```clojure
;; see https://ask.clojure.org/index.php/10905/control-transient-deps-that-compiled-assembled-into-uberjar?show=10913#c10913
(require 'clojure.tools.deps.alpha.util.s3-transporter)

(ns build
  (:refer-clojure :exclude [compile])
  (:require #_[babashka.fs :as fs]
            [clojure.tools.build.api :as b]))

(def lib 'net.b12n.viip/viip)

(def class-dir "target/classes")

(def uber-file "target/viip.jar")

(def src-dirs ["src" "resources"])

(def basis (b/create-basis))

(defn uber [_]
  (println "Writing pom")
  (b/write-pom {:class-dir class-dir
                :lib lib
                :version "1.0.0"
                :basis basis
                :src-dirs ["src" "resources"]})
  (b/copy-dir {:src-dirs src-dirs
               :target-dir class-dir})

  (println "Compile sources to classes")
  (b/compile-clj {:basis basis
                  :src-dirs src-dirs
                  :class-dir class-dir
                  :ns-compile '[net.b12n.viip.core]})

  (println "Building uberjar")
  (b/uber {:class-dir class-dir
           :uber-file uber-file
           :basis basis
           :main 'net.b12n.viip.core}))
```

```clojure
(ns net.b12n.viip.core
  (:gen-class)
  (:require
   [babashka.fs :as fs]
   [babashka.process :refer [process shell]]
   [clojure.java.io :as io]
   [clojure.string :as str]
   [docopt.core :as docopt]
   [selmer.parser :refer [render]]))

(defn map-keys
  "Given a function and a map, returns the map resulting from applying
  the function to each key.
  e.g. (map-keys name {:a 1 :b 2 :c 3}) ;;=> {\"a\" 1, \"b\" 2, \"c\" 3}
  "
  [f m]
  (zipmap (map f (keys m)) (vals m)))

(defn transform-keys
  "Transform the options arguments

  ;; a) where keys contain `--`
  (transform-keys {\"--profile\" \"dev\", \"--region\" \"us-east-one\" })
  ;;=> {:profile \"dev\" :region \"us-east-1\"}

  ;; b) work with keys that does not have `--` in it
  (transform-keys {\"--profile\" \"dev\", \"create\" \"true\" })
  ;;=> {:profile \"dev\" :create \"true\"} "
  [opts]
  (map-keys (fn [x] (-> x (str/replace-first "--" "") keyword)) opts))

(defn ^:private source->xhtml
  "Font that confirm to work `Source Code Pro`, `Fira Code Mono`, and `Fira Code`"
  [& [{:keys [input-file color-scheme font-name line-numbers?]
       :or {color-scheme "jellybeans"
            font-name "Source Code Pro"}}]]
  (let [opts ["let g:html_expand_tabs = 1"
              ;"let g:html_use_css = 1"      ;; NOTE: this will throw all other options off!
              ;"let g:html_no_progress = 1" ;; NOTE: gives errors when use
              (format "let g:html_number_lines = %s" (if line-numbers? 1 0))
              "let g:html_no_progress = 1"
              "let g:html_use_xhtml = 1"
              "let g:html_ignore_folding = 1"
              (format "let g:html_font = \"%s\"" font-name)
              (format "colorscheme %s" color-scheme)
              "TOhtml"
              "w"
              "qa!"]
        c-opts (->> opts (map (fn [x] ["-c" (format "'%s'" x)])) flatten)
        cmd-list (flatten (conj ["vim" "-E"] c-opts input-file))
        cmd-str (str/join " " cmd-list)]
    (when-let [{:keys [out exit err]} (shell cmd-str)]
      (if (zero? exit)
        out
        err))))

(defn ^:private html->pdf
  [& [{:keys [input-file output-file title]}]]
  (let [opts ["--margin-top" "5mm" ;"2cm"
              "--margin-bottom" "5mm" ;"2cm"
              "--margin-left" "0mm" ;"2cm"
              "--margin-right" "0mm" ;"2cm"
              "--title" title ;; NOTE: maybe simplify this a bit
              ;"--header-font-name" "times"
              ;"--header-center" "[title]::[page]/[topage]"
              "--header-left" "[webpage]"
              "--header-right" "[page] of [topage]"
              ;"--header-line"
              ;"--header-center" "[webpage]::[page]/[topage]"
              "--header-spacing" "2" ;; "mm"
              "--header-font-size" "8"
              "--footer-spacing" "2" ;; "mm"
              "--footer-font-size" "8"
              #_"--footer-line"]
        opt-str (->> opts
                     (map (fn [x] (format "'%s'" x)))
                     (str/join " "))
        cmd-str (format "%s %s %s %s"
                        "wkhtmltopdf"
                        opt-str
                        input-file
                        output-file)]
    (when-let [{:keys [out exit] :as response} (shell cmd-str)]
      (if (zero? exit)
        out
        response))))

(defn ^:private pages-count
  "Extract page count using `pdfinfo`"
  [input-pdf]
  (as-> (process (format "pdfinfo %s" input-pdf)) $
    (:out $)
    (slurp $)
    (str/split-lines $)
    (filter (fn [x] (str/starts-with? x "Pages:")) $)
    (first $)
    (str/split $ #":")
    (last $)
    (str/trim $)
    (Integer/parseInt $)))

(defn ^:private pages-numbering
  [input-files]
  (when (seq input-files)
    (let [xs (->> input-files
                  (map (fn [x] (let [fname (str x)] [fname (pages-count fname)])))
                  (sort-by first) ;; Note: maybe not needed?
                  (map-indexed vector))]
      (loop [[y & ys] xs
             total 0
             rs []]
        (if-not (seq y)
          rs
          (let [[idx entry] y
                [fname page] entry]
            (if (= idx 0)
              (recur ys page (conj rs [fname 1]))
              (recur ys (+ total page) (conj rs [fname (+ total 1)])))))))))

(comment
  (require '[clojure.set :as set])
  (set/difference (set (map str (fs/glob "../core.async" "**{.clj,cljc,cljs}"))) ;;=> 50
                  (set (map str (fs/glob "../core.async" "**{.clj,.cljc,.cljs}"))))
  (comment
    #{"../core.async/src/test/cljs/cljs"
      "../core.async/src/main/clojure/cljs"
      "../core.async/src/test/cljs"})
  nil)

;; Include hidden files
(defn create-pdfmarks
  [& [{:keys [title author tags input-files output-file]
       :or {tags ["pdf" "utility" "printing"]
            output-file "pdfmarks"}}]]
  (when (seq input-files)
    (let [pdf-files (pages-numbering input-files)
          pdfmarks-template (slurp (io/resource "pdfmarks.tmpl"))]
      (spit output-file (with-out-str
                          (println (render pdfmarks-template {:title title
                                                              :author author
                                                              :tags (str/join ", " tags)
                                                              :pdf-files pdf-files})))))))

(defn merge-pdfs
  [& [{:keys [gs-binary paper-size input-files pdfmarks output-file]
       :or {gs-binary "gs"
            paper-size "letter"
            output-file "merged-pdfs.pdf"
            pdfmarks "pdfmarks"}}]]
  (let [opts [gs-binary
              "-q"
              "-dN"
              "-dNOPAUSE"
              "-dBATCH"
              "-sDEVICE=pdfwrite"
              (format "-sPAPERSIZE=%s" (name paper-size))
              (format "-sOutputFile=%s" output-file)]
        cmd-str (->> (conj opts input-files pdfmarks)
                     flatten
                     (str/join " "))]
    (when-let [{:keys [out exit] :as response} (shell cmd-str)]
      (if (zero? exit)
        out
        response))))

(def usage "viip - Vim Interactive Printing

Usage:
  viip [options] --base-dir=<base-dir> --exts=<exts> --output-file=<output-file> [ --preset=<preset>
| --color-scheme=<color-scheme> --font-name=<font-name> ]
  viip --list-presets
  viip --version
  viip --help

Options:
  -b, --base-dir=<base-dir>           Base directory [default: .]
  -e, --exts=<exts>                   Extension to print [default: clj,cljc,cljs ]
  -p, --preset=<preset>               Preset to use [default: seoul256-dark]
  -o, --output-file=<output-file>     Output file [default: codes.pdf]
  -c, --color-scheme=<color-scheme>   Color scheme to use [default: jellybeans]
  -f, --font-name=<font-name>         Font name to use [default: Fira Code]
  -z, --paper-size=<paper-size>       Output paper size to use [default: letter]
  -n, --line-numbers                  Print line numbers
  -t, --title=<title>                 Title to use [default: viip]
  -a, --author=<author>               Author for the document [default: viip ]
  -s, --tags=<tags>                   Tags to use [default: codes,pdf,command-line]
  -l, --list-presets                  List available presets
  -h, --help                          Print help
  -v, --version                       Display version information

  # ------------------------------------------------------------ ##
  # Common Usage:
  # ------------------------------------------------------------ ##
  ~/bin/viip -b . -e clj,cljc,cljs -c github -f \"Fira Code\"       -t viip-demo -a viip -o code-
github-fira-code.pdf
  ~/bin/viip -b . -e clj,cljc,cljs -c github -f \"Fira Code Mono\"  -t viip-demo -a viip -o code-
github-fira-code-mono.pdf
  ~/bin/viip -b . -e clj,cljc,cljs -c github -f \"Inconsolata\"     -t viip-demo -a viip -o code-
github-inconsolata.pdf

  ~/bin/viip -b . -e clj,cljc,cljs -c envy -f \"Fira Code\"       -t viip-demo -a viip -o code-envy-
fira-code.pdf
  ~/bin/viip -b . -e clj,cljc,cljs -c envy -f \"Fira Code Mono\"  -t viip-demo -a viip -o code-envy-
fira-code-mono.pdf
  ~/bin/viip -b . -e clj,cljc,cljs -c envy -f \"Inconsolata\"     -t viip-demo -a viip -o code-envy-
inconsolata.pdf

  ~/bin/viip -b . -e clj,cljc,cljs -c onehalflight -f \"Fira Code\"       -t viip-demo -a \"Vim
Interactive Printer\" -o code-onehalflight-fira-code.pdf
  ~/bin/viip -b . -e clj,cljc,cljs -c onehalflight -f \"Fira Code Mono\"  -t viip-demo -a \"Vim
Interactive Printer\" -o code-onehalflight-fira-code-mono.pdf
  ~/bin/viip -b . -e clj,cljc,cljs -c onehalflight -f \"Inconsolata\"     -t viip-demo -a \"Vim
Interactive Printer\" -o code-onehalflight-inconsolata.pdf

  ~/bin/viip -b . -e clj,cljc,cljs -c onehalfdark -f \"Fira Code\"       -t viip-demo -a \"Vim
Interactive Printer\" -o code-onehalfdark-fira-code.pdf
  ~/bin/viip -b . -e clj,cljc,cljs -c onehalfdark -f \"Fira Code Mono\"  -t viip-demo -a \"Vim
Interactive Printer\" -o code-onehalfdark-fira-code-mono.pdf
  ~/bin/viip -b . -e clj,cljc,cljs -c onehalfdark -f \"Inconsolata\"     -t viip-demo -a \"Vim
Interactive Printer\" -o code-onehalfdark-inconsolata.pdf

  ~/bin/viip -b . -e clj,cljc,cljs -c onehalfdark -f \"Fira Code\"       -t viip-demo -a \"Vim
Interactive Printer\" -o code-onehalfdark-fira-code.pdf
  ~/bin/viip -b . -e clj,cljc,cljs -c onehalfdark -f \"Fira Code Mono\"  -t viip-demo -a \"Vim
```

```
Interactive Printer\" -o code-onehalfdark-fira-code-mono.pdf
  ~/bin/viip -b . -e clj,cljc,cljs -c onehalfdark -f \"Inconsolata\"     -t viip-demo -a \"Vim
Interactive Printer\" -o code-onehalfdark-inconsolata.pdf

  ~/bin/viip -b . -e clj,cljc,cljs -c jellybeans  -f \"Fira Code\"       -t viip-demo -a \"Vim
Interactive Printer\" -o code-jellybeans-fira-code.pdf
  ~/bin/viip -b . -e clj,cljc,cljs -c jellybeans  -f \"Fira Code Mono\"  -t viip-demo -a \"Vim
Interactive Printer\" -o code-jellybeans-fira-code-mono.pdf
  ~/bin/viip -b . -e clj,cljc,cljs -c jellybeans  -f \"Inconsolata\"     -t viip-demo -a \"Vim
Interactive Printer\" -o code-jellybeans-inconsolata.pdf

  ~/bin/viip -b . -e clj,cljc,cljs -c PaperColor -f \"Fira Code\"       -t viip-demo -a \"Vim
Interactive Printer\" -o code-papercolor-fira-code.pdf
  ~/bin/viip -b . -e clj,cljc,cljs -c PaperColor -f \"Fira Code Mono\"  -t viip-demo -a \"Vim
Interactive Printer\" -o code-papercolor-fira-code-mono.pdf
  ~/bin/viip -b . -e clj,cljc,cljs -c PaperColor -f \"Inconsolata\"     -t viip-demo -a \"Vim
Interactive Printer\" -o code-papercolor-inconsolata.pdf

  # ------------------------------------------------------------- ##
  ## Show help
  viip -h

  # ------------------------------------------------------------- ##
  ## Show version
  viip -v")

(def presets {:seoul256-dark {:color-scheme "seoul256"
                              :font-name "Fira Code Mono"}
              :seoul256-light {:color-scheme "seoul256-light"
                               :font-name "Fira Code Mono"}
              :jellybeans {:color-scheme "jellybeans"
                           :font-name "Source Code Pro"}})

(defn ^:private validate-options
  [{:keys [base-dir
           exts
           output-file
           preset
           font-name
           color-scheme
           line-numbers
           title
           author
           tags
           paper-size]}]
  (let [base-dir (str (fs/expand-home base-dir))
        output-file (str (fs/expand-home output-file))]
    (cond
      (not (and (fs/exists? base-dir)
                (fs/directory? base-dir)))
      (println (format "base-dir '%s' is not exist or not a regular directory." base-dir))

      (not exts)
      (println "Must supply the extension like clj,cljc,edn")

      (not title)
      (println "Must supply the title like \"My Clojure Project\"")

      (not author)
      (println "Must supply the author like \"Vim Interactive Printer\"")

      (not tags)
      (println "Must supply the tags like \"pdf,print,clojure,etc\"")

      (not (or (and font-name color-scheme)
               (and preset (-> presets keys set (get (keyword preset))))))
      (println "Must supply either `font-name` with `color-scheme` Or select one of the presets " (-
>> presets keys (map name) (str/join ", ")))

      (not paper-size)
      (println "Must supply the paper-size like \"letter\" or\"a4\"")
      :else
      {:base-dir base-dir
       :exts (map str/trim (str/split exts #","))
       :preset (keyword preset)
       :font-name (if (seq font-name) (str/join " " font-name) font-name) ;; e.g. ["Source" "Code"
"Pro"] => "Source Code Pro"
       :color-scheme color-scheme
       :line-numbers? line-numbers
       :paper-size (name paper-size)
       :title title
       :author author
       :tags (map str/trim (str/split tags #","))
       :output-file (str (fs/expand-home output-file))}))))

(defn ^:private source->pdf
  [& [{:keys [base-dir
              exts
              preset
              font-name
              color-scheme
              line-numbers?]
       :or {base-dir "."}}]]
  (let [glob-expr (->> exts (map (fn [x] (format ".%s" x))) (str/join ",") (format "**{%s}"))
        input-files (->> (fs/glob base-dir glob-expr) (map str) sort)]
    (when (seq input-files)
      (let [{:keys [color-scheme font-name]} (merge (presets (keyword preset))
                                                     {:font-name font-name
                                                      :color-scheme color-scheme})]
        (doseq [input-file input-files]
          (let [base-dir (if (fs/parent input-file)
                           (str (fs/parent input-file))
                           ".")
                base-name (fs/file-name input-file)
                base-ext (fs/extension base-name)
                base-name-no-ext (str/replace base-name (format ".%s" base-ext) "")
                xhtml-file (format "%s/%s.%s.xhtml" base-dir base-name-no-ext base-ext)
                xhtml-pdf-file (format "%s/%s.%s.xhtml.pdf" base-dir base-name-no-ext base-ext)]
            (source->xhtml {:input-file input-file
                            :color-scheme color-scheme
                            :font-name font-name
                            :line-numbers? line-numbers?})
            (html->pdf {:input-file xhtml-file
                        :output-file xhtml-pdf-file
                        :title base-name})))))))

(defn cleanup
  [xhtml-files input-files]
  (println "Delete *.xhtml files ...")
  (doseq [xhtml-file xhtml-files]
    (println "Delete : " xhtml-file)
    (fs/delete-if-exists xhtml-file))

  (println "Delete *.xhtml.pdf files ...")
  (doseq [pdf-file input-files]
    (println "Delete : " pdf-file)
    (fs/delete-if-exists pdf-file)))

(defn ^:private join-pdfs
  [& [{:keys [base-dir title author paper-size gs-binary tags output-file]
       :or {base-dir "."
            author "viip"
            paper-size "letter"
            gs-binary "gs"
            output-file "merged-source.pdf"}}]]
  (let [input-files (->> (fs/glob base-dir "**{.xhtml.pdf}") (map str) sort)
        xhtml-files (->> (fs/glob base-dir "**{.xhtml}") (map str) sort)
        pdfmarks "pdfmarks"]
    (when (seq input-files)
      (create-pdfmarks
       {:title title
        :author author
        :tags tags
        :input-files input-files
        :output-file pdfmarks})
      (merge-pdfs {:gs-binary gs-binary
                   :paper-size (name paper-size)
                   :input-files input-files
                   :output-file output-file})
      (cleanup xhtml-files input-files)
      (println "Delete pdfmarks file ...")
      (fs/delete-if-exists pdfmarks))))

#_{:clj-kondo/ignore [:unused-binding]}
(defn ^:private run
  [& [{:keys [base-dir
              exts
              output-file
              preset
              font-name
              color-scheme
              paper-size
              line-numbers
              title
              author
              tags
              list-presets
              help
              version] :as args
       :or {paper-size "letter"}}]]
  (cond
    help
    (println usage)

    version
    (println "viip : v1.0.0")

    list-presets
    (println "Available presets : " (->> presets keys (map name) (str/join ", ")))

    :else
    (when-let [opts (validate-options args)]
      (println "Convert files to -> xhtml -> pdf ...")
      (source->pdf opts)

      (println "Combine multiple pdf files into one ...")
      (join-pdfs (merge opts {:gs-binary "gs"}))
      (println "Your output file is " (opts :output-file)))))

(defn -main
  [& args]
  (docopt/docopt usage args (fn [arg-map] (-> arg-map transform-keys run)))
  (System/exit 0))
```

```clojure
(ns net.b12n.viip.core-test
  (:require [net.b12n.viip.core :as sut]
            [clojure.test :as t]))

(deftest dummy-test
  (is (= 5 (+ 2 3))))
```