / "CONTACT")
; are printed in the cover page. Any other keys are regarded as metadata and ignored.
cover-key = 1*<regex:[^:[:newline-char:]]>

; A cover value follows on the same line, or has multiple indented lines starting in the
; line below
cover-value = single-value / multi-line-value

; Single value follows right after the colon
single-value = 1*non-newline

; Multi-values are preceded by newlines and spaces
multi-line-value = 1*(newline 1*space 1*non-newline)

; At the highest level, a script can have sections, synopses, transitions and scene headers --
; though in practice some authors include bits of prose and scene contents (i.e., action and
; dialogue) before the first explicit scene header, as if there was an implicit first scene.
; Thus the script content begins, by default, at the zeroth scene and the zero-level section
; (which encompasses the whole document and all sections.)
script-element = *space (section / synopsis / boneyard / note)
script-element = / 1*empty-line *space (section / synopsis / boneyard / note / transition / scene / scene-content)

; The section indicator starts with one or more hashes, indicating section hierarchy
; with the number of hashes. Thus, the highest explicitly declared section level in the
; hierarchy is level one, always under zero. The section encompasses all content below
; until the next section markup.
section = 1*"#" 1*non-newline newline <lookahead: empty-line>

; A synopsis is a single line starting with an equals sign.
```

```

synopse = "=" *non-newline newline

; A boneyard is non-nestable C-style comment (multiline) that can happen just anywhere.
boneyard = "/" * (<regex:[^*]> / "*" <lookahead: [^/]>) "*" /

; A note goes between double-brackets. Can appear everywhere just like boneyards.
; However, notes can't
note = "[[" "]"

; The empty line only contains spaces, if anything
empty-line = *space newline

; Transitions come in three types: forced, uppercase (ending in TO:) and commonplace transitions.
; In all cases,
transition = (forced-transition / uppercase-transition / common-transition) newline <lookahead: empty-line>

; If it begins with ">", we need to make sure it doesn't end in "<" as that's centered text
forced-transition = ">" 1*<regex:[^[:newline-char:]]>

; Uppercase transitions are all uppercase end in "TO:"
uppercase-transition = 1*<regex:[^p{Ll}[:newline-char:]]> 1*space %s"TO" *space ":" *space

; Some extra patterns that represent transitions, such as cuts, dissolves and fades (including
; "fade in" and "fade out" at the beginning/end of the script.) Allow a period, colon or
; ellipsis at the end of such sentences.
common-transition = (fade-transition / cut-dissolve-transition) *space [{"." [".."] / ":"}] *space]

fade-transition = "FADE" 1*space ("IN" / "OUT" / "TO")

cut-dissolve-transition = ("CUT" / "DISSOLVE") 1*space "TO"

; A scene has a heading and content
scene = scene-heading scene-content

; The heading is either a forced scene (starting with ".") or starts with INT/EXT/EST combinations.
; Includes a description and an optional scene number, followed by an empty line.
scene-heading = ( "." / int-ext) scene-description [scene-number] newline <lookahead: empty-line>

; Scenes might begin with I[NT] or E[ST/XT]
int-ext = inte / esxt

inte = "I" ("/E" int-ext-ender / "." ("/E" int-ext-ender / *space) / nte)

int-ext-ender = "." *space / 1*space

nte = "NT" ("/EXT" int-ext-ender / "." ("/EXT" int-ext-ender / *space))

esxt = "E" ["ST" / "XT"] int-ext-ender

; Scene descriptions are merely prose. Note hashes are not excluded as there might be
; numbered characters or props, even if a scene-number might follow. This is handled
; with lookahead.
scene-description = 1*<regex:[^[:newline-char:]]>

; Scene numbers admit alphanumerics, dashes and periods, surrounded by hashes.
scene-number = "#" 1*scene-number-character "#" *space

scene-number-character = alphanumeric / "-" / "."

;;power-action-line = "!" *<regex:[^!\n]> "\n"

;;power-character-line = "@ 1*<regex:[^[:newline-char:]]> [{"(" <regex:[^[:newline-char:]]> ")] *space ["^"
*space]

vtab = %x0B

ff = %x0C

newline = CR [LF]
/ LF [CR]
/ vtab ; We interpret vertical tabbing as a newline too

```

```

/ ff      ; Same for form-feeds
/ %x0085  ; Unicode next-line
/ %x2028  ; Unicode line-separator
/ %x2029  ; Unicode paragraph-separator
; These are all converted into your OS's native newline at the end.

newline-char = CR / LF / vtab / ff / %x0085 / %x2028 / %x2029 ; characters used in the former

space = SP      ; normal space
/ HTAB         ; tabulator -- converts into 4 spaces
/ %x00A0       ; non-breaking
/ %x2000-2009  ; varying-width Em/En-based spaces
/ %x202F       ; narrow non-breaking
/ %x205F       ; mathematical middle-space
/ %x3000       ; Ideographic space
; These are turned into one or more fixed-width spaces (SP); we're trying to imitate
; a typewriter.
; Hairline or zero-width spaces and joiners are removed previous to parsing.
; Same goes for any control characters not listed as space or newline.

alpha = <regex:[\p{L}]>

numeric = <regex:[\p{N}]>

alphanumeric = alpha / numeric

non-newline = <regex:[^[:newline-char:]]>

non-newline-or-hash = <regex:[^[:newline-char:]]#>

```

## Building

GHC 9.6.7 and CABAL 3.0 (or greater) are required to compile the library and run the tests (*not implemented yet.*)

The project uses the GHC2021 language default. While it might be possible to compile it in earlier versions than 9.6.7, this default is only available since 9.2.1, constituting a hard limit.

Some of the included scripts require `make`, `sed` and other similar utilities usually found in LINUX or LINUX-like environments (e.g., [MSYS2](#). For WINDOWS users, it is recommended to use the [GHCUP](#) distribution, allow the installer script to deploy MSYS2 and then install the development packages.)

## Contact

Please [create an issue](#) if you find a bug.

I can be reached at `10951848+CübQfJúdãhsLîòn ä(t) users/noreply/github/còm` (without diacritics and replacing slashes by periods.)