Holy Haskell Project Starter

yannesposito.com/Scratch/en/blog/Holy-Haskell-Starter/

tl;dr: Learn how to start a new Haskell project. Translate a starter tool written in `zsh` in Haskell using its own result.

> *"Good Sir Knight, will you come with me to Camelot, and join us at the Round Table?"*

In order to work properly with Haskell you need to initialize your environment. Typically, you need to use a cabal file, create some test for your code. Both, unit test and propositional testing (random and exhaustive up to a certain depth). You need to use `git` and generally hosting it on github. Also, it is recommended to use cabal sandboxes. And as bonus, an auto-update tool that recompile and retest on each file save.

In this article, we will create such an environment using a zsh script. Then we will write a Haskell project which does the same work as the zsh script. You will then see how to work in such an environment.

If you are starting to understand Haskell but consider yourself a beginner, this tutorial will show you how to make a real application using quite surprisingly a lot of features:

- use colorized output
- interact with a user in command line
- read/write files
- kind of parse a file (in fact, simply split it)
- use a templating system (mustache: fill a data structure, write files)

- make a HTTP GET request then parse the JSON answer and use it
- use random
- create a cabal package
- add and use non source files to a cabal package
- Test your code (both unit testing and property testing)

☞ zsh is by its nature more suitable to file manipulation. But the Haskell code is clearly more organized while quite terse for a multi-purpose language.

☞ holy-project is on hackage. It can be installed with `cabal update && cabal install holy-project`.

I recently read this excellent article: How to Start a New Haskell Project.

While the article is very good, I lacked some minor informations. Inspired by it, I created a simple script to initialize a new Haskell project. During the process I improved some things a bit.

If you want to use this script, the steps are:

1. Install Haskell
2. Make sure you have the latest `cabal-install` (at least 1.18)

```
> cabal install cabal-
install
```

3. Download and run the script

```
# Download the script
git clone https://github.com/yogsototh/init-haskell-
project.git
# Copy the script in a directory of you PATH variable
cp init-haskell-project/holy-haskell.sh ~/bin
# Go to the directory containing all your projects
cd my/projects/directory
# Launch thcript
holy-haskell.sh
```

What does this script do that `cabal init` doesn't do?

- Use cabal sandbox
- It initialize `git` with the right `.gitignore` file.
- Use `tasty` to organize your tests (HUnit, QuickCheck and SmallCheck).
- Use `-Wall` for `ghc` compilation.
- Will make references to Holy Grail
- Search your default github username via github api.

`zsh` really?

Developing the script in `zsh` was easy. But considering its size, it is worth to rewrite it in Haskell. Furthermore, it will be a good exercise.

Patricide

In a first time, we initialize a new Haskell project with `holy-haskell.sh`:

```
> ./holy-haskell.sh
Bridgekeeper: Stop!
Bridgekeeper: Who would cross the Bridge of Death
Bridgekeeper: must answer me these questions three,
Bridgekeeper: ere the other side he see.
You: Ask me the questions, bridgekeeper, I am not afraid.

Bridgekeeper: What is the name of your project?
> Holy project
Bridgekeeper: What is your name? (Yann Esposito (Yogsototh))
>
Bridgekeeper: What is your email? (Yann.Esposito@gmail.com)
>
Bridgekeeper: What is your github user name? (yogsototh)
>
Bridgekeeper: What is your project in less than ten words?
> Start your Haskell project with cabal, git and tests.
Initialize git
Initialized empty Git repository in .../holy-project/.git/
Create files
    .gitignore
    holy-project.cabal
    Setup.hs
    LICENSE (MIT)
```

```
    LICENSE (MIT)
    test/Test.hs
    test/HolyProject/Swallow/Test.hs
    src/HolyProject/Swallow.hs
    test/HolyProject/Coconut/Test.hs
    src/HolyProject/Coconut.hs
    src/HolyProject.hs
    src/Main.hs
Cabal sandboxing, install and test
...
  many compilations lines
...
Running 1 test suites...
Test suite Tests: RUNNING...
Test suite Tests: PASS
Test suite logged to: dist/test/holy-project-0.1.0.0-Tests.log
1 of 1 test suites (1 of 1 test cases) passed.
All Tests
  Swallow
    swallow test:     OK
  coconut
    coconut:          OK
    coconut property: OK
      148 tests completed

All 3 tests passed



Bridgekeeper: What... is the air-speed velocity of an unladen
swallow?
You: What do you mean? An African or European swallow?
Bridgekeeper: Huh? I... I don't know that.
[the bridgekeeper is thrown over]
Bridgekeeper: Auuuuuuuuuuuugh
Sir Bedevere: How do you know so much about swallows?
You: Well, you have to know these things when you're a king, you
know.
```

The different steps are:

- small introduction quotes
- ask five questions – *three question sir…*
- create the directory for the project
- init git
- create files
- sandbox cabal
- cabal install and test
- run the test directly in the terminal
- small goodbye quotes

Features to note:

- color in the terminal
- check some rules on the project name
- random message if error
- use `~/.gitconfig` file in order to provide a default name and email.
- use the github API which returns JSON to get the default github user name.

So, apparently nothing too difficult to achieve.

We should now have an initialized Haskell environment for us to work. The first thing you should do, is to go into this new directory and launch './auto-update' in some terminal. I personally use `tmux` on Linux or the splits in `iTerm 2` on Mac OS X. Now, any modification of a source file will relaunch a compilation and a test.

The dialogs

To print the introduction text in `zsh`:

```
# init colors
autoload colors
colors
for COLOR in RED GREEN YELLOW BLUE MAGENTA CYAN BLACK WHITE; do
    eval $COLOR='$fg_no_bold[${(L)COLOR}]'
    eval BOLD_$COLOR='$fg_bold[${(L)COLOR}]'
done
eval RESET='$reset_color'
# functions
bk(){print -- "${GREEN}Bridgekeeper: $*${RESET}"}
bkn(){print -n -- "${GREEN}Bridgekeeper: $*${RESET}"}
you(){print -- "${YELLOW}You: $*${RESET}"}
...
# the introduction dialog
bk "Stop!"
bk "Who would cross the Bridge of Death"
bk "must answer me these questions three,"
bk "ere the other side he see."
you "Ask me the questions, bridgekeeper, I am not afraid.\n"
...
# the final dialog
print "\n\n"
bk "What... is the air-speed velocity of an unladen swallow?"
you "What do you mean? An African or European swallow?"
bk "Huh? I... I don't know that."
log "[the bridgekeeper is thrown over]"
bk "Auuuuuuuuuuuugh"
log "Sir Bedevere: How do you know so much about swallows?"
you "Well, you have to know these things when you're a king, you
know."
```

In the first Haskell version I don't use colors. We see we can almost copy/paste. I just added the types.

```
bk :: String -> IO ()
bk str = putStrLn $ "Bridgekeeper: " ++ str

bkn :: String -> IO ()
bkn str = pustStr $ "Bridgekeeper: " ++ str

you :: String -> IO ()
you str = putStrLn $ "You: " ++ str

intro :: IO ()
intro = do
    bk "Stop!"
    bk "Who would cross the Bridge of Death"
    bk "must answer me these questions three,"
    bk "ere the other side he see."
    you "Ask me the questions, bridgekeeper, I am not afraid.\n"

end :: IO ()
end = do
    putStrLn "\n\n"
    bk "What... is the air-speed velocity of an unladen swallow?"
    you "What do you mean? An African or European swallow?"
    bk "Huh? I... I don't know that."
    putStrLn "[the bridgekeeper is thrown over]"
    bk "Auuuuuuuuuuuugh"
    putStrLn "Sir Bedevere: How do you know so much about swallows?"
    you "Well, you have to know these things when you're a king, you
know."
```

Now let's just add the colors using the `ansi-terminal` package. So we have to add `ansi-terminal` as a build dependency in our cabal file.

Edit `holy-project.cabal` to add it.

```
...
build-depends:  base >=4.6 &&
<4.7
                , ansi-terminal
...
```

Now look at the modified Haskell code:

```
import System.Console.ANSI

colorPutStr :: Color -> String -> IO ()
colorPutStr color str = do
    setSGR  [ SetColor Foreground Dull color
            , SetConsoleIntensity NormalIntensity
            ]
    putStr str
    setSGR []


bk :: String -> IO ()
bk str = colorPutStr Green ("Bridgekeeper: " ++ str ++ "\n")
bkn :: String -> IO ()
bkn str = colorPutStr Green ("Bridgekeeper: " ++ str)
you :: String -> IO ()
you str = colorPutStr Yellow ("You: " ++ str ++ "\n")

intro :: IO ()
intro = do
    bk "Stop!"
    bk "Who would cross the Bridge of Death"
    bk "must answer me these questions three,"
    bk "ere the other side he see."
    you "Ask me the questions, bridgekeeper, I am not afraid.\n"

end :: IO ()
end = do
    putStrLn "\n\n"
    bk "What... is the air-speed velocity of an unladen swallow?"
    you "What do you mean? An African or European swallow?"
    bk "Huh? I... I don't know that."
    putStrLn "[the bridgekeeper is thrown over]"
    bk "Auuuuuuuuuuuugh"
    putStrLn "Sir Bedevere: How do you know so much about swallows?"
    you "Well, you have to know these things when you're a king, you
know."
```

We could put this code in `src/Main.hs`. Declare a main function:

```
main :: IO ()
main = do
    intro
    end
```

Make `cabal install` and run `cabal run` (or `./.cabal-sandbox/bin/holy-project`). It works!

Five Questions – Three questions Sir!

In order to ask questions, here is how we do it in shell script:

```
print -- "What is your
name?"
read name
```

If we want to abstract things a bit, the easiest way in shell is to use a global variable which will get the value of the user input like this:

```
answer=""
ask(){
    local info="$1"
    bk "What is your
$info?"
    print -n "> "
    read answer
}
...
ask name
name="$answer"
```

In Haskell we won’t need any global variable:

```
import System.IO (hFlush, stdout)
...
ask :: String -> IO String
ask info = do
    bk $ "What is your " ++ info ++ "?"
    putStr "> "
    hFlush stdout -- Because we want to ask on the same
line.
    getLine
```

Now our main function might look like:

```
main = do
    intro
    _ <- ask "project name"
    _ <- ask "name"
    _ <- ask "email"
    _ <- ask "github account"
    _ <- ask "project in less than a dozen
word"
    end
```

You could test it with `cabal install` and then `./.cabal-sandbox/bin/holy-project`.

We will see later how to guess the answer using the `.gitconfig` file and the github API.

Using answers

Create the project name

I don't really like the ability to use capital letter in a package name. So in shell I transform the project name like this:

```
# replace all spaces by dashes then lowercase the
string
project=${${project:gs/ /-/}:l}
```

In order to achieve the same result in Haskell (don't forget to add the `split` package):

```
import Data.Char        (toLower)
import Data.List        (intercalate)
import Data.List.Split  (splitOneOf)
...
projectNameFromString :: String -> String
projectNameFromString str = intercalate "-" (splitOneOf " -" (map toLower
str))
```

One important thing to note is that in zsh the transformation occurs on strings but in haskell we use list as intermediate representation:

```
zsh:
"Holy grail" ==( ${project:gs/ /-/} )=> "Holy-grail"
             ==( ${project:l}       )=> "holy-
grail"

haskell
"Holy grail" ==( map toLower     )=> "holy grail"
             ==( splitOneOf " -" )=>
["holy","grail"]
             ==( intercalate "-" )=> "holy-grail"
```

Create the module name

The module name is a capitalized version of the project name where we remove dashes.

```
# Capitalize a string
capitalize(){
    local str="$(print -- "$*" | sed 's/-/
/g')"
    print -- ${(C)str} | sed 's/ //g'
}
```

```
-- | transform a chain like "Holy project" in "HolyProject"
capitalize :: String -> String
capitalize str = concatMap capitalizeWord (splitOneOf " -"
str)
    where
        capitalizeWord :: String -> String
        capitalizeWord (x:xs)   = toUpper x:map toLower xs
        capitalizeWord  _       = []
```

The haskell version is made by hand where zsh already had a capitalize operation on string with many words. Here

is the difference between the shell and haskell way (note I splitted the effect of `concatMap` as `map` and `concat`):

```
shell:
"Holy-grail" ==( sed 's/-/ /g' )=> "Holy grail"
             ==( ${(C)str}     )=> "Holy Grail"
             ==( sed 's/ //g'  )=> "HolyGrail"

haskell:
"Holy-grail" ==( splitOneOf " -"    )=>
["Holy","grail"]
             ==( map capitalizeWord )=>
["Holy","Grail"]
             ==( concat             )=> "HolyGrail"
```

As the preceding example, in shell we work on strings while Haskell use temporary lists representations.

Check the project name

Also I want to be quite restrictive on the kind of project name we can give. This is why I added a check function.

```
ioassert :: Bool -> String -> IO ()
ioassert True _ = return ()
ioassert False str = error str

main :: IO ()
main = do
  intro
  project <- ask "project name"
  ioassert (checkProjectName project)
       "Use only letters, numbers, spaces and dashes
please"
  let projectname = projectNameFromString project
      modulename = capitalize project
```

Which verify the project name is not empty and use only letter, numbers and dashes:

```
-- | verify if project name is conform
checkProjectName :: String -> Bool
checkProjectName [] = False
checkProjectName str =
    all (\c -> isLetter c || isNumber c || c=='-' || c==' ')
str
```

Create the project

Making a project will consists in creating files and directories whose name and content depends on the answer we had until now.

In shell, for each file to create, we used something like:

```
> file-to-create cat <<END
file content here.
We can use $variables
here
END
```

In Haskell, while possible, we shouldn't put the file content in the source code. We have a relatively easy way to include external file in a cabal package. This is what we will be using.

Furthermore, we need a templating system to replace small part of the static file by computed values. For this task, I choose to use `hastache`, a Haskell implementation of Mustache templates.

Add external files in a cabal project

Cabal provides a way to add files which are not source files to a package. You simply have to add a `Data-Files:` entry in the header of the cabal file:

```
data-files: scaffold/LICENSE
            , scaffold/Setup.hs
            , scaffold/auto-update
            , scaffold/gitignore
            , scaffold/interact
            , scaffold/project.cabal
            , scaffold/src/Main.hs
            , scaffold/src/ModuleName.hs
            , scaffold/src/ModuleName/Coconut.hs
            , scaffold/src/ModuleName/Swallow.hs
            ,
scaffold/test/ModuleName/Coconut/Test.hs
            ,
scaffold/test/ModuleName/Swallow/Test.hs
            , scaffold/test/Test.hs
```

Now we simply have to create our files at the specified path. Here is for example the first lines of the LICENSE file.

```
The MIT License (MIT)

Copyright (c) {{year}} {{author}}

Permission is hereby granted, free of charge, to any person obtaining a
copy
...
```

It will be up to our program to replace the `{{year}}` and `{{author}}` at runtime. We have to find the files. Cabal will create a module named `Paths_holy_project`. If we import this module we have the function `genDataFileName` at our disposal. Now we can read the files at runtime like this:

```
  ...
  do
    pkgFilePath     <- getDataFileName
"scaffold/LICENSE"
    templateContent <- readFile pkgFilePath
    ...
```

Create files and directories

A first remark is for portability purpose we shouldn't use String for file path. For example on Windows `/` isn't considered as a subdirectory character. To resolve this problem we will use `FilePath`:

```
import System.Directory
import System.FilePath.Posix        (takeDirectory,(</>))
...
createProject ... = do
      ...
      createDirectory projectName     -- mkdir
      setCurrentDirectory projectName -- cd
      genFile "LICENSE" "LICENSE"
      genFile "gitignore" ".gitignore"
      genFile "src/Main.hs" ("src" </> "Main.hs")

genFile dataFilename outputFilename = do
    pkgfileName <- getDataFileName ("scaffold/" ++ filename)
    template <- readFile pkgfileName
    transformedFile <- ??? -- hastache magic here
    createDirectoryIfMissing True (takeDirectory
outputFileName)
    writeFile outputFileName transformedFile
```

Use Hastache

In order to use hastache we can either create a context manually or use generics to create a context from a record. This is the last option we will show here. So in a first time, we need to import some modules and declare a record containing all necessary informations to create our project.

```
{-# LANGUAGE DeriveDataTypeable #-}
...
import Data.Data
import Text.Hastache
import Text.Hastache.Context
import qualified Data.ByteString            as
BS
import qualified Data.ByteString.Lazy.Char8 as
LZ


data Project = Project {
    projectName   :: String
    , moduleName    :: String
    , author        :: String
    , mail          :: String
    , ghaccount     :: String
    , synopsis      :: String
    , year          :: String
    } deriving (Data,
Typeable)
```

Once we have declared this, we should populate our Project record with the data provided by the user. So our main function should look like:

```
main :: IO ()
main = do
    intro
    project <- ask "project name"
    ioassert (checkProjectName project)
             "Use only letters, numbers, spaces and dashes please"
    let projectname = projectNameFromString project
        modulename  = capitalize project
    in_author       <- ask "name"
    in_email        <- ask "email"
    in_ghaccount    <- ask "github account"
    in_synopsis     <- ask "project in less than a dozen word?"
    current_year    <- getCurrentYear
    createProject $ Project projectname modulename in_author
in_email
                            in_ghaccount in_synopsis current_year
    end
```

Finally we could use hastache this way:

```
createProject :: Project -> IO ()
createProject p = do
    let context = mkGenericContext p
    createDirectory (projectName p)
    setCurrentDirectory (projectName p)
    genFile context "gitignore"      $ ".gitignore"
    genFile context "project.cabal"  $ (projectName p) ++
".cabal"
    genFile context "src/Main.hs")   $ "src" </> "Main.hs"
    ...

genFile :: MuContext IO -> FilePath -> FilePath -> IO ()
genFile context filename outputFileName = do
    pkgfileName <- getDataFileName ("scaffold/"++filename)
    template <- BS.readFile pkgfileName
    transformedFile <- hastacheStr defaultConfig template context
    createDirectoryIfMissing True (takeDirectory outputFileName)
    LZ.writeFile outputFileName transformedFile
```

We use external files in mustache format. We ask question to our user to fill a data structure. We use this data structure to create a context. Hastache use this context with the external files to create the project files.

Git and Cabal

We need to initialize git and cabal. For this we simply call external command with the `system` function.

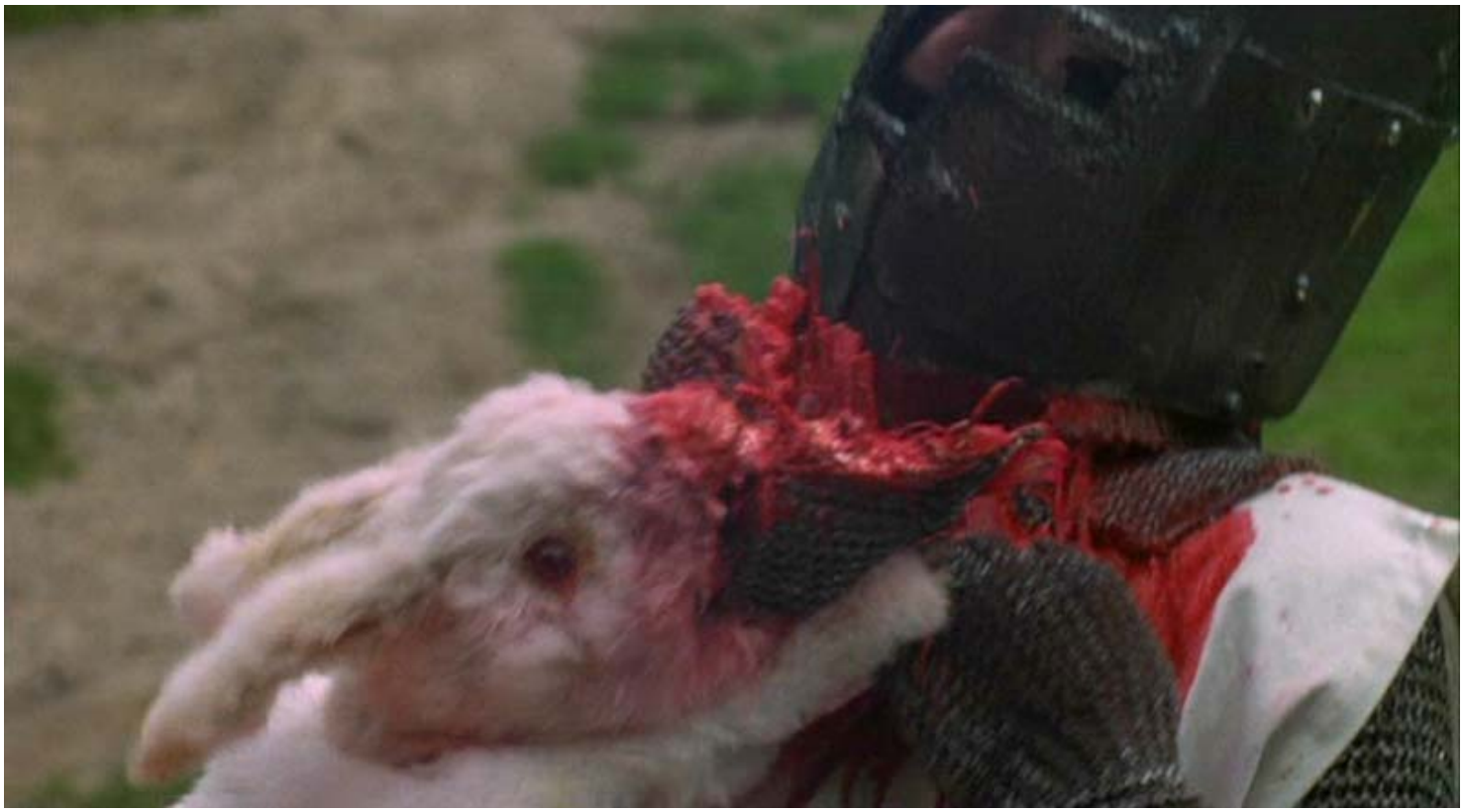
```
import System.Cmd

...
main = do
    ...
    _ <- system "git init ."
    _ <- system "cabal sandbox init"
    _ <- system "cabal install"
    _ <- system "cabal test"
    _ <- system $ "./.cabal-sandbox/bin/test-" ++
projectName
```

Ameliorations

Our job is almost finished. Now, we only need to add some nice feature to make the application more enjoyable.

Better error message



The first one would be to add a better error message.

```
import System.Random

holyError :: String -> IO ()
holyError str = do
    r <- randomIO
    if r
        then
            do
                bk "What... is your favourite colour?"
                you "Blue. No, yel..."
                putStrLn "[You are thrown over the edge into the
volcano]"
                you "You: Auuuuuuuuuuuugh"
                bk " Hee hee heh."
        else
            do
                bk "What is the capital of Assyria?"
                you "I don't know that!"
                putStrLn "[You are thrown over the edge into the
volcano]"
                you "Auuuuuuuuuuuugh"
    error ('\n':str)
```

And also update where this can be called

```
ioassert :: Bool -> String -> IO
()
ioassert True _ = return ()
ioassert False str = holyError str
```

Use `.gitconfig`

We want to retrieve the `~/.gitconfig` file content and see if it contains a name and email information. We will need to access to the `HOME` environment variable. Also, as we use bytestring package for hastache, let's take advantage of this library.

```
import Data.Maybe           (fromJust)
import System.Environment   (getEnv)
import Control.Exception
import System.IO.Error
import Control.Monad        (guard)

safeReadGitConfig :: IO LZ.ByteString
safeReadGitConfig = do
    e <- tryJust (guard . isDoesNotExistError)
                 (do
                    home <- getEnv "HOME"
                    LZ.readFile $ home ++ "/.gitconfig"
)
    return $ either (const (LZ.empty)) id e
...
main = do
    gitconfig <- safeReadGitConfig
    let (name,email) = getNameAndMail gitconfig
    project <- ask "project name" Nothing
    ...
    in_author       <- ask "name" name
    ...
```

We could note I changed the ask function slightly to take a maybe parameter.

```
ask :: String -> Maybe String -> IO String
ask info hint = do
    bk $ "What is your " ++ info ++ "?" ++ (maybe "" (\h -> " ("++h++")")
hint)
    ...
```

Concerning the parsing of `.gitconfig`, it is quite minimalist.

```
getNameAndMail :: LZ.ByteString -> (Maybe String,Maybe String)
getNameAndMail gitConfigContent = (getFirstValueFor splitted "name",
                                   getFirstValueFor splitted "email")
    where
        -- make lines of words
        splitted :: [[LZ.ByteString]]
        splitted = map LZ.words (LZ.lines gitConfigContent)

-- Get the first line which start with
-- 'elem =' and return the third field (value)
getFirstValueFor :: [[LZ.ByteString]] -> String -> Maybe String
getFirstValueFor splitted key = firstJust (map (getValueForKey key)
splitted)

-- return the first Just value of a list of Maybe
firstJust :: (Eq a) => [Maybe a] -> Maybe a
firstJust l = case dropWhile (==Nothing) l of
    [] -> Nothing
    (j:_) -> j

-- Given a line of words ("word1":"word2":rest)
-- getValue will return rest if word1 == key
-- 'elem =' or Nothing otherwise
getValueForKey :: String            -- key
                  -> [LZ.ByteString] -- line of words
                  -> Maybe String    -- the value if found
getValueForKey el (n:e:xs) = if (n == (LZ.pack el)) && (e == (LZ.pack "="))
                       then Just (LZ.unpack (LZ.unwords xs))
                       else Nothing
getValueForKey _ _ = Nothing
```

We could notice, `getNameAndMail` doesn't read the full file and stop at the first occurrence of name and mail.

Use the github API

The task seems relatively easy, but we'll see there will be some complexity hidden. Make a request on `https://api.github.com/search/users?q=<email>`. Parse the JSON and get the `login` field of the first item.

So the first problem to handle is to connect an URL. For this we will use the `http-conduit` package.

Generally, for simple request, we should use:

```
do
    body <- simpleHTTP ("https://api.github.com/search/users?q=" ++
email)
    ...
```

But, after some research, I discovered we must declare an User-Agent in the HTTP header to be accepted by the github API. So we have to change the HTTP Header, and our code became slightly more complex:

```
{-# LANGUAGE OverloadedStrings #-}
...
simpleHTTPWithUserAgent :: String -> IO LZ.ByteString
simpleHTTPWithUserAgent url = do
    r  <- parseUrl url
    let request = r { requestHeaders =  [ ("User-Agent","HTTP-Conduit") ]
}
    withManager $ (return.responseBody) <=< httpLbs request

getGHUser :: String -> IO (Maybe String)
getGHUser ""    = return Nothing
getGHUser email = do
            let url = "https://api.github.com/search/users?q=" ++ email
            body <- simpleHTTPWithUserAgent url
            ...
```

So now, we have a String containing a JSON representation. In javascript we would have used `login=JSON.parse(body).items[0].login`. How does Haskell will handle it (knowing the J in JSON is for Javascript)?

First we will need to add the `lens-aeson` package and use it that way:

```
import Control.Lens.Operators           ((^?))
import Control.Lens.Aeson
import Data.Aeson.Encode                (fromValue)
import qualified Data.Text.Lazy         as TLZ
import qualified Data.Text.Lazy.Builder as TLB

getGHUser :: String -> IO (Maybe String)
getGHUser email = do
    let url = "https://api.github.com/search/users?q=" ++ email
    body <- simpleHTTPWithUserAgent url
    let login = body ^? key "items" . nth 0 . key "login"
    return $ fmap jsonValueToString login
    where
        jsonValueToString = TLZ.unpack . TLB.toLazyText .
fromValue
```

It looks ugly, but it's terse. In fact each function `(^?)`, `key` and `nth` has some great mathematical properties and everything is type safe. Unfortunately I had to make my own `jsonValueToString`. I hope I simply missed a simpler existing function.

You can read this article on `lens-aeson` and prisms to know more.

Concurrency

We now have all the feature provided by the original `zsh` script shell. But here is a good occasion to use some Haskell great feature.

We will launch the API request sooner and in parallel to minimize our wait time:

```
import Control.Concurrent
...
main :: IO ()
main = do
    intro
    gitconfig <- safeReadGitConfig
    let (name,email) = getNameAndMail gitconfig
    earlyhint <- newEmptyMVar
    maybe   (putMVar earlyhint Nothing) -- if no email found put Nothing
            (\hintmail -> do  -- in the other case request the github
API
                forkIO (putMVar earlyhint =<< getGHUser hintmail)
                return ())
            email
    project <- ask "project name" Nothing
    ioassert (checkProjectName project)
             "Use only letters, numbers, spaces and dashes please"
    let projectname = projectNameFromString project
        modulename  = capitalize project
    in_author       <- ask "name" name
    in_email        <- ask "email" email
    ghUserHint      <- if maybe "" id email /= in_email
                          then getGHUser in_email
                          else takeMVar earlyhint
    in_ghaccount    <- ask "github account" ghUserHint
    in_synopsis     <- ask "project in less than a dozen word?" Nothing
    current_year    <- getCurrentYear
    createProject $ Project projectname modulename in_author in_email
                          in_ghaccount in_synopsis current_year
    end
```

While it might feel a bit confusing, it is in fact quite simple.

1. declare an `MVar`. Mainly a variable which either is empty or contains something.
2. If we didn't found any email hint, put Nothing in the `MVar`.
3. If we have an email hint, ask on the github API in a new process and once finished put the result in the `MVar`.
4. If the user enter an email different from the hint email, then just request the github api now.
5. If the user enter the same email, then wait for the MVar to be filled and ask the next question with the result.

If you have a github account and had set correctly your `.gitconfig`, you might not even wait.

Project Structure

We have a working product. But, I don't consider our job finished. The code is about 335 lines.

Considering that we:

- have 29 lines of import and 52 lines of comments (rest 255 lines)
- ask questions
- use a templating system to generate files
- call an asynchronous HTTP request

- parse JSON
- parse `.gitconfig`
- use colored output

This is quite few.

Modularizing

For short programs it is not obvious to split them into different modules. But my personal preference is to split it anyway.

First we put all content of `src/Main.hs` in `src/HolyProject.hs`. We rename the `main` function by `holyStarter`. And our `src/Main.hs` should contains:

```
module Main where
import
HolyProject
main :: IO ()
main =
holyStarter
```

Of course you have to remember to rename the module of `src/HolyProject.hs`. I separated all functions in different submodules:

- `HolyProject.GitConfig`
    - `getNameAndMailFromGitConfig`: retrieve name an email from `.gitconfig` file

- `HolyProject.GithubAPI`
  - `searchGHUser`: retrieve github user name using github API.
- `HolyProject.MontyPython`
  - `bk`: bridge keeper speaks
  - `you`: you speak
  - `ask`: Ask a question and wait for an answer
- `HolyProject.StringUtils`: String helper functions
  - `projectNameFromString`
  - `capitalize`
  - `checkProjectName`

The `HolyProject.hs` file contains mostly the code that ask questions, show errors and copy files using hastache.

One of the benefits in modularizing the code is that our main code is clearer. Some functions are declared only in a module and are not exported. This help us hide technical details. For example, the modification of the HTTP header to use the github API.

Documenting

We didn't take much advantage of the project structure yet. A first thing is to generate some documentation. Before most function I added comment starting with `-- |`. These comment will be used by haddock to create a documentation. First, you need to install `haddock` manually.

```
cabal install
haddock
```

Be sure to have `haddock` in your `PATH`. You could for example add it like this:

```
# You might want to add this line in your
.profile
export PATH=$PATH:./.cabal-sandbox/bin
```

And if you are at the root of your project you'll get it. And now just launch:

```
cabal haddock
```

And magically, you'll have a documentation in `dist/doc/html/holy-project/index.html`.

Tests

While the Haskell static typing is quite efficient to prevent entire classes of bugs, Haskell doesn't discard the need to test to minimize the number of bugs.

Unit Testing with HUnit

It is generally said to test we should use unit testing for code in IO and QuickCheck or SmallCheck for pure code.

A unit test example on pure code is in the file `test/HolyProject/Swallow/Test.hs`:

```
module HolyProject.Swallow.Test
    (swallowSuite)
where
import Test.Tasty (testGroup, TestTree)
import Test.Tasty.HUnit
import HolyProject.Swallow (swallow)

swallowSuite :: TestTree
swallowSuite = testGroup "Swallow"
    [testCase "swallow test" testSwallow]

-- in Swallow: swallow = (++)
testSwallow :: Assertion
testSwallow = "something" @=? swallow "some" "thing"
```

Note `swallow` is `(++)`. We group tests by group. Each group can contain some test suite. Here we have a test suite with only one test. The `(@=?)` verify the equality between its two parameters.

So now, we could safely delete the directory `test/HolyProject/Swallow` and the file `src/HolyProject/Swallow.hs`. And we are ready to make our own real world unit test. We will first test the module `HolyProject.GithubAPI`. Let's create a file `test/HolyProject/GithubAPI/Test.hs` with the following content:

```
module HolyProject.GithubAPI.Test
( githubAPISuite
) where
import Test.Tasty (testGroup, TestTree)
import Test.Tasty.HUnit
import HolyProject.GithubAPI

githubAPISuite :: TestTree
githubAPISuite = testGroup "GithubAPI"
    [ testCase "Yann" $ ioTestEq
            (searchGHUser "Yann.Esposito@gmail.com")
            (Just "\"yogsototh\"")
    , testCase "Jasper" $ ioTestEq
            (searchGHUser "Jasper Van der Jeugt")
            (Just "\"jaspervdj\"")
    ]

-- | Test if some IO action returns some expected value
ioTestEq :: (Eq a, Show a) => IO a -> a -> Assertion
ioTestEq action expected = action >>= assertEqual ""
expected
```

You have to modify your cabal file. More precisely, you have to add `HolyProject.GithubAPI` in the exposed modules of the library secion). You also have to update the `test/Test.hs` file to use `GithubAPI` instead of `Swallow`.

So we have our example of unit testing using IO. We search the github nickname for some people I know and we verify github continue to give the same answer as expected.

Property Testing with SmallCheck and QuickCheck

When it comes to pure code, a very good method is to use QuickCheck and SmallCheck. SmallCheck will verify all cases up to some depth about some property. While QuickCheck will verify some random cases.

As this kind of verification of property is mostly doable on pure code, we will test the `StringUtils` module.

So don't forget to declare `HolyProject.StringUtils` in the exposed modules in the library section of your cabal file. Remove all references to the `Coconut` module.

Modify the `test/Test.hs` to remove all references about `Coconut`. Create a `test/HolyProject/StringUtils/Test.hs` file containing:

```
module HolyProject.StringUtils.Test
( stringUtilsSuite
) where
import              Test.Tasty                      (testGroup,
TestTree)
import              Test.Tasty.SmallCheck           (forAll)
import qualified    Test.Tasty.SmallCheck       as  SC
import qualified    Test.Tasty.QuickCheck       as  QC
import              Test.SmallCheck.Series          (Serial)
import              HolyProject.StringUtils

stringUtilsSuite :: TestTree
stringUtilsSuite = testGroup "StringUtils"
    [ SC.testProperty "SC projectNameFromString idempotent" $
            idempotent projectNameFromString
    , SC.testProperty "SC capitalize idempotent" $
            deeperIdempotent capitalize
    , QC.testProperty "QC projectNameFromString idempotent" $
            idempotent capitalize
    ]

idempotent f = \s -> f s == f (f s)

deeperIdempotent :: (Eq a, Show a, Serial m a) => (a -> a) -> SC.Property
m
deeperIdempotent f = forAll $ SC.changeDepth1 (+1) $ \s -> f s == f (f s)
```

The result is here:

```
All Tests
  StringUtils
    SC projectNameFromString idempotent: OK
      206 tests completed
    SC capitalize idempotent:            OK
      1237 tests completed
    QC projectNameFromString idempotent: FAIL
      *** Failed! Falsifiable (after 19 tests and 5 shrinks):
      "a a"
      Use --quickcheck-replay '18 913813783 2147483380' to
reproduce.
  GithubAPI
    Yann:                                OK
    Jasper:                              OK

1 out of 5 tests failed
```

The test fail, but this is not an error. Our `capitalize` function shouldn't be idempotent. I simply added this test to show what occurs when a test fail. If you want to look more closely to the error you could do this:

```
$ ./interact
GHCi, version 7.6.2: http://www.haskell.org/ghc/   :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> :l src/HolyProject/StringUtils
[1 of 1] Compiling HolyProject.StringUtils ( src/HolyProject/StringUtils.hs,
interpreted )
Ok, modules loaded: HolyProject.StringUtils.
*HolyProject.StringUtils> capitalize "a a"
"AA"
*HolyProject.StringUtils> capitalize (capitalize "a a")
"Aa"
*HolyProject.StringUtils>
```

It is important to use `./interact` instead of `ghci`. Because we need to tell `ghci` how to found the package installed.

Apparently, SmallCheck didn't found any counter example. I don't know how it generates Strings and using deeper search is really long.

Conclusion

Congratulation!

Now you could start programming in Haskell and publish your own cabal package.