Если вы видите что-то необычное, просто сообщите мне.

Скрипты на Хаскеле (пробую писать)

Я, кажется, созрел, чтобы переходить от чтения книжек и статей про Хаскель к попыткам что-то на нём писать самому. Вначале какую-нибудь мелочь. Скрипты, в общем. Поскольку я уже как-то публиковал здесь bash-скрипт rss2lj (кросспост RSS в ЖЖ), то решил в качестве упражнения его переписать и улучшить. Думаю, получилось. В этой заметке расскажу о том, как писал. Ну и о впечатлениях. Скрипт выложен на BitBucket и на Hackage.

Задача состоит из кучи рутинных операций. Я думаю, именно поэтому, будет полезно и мне на будущее, и другим начинающим и пробующим, увидеть, как они выполняются на Хаскеле. В частности, по ходу дела я разобрался как

- обрабатывать аргументы коммандной строки,
- читать и писать файлы,
- использовать регулярные выражения,
- отсылать HTTP-запросы,
- выполнять ввод-вывод в уникоде (UTF-8),
- получать системное время.

Писать буду как начинающий — начинающим. На словах получается довольно долго, но сам код получился гораздо короче, чем эта статья (около 200 строк, считая комментарии, необязательные декларации типов, пустые строки и декларации импорта внешних модулей).

Хотя Хаскель язык компилируемый и строго типизированный, использовать его для таких дел вполне можно. Код получается примерно такой же, если не более, краткий, как на Python, а компилируется даже на лету достаточно быстро. Есть и особенности. Во-первых, вместо беззаботного duck-typing здесь — строгая типизация. Поэтому писать надо

аккуратнее (но и ошибок при исполнении меньше). Однако в Хаскеле эта строгая типизация сделана на основе системы типов Хиндли–Миллнера и, в отличие от C++, под ногами не путается. Во-вторых, чтобы использовать преимущества функционального подхода (например, отложенные вычисления, частичное применение функций) нужно отделять чисто функциональную часть программы от императивных фрагментов. В простейшем случае, это означает необходимость отделить операции ввода-вывода от вычислений (преобразования информации). Переводя на Хаскель: функции ввода-вывода будут иметь монадный тип IO a, остальные же будут чистыми (без IO в типе).

Предварительное описание задачи и подхода

В моём примере можно выделить следущие операции ввода-вывода:

- получение URL из аргументов командной строки,
- чтение содержимого RSS или Atom фида по заданному URL,
- чтение (и потом запись) файла со списком уже обработанных записей,
- чтение файла с настройками доступа к учётной записи ЖЖ,
- получение системного времени,
- коммуникация с ЖЖ по установленному протоколу.

И соответственно следующие преобразования данных:

- извлечение идентификаторов всех записей в фиде,
- отсев уже обработанных записей,
- извлечение заголовков, ссылок и текста оставшихся записей,
- форматирование записей по заданному шаблону,
- разбор файла с настройками.

Для разбора произвольных фидов я велосипед изобретать не стал, а воспользовался библиотекой feed. А для всех коммуникаций по HTTP протоколу использовал библиотеку curl (мне понравился её интерфейс). Обе библиотечки нашёл на Hoogle, а установил с помощью cabal. Из остальных зависимостей: нужен модуль Codec.Binary.UTF8.String (в убунту и дебиан он помещён в пакет libghc6-utf8-string-dev), модуль Text.Regex.Posix (соответственно, пакет libghc6-regex-posix-dev). Потом я сейчас заметил, что использовал urlEncode из Network.HTTP (у меня в ~/.cabal), хотя можно было обойтись пакетным escapeURIString (из Network.URI). То есть одна зависимость могла бы быть попроще.

В отдельный модуль я выделил всё, что касается связи связи с ЖЖ и его протокола (файл LjPost.hs). Собственно всю логику скрипта я поместил в другом файле (Feed2Lj.hs). Вспомогательную утилитку для тестирования модуля LjPost я поместил в RunLjPost.hs. Для использования скрипта она не нужна, я её использовал при его написании.

Модуль отправки сообщений в ЖЖ (LjPost)

Использование библиотеки Curl

Как я уже сказал, для работы по HTTP протоколу я использовал библиотечку curl. Соответственно, помещаю в списке импортов

```
import Network.Curl
```

а основную функцию оформляю так, всё это достаточно «императивно»:

```
postToLj ljuser ljpass subj msg = withCurlDo $ do
  curl <- initialize
  ...
```

Функция withCurlDo должна охватывать все вызовы к curl и отвечает за инициализацию и деинициализацию библиотеки; initialize собственно и позволяет к библиотеке потом

обращаться. Собственно HTTP запрос делается так (запрашиваю аутентификационный токен ЖЖ):

```
  r <- do_curl_ curl ljFlatUrl getChallengeOpts :: IO CurlResponse
```

Т.е. используем do_curl_, чтобы получить данные HTTP-ответа; результат (HTTP-ответ) связываю (<-) с переменной r; аргументы do_curl_ были определены мной ранее, URL ЖЖ-API

```
ljFlatUrl = "www.livejournal.com/interface/flat"
```

и собственно параметры запроса:

```
getChallengeOpts = CurlPostFields ["mode=getchallenge"] : postFlags
postFlags = [CurlPost True]
```

Дальнейшие действия определяются логикой протокола ЖЖ.

Разбор ответа ЖЖ

Во flat-протоколе, ответ сервера выглядит так:

```
ключ_1
значение_1
ключ_2
значение_2
...
```

Нужно, во-первых, проверять значение ключа success, во-вторых извлекать значения других ключей, для начала ключа challenge.

Поскольку здесь никакого ввода-вывода уже нет, эту часть кода вполне можно написать «функционально». Самый простой и универсальный сделать это, мне кажется, разбить тело ответа (respBody) на строчки (lines), преобразовать их в ассоциативный список (list2alist) и поискать в нём нужный ключ (lookup), получив, может быть (монада Maybe), значение:

```
lookupLjKey :: String -> CurlResponse -> Maybe String
lookupLjKey k = ( lookup k . list2alist . lines . respBody )
```

При этом функция преобразования списка в ассоциативный список простая двухстрочная рекурсия:

```
list2alist :: [a] -> [(a,a)]
list2alist (k:v:rest) = (k,v) : list2alist rest
list2alist _ = []
```

Всё, мы написали всё необходимое, чтобы разбирать ответы сервера.

Вспомогательная функция, проверяем, успешен ли был запрос (тогда и только тогда, когда в ответе есть ключ success со значением OK):

```
isSuccess :: CurlResponse -> Bool
isSuccess = (=="OK") . fromMaybe "" . lookupLjKey "success"
```

Мы определили isSuccess композицией трёх функций. lookupLjKey возвращает монаду Maybe String. Функция fromMaybe достаёт из неё строковое значение. Функция сравнения (==) записана в префиксной форме и сравнивает значение со строкой «OK».

Прошу заметить, что вытащить из монады Maybe собственно значение всегда можно с помощью fromJust, но если там ничего нет (Nothing), то будет возбуждена ошибка. Здесь функция fromMaybe возвращает в такой ситуации значение по умолчанию (пустую строку), но в других местах скрипта я часто использую fromJust без проверок (т.е. при отсутствии значения скрипт будет прерываться). В программах посерьёзнее, я думаю, лучше всегда использовать функции maybe или fromMaybe, позволяющие использовать Maybe-значения, указав для них значения по-умолчанию.

Отправка сообщения в ЖЖ

Возвращаемся к функции postToLj и пишем, что если аутентификационный токе был успешно получен (isSuccess r), взять текущее время (timeopts <- currentTimeOpts, об этом ниже), подготовить запрос для публикациии сообщения (let opts = postOpts ...) и отправить. Результатом функции будет ответ на последний выполненный запрос:

```
if (isSuccess r)
  then do
```

```
    let challenge = fromJust $ lookupLjKey "challenge" r
    timeopts <- currentTimeOpts
    let opts = postOpts ljuser ljpass challenge subj msg timeopts
    r <- do_curl_ curl ljFlatUrl opts :: IO CurlResponse
    return r
  else return r
```

Как всегда в Хаскеле, если сказал if — then, говори и else (с тем же типом).

Ещё одно «новичковое» замечание: в блоке do мы связываем переменные с монадным значением с помощью (<-) (это соответствует присваиванию в императивных языках), но определяем переменные чистыми выражениями с помощью (=). Вообще, (=) в Хаскеле почти всегда можно читать как «равно по определению». Как только я это понял — жить стало проще ;-)

Теперь подробности. Чтобы отправить сообщение, нужно сформировать POST-запрос согласно протоколу. В моём примере этим занимается функция

```
postOpts u p c subj msg topts =
  CurlPostFields ("mode=postevent" : (authOpts u p c)
                  ++ ["event=" ++ quoteOpt msg, "subject=" ++ quoteOpt subj,
                    "lineendings=unix", "ver=1"]
                  ++ topts ) : postFlags
```

которая аналогичная getChallengeOpts, только список полей, которые нужно отослать, гораздо больше. И есть некоторые тонкости.

Во-первых, нужно защищать («квотировать») некоторые символы в отсылаемых значениях. Их немного, на помощь приходит определение функции с помощью шаблонов аргумента:

```
quoteOpt ('=':xs) = "%3d" ++ quoteOpt xs
quoteOpt ('&':xs) = "%26" ++ quoteOpt xs
quoteOpt (x:xs) = x : quoteOpt xs
quoteOpt [] = []
```

Одно дело сделано. Во-вторых, нужно по имени пользователя, паролю и аутентификационному токену подготовить все поля запроса, касающиеся аутентификации:

```
authOpts u p c = [ "user=" ++ quoteOpt u, "auth_method=challenge",
  "auth_challenge=" ++ quoteOpt c,
  "auth_response=" ++ quoteOpt (evalResponse c p) ]
```

Собственно ответ на токен рассчитывается в одну строчку: evalResponse c p = smd5 (c ++ (smd5 p)) where smd5 = md5sum . fromString Кроме этого нужно импортировать соответствующие функции преобразования уникодной строки в байт-строку UTF-8 и функцию вычисления MD5-суммы:

```
import Data.ByteString.UTF8 (fromString)
import Data.Digest.OpenSSL.MD5 (md5sum)
```

И в-третьих, нужно заполнить в запросе поля, касающиеся времени публикации (текущего времени). Импортируем:

```
import Data.Time
import System.Locale (defaultTimeLocale)
```

Берём текущее время:

```
currentTime = do
  t <- getCurrentTime
  tz <- getCurrentTimeZone
  return $ utcToLocalTime tz t
```

Заметим, что функция эта связана с вводом-выводом и не является «чистой» (не возвращает одно и то же значение всякий раз). По этой причине я предпочёл не вызывать её из «чистой» postOpts, а передать уже готовый список опций, касающихся времени в postOpts из postToLj. Там, напомню, я писал:

```
timeopts <- currentTimeOpts
a currentTimeOpts определил так:
currentTimeOpts :: IO [String]
currentTimeOpts = do
  t <- currentTime
  let opts = [ "year=%Y", "mon=%m", "day=%d", "hour=%H", "min=%M" ]
  return $ map (flip showTime t) opts
```

Т.е. взял текущее время и подставил его в каждый из списка форматов (ЖЖ хочет в таков виде). Вспомогательная функция преобразования времени в строку по формату выглядит так: showTime = formatTime defaultTimeLocale Эта функция двух (неуказанных) аргументов получена каррированием функции formatTime. В map я меняю местами её аргументы (flip), чтобы формат передавался последним, и «перчу» ещё раз текущим временем.

Всё, у нас уже есть всё необходимое для отправки любых сообщений в любой ЖЖ. Нужно только знать логин и пароль.

Чтение файла конфигурации

Где-то логин и пароль хранить надо, и самое простое, что приходит в голову, поместить его в файле настроек, написанном в виде username=мойлогин password=мойпароль В коде скрипта указываю путь по-умолчанию к этому файлу:

```
ljPassFile = "~/.ljpass"
```

Читаем этот файл и делаем из него знакомый и удобный ассоциативный список:

```
readPassFile f = do
  ljpass <- readFile f
  return $ map (\(f,s) -> (f,tail s)) $ map (break (== '=')) $ lines ljpass
```

Поскольку файл заведомо небольшой, можно использовать простую в обращении readFile. Далее как обычно: режем на строки (lines), каждую строку разбиваем по первому знаку «равно» (map (break (== '='))), правим получившийся ассоциативный список список, откидывая знаки «равно» (λ-функция во втором map). Результат заворачиваем в IO-монаду (return) как того требует тип функции.

Почти готово. Для пущего удобства сделаем себе раскрытие тильды в пути к файлу: expandhome ('~':'/':p) = do h <- getHomeDirectory ; return (h ++ "/" ++ p) expandhome p = return p и собственно функцию, которая, будет нам давать значение любого ключа из файла конфигурации:

```
readLjSetting key = do
  passfile <- expandhome ljPassFile
  s <- readPassFile passfile
  return (lookup key s)
```

В этот раз нам надо добавить ещё две декларации импорта:

```
import IO
import System.Directory (getHomeDirectory)
```

Последний штрих: в объявлении модуля перечисляем экспортируемые вовне функции, а вспомогательные замалчиваем:

```
module LjPost (readLjSetting, postToLj, isSuccess, lookupLjKey, putLjKey) where
```

Наш модуль готов к использованию. Он позволяет нам задавать настройки доступа в файле конфигурации, понимает ЖЖ-протокол, поддерживает challenge-response аутентификацию и позволяет публиковать в ЖЖ сообщения. Меньше 100 строк кода, если не считать комментарии.

Обработка RSS/Atom фида (Feed2Lj)

Переходим к заключительной части рассказа. Скрипт Feed2Lj.hs берёт URL фида из командной строки, настройки ЖЖ из файла с настройками (для него там добавляем третью настройку, имя файла со списком уже обработанных записей), скачивает фид и отсеивает уже обработанные, необработанные преобразует в plain-text, форматирует по образцу и отсылает в ЖЖ, обновляя список обработанных записей. Теперь подробно.

Получение аргументов командной строки

Получить список аргументов просто, его даёт функция getArgs из System.Environment. У нас аргумент один, адрес фида, поэтому может сразу связать нужную переменную (url) с первым элементом списка, проигнорировав остальные:

```
url:_ <- getArgs
```

Такое связывание по шаблону мне кажется очень элегантным приёмом.

Скачивание фида

На помощь опять приходит библиотечка curl. И опять связывание по шаблону, чтобы взять только интересующую нас часть результата:

```
(_,rawfeed) <- curlGetString url []
```

Используем модуль LjPost для чтения настроек

В общем-то вся работа уже сделана, осталось только использовать функцию readLjSetting. У неё тип [Char] -> IO (Maybe [Char]), т.е. по строке она возвращает IO-монаду, внутри которой, может быть строка (значения настройки найдено и считано), а может и не быть (настройка не найдена). Поскольку у нас тут сразу две монады (IO и Maybe), одна в другой, то, чтобы вытащить просто (Just) значение, я поступаю так:

```
ljuser <- return fromJust `ap` readLjSetting "username"
```

т.е. функцию fromJust применяю внутри монады IO (ap из Control.Monad). Аналогично с остальными значениями из файла настроек. Кажется немного громоздно с непривычки, но не так уж сложно потом. Уверен, можно написать короче.

Чтение списка обработанных записей

Мой старый bash-скрипт писал ID записей в файл, одно на строчку, поэтому новый скрипт использует тот же формат (и тот же файл). Читаем файл и преобразуем в список строк:

```
sent_ids <- (return . lines) =<< readFile sentfile
```

Здесь, чтобы не вводить временную переменную, я явно указал функцию связывания вычислений (=<<). return требуется типом (=<<). Результат эквивалентен записи

```
tmp <- readFile sentFile
let sent_ids = lines tmp
```

Отсеиваем обработанные записи

Для начала разберём содержимое фида и подготовим список всех записей. Благодаря библиотечке feed это легко:

```
  let feed = fromJust $ parseFeedString rawfeed
  let items = feedItems feed
```

Ну а отсеять уже обработанные можно с помощью filter:

```
  let newitems = reverse $ filter (isNotSent sent_ids) items
```

Функция-предикат получилась за счёт каррирования isNotSent:

```
isNotSent sent i = ((snd . fromJust . getItemId) i) `notElem` sent
```

Буквально: взять просто ID элемента (возможна ошибка), проверить, что не входит в список sent. Сразу подготовим список ID подлежащих обработке записей:

```
let new_ids = map ( snd . fromJust . getItemId) newitems
```

Отправляем запись в ЖЖ

Тупо используем уже написанный модуль LjPost. Если даны имя пользователя, пароль, шаблон записи для отправки и собственно запись:

```
postItem u p t i = do
  let message = renderItem t i
  let subj = fromJust $ getItemTitle i
  r <- postToLj u p subj message
  if isSuccess r
    then putLjKey "url" r
    else putLjKey "errmsg" r
```

Стоп-стоп-стоп! Какой ещё такой шаблон записи (t) и что делает renderItem? Объясняю: отослать запись нам надо в HTML-е, и хорошо бы можно было менять формат записи, не переделывая весь код. В общем, renderItem — это маленькая template engine, t — её шаблон. Я её опишу в следующих разделах статьи.

Вызываем из main для каждой записи из списка необработанных:

```
  let t = encodeString "<p>%text%</p><p>( <a href=\"%link%\" title=\"%title%\">дальше</a> )</p>"
  mapM_ (postItem ljuser ljpass t) newitems
```

Здесь мы формируем список IO-действий и их последовательно исполняем (mapM_). То есть последовательно отсылаем все записи из нашего списка. Обратим ещё внимание на encodeString из Codec.Binary.UTF8.String, которая кодирует строку в UTF-8.

Форматирование по шаблону (маленькая template engine)

Напишем нашу маленькую функцию форматирования по шаблону. Пусть, допустим, все параметры шаблона будут представлены как «%параметр%», а спецсимвол «%» будет представлен в шаблоне как «%%». Параметры будет передавать ассоциативным списком, а шаблон — строчкой. На выходе — строчка с подставленными в шаблон параметрами:

```
renderTemplate _ [] = []
renderTemplate alist s =
  let (b,t,a) = s =~ "%[a-z0-9]*%" :: (String,String,String)
      tagval t
        | t == "%%" = Just "%"
        | otherwise = let inner = take (length t - 2) $ drop 1 t
                      in  lookup inner alist
      val = tagval t
  in if isJust val
       then b ++ (fromJust val) ++ renderTemplate alist a
       else b ++ t ++ renderTemplate alist a
```

Функция форматирования сообщения по шаблону готова. В ней мы последовательно «раскусываем» шаблон с помощью регулярных выражений на «текст-до», «тег» и «текст-после». Подставляем на место «тега» (t) значение соответствующего параметра, если есть, или буквальный «%», если тэг пустой. Продолжаем, пока не кончится шаблон.

О регулярных выражениях. Включаем импортом

```
import Text.Regex.Posix ((=~))
```

После этого можем в любой строчке искать регулярное выражение: строка =~ выражение :: возвращаемый тип Регулярные выражения ведут себя по-разному в зависимости от возвращаемого типа. Мне пока что пригождаются больше всего два из них: Bool для проверки соотвествия строки выражению и тройной кортеж (String,String,String), разрезающий строчку на три части.

Функция форматирования по шаблону готова. Она просто работает со строками (шаблонами) и ассоциативными списками (словарями). А где же обещанная renderItem?

Форматируем запись по шаблону

Итак, renderItem должна получать шаблон и запись из фида, а возвращать строчку. Всё, что делает эта функция — просто достаёт нужные параметры записи, помещает их в ассоциативный список и вызывает функцию форматирования по шаблону renderTemplate. В виде кода это выглядит гораздо понятнее:

```
renderItem :: String -> Item -> String
renderItem t i =
  let title = ( fromJust . getItemTitle ) i
      link = ( fromJust . getItemLink ) i
      summary = ( takeSentences 5 . eatTags . fromJust . getItemSummary) i
      tags = zip [ "title","link","text" ]
                 [ title, urlEncode link,summary ]
  in renderTemplate tags t
```

Нетривиальна здесь только функция подготовки текста сообщения (summary).

Поскольку я весь текст пересылать не хочу, а хочу только первые несколько предложений, то я вначале преобразую HTML в простой текст (в котором уже нет HTML-тэгов), а затем просто берую первые пять предложений. Таким образом, мне не нужно заботиться о предолжения будут гарантировано законченными.

Функция eatTags использует тот же приём рекурсивного раскусывания строки с помощью регулярных выражений, что и renderTemplate:

```
eatTags [] = []
eatTags s =
  let (b,t,a) = s =~ "</?[^>]*/?>" :: (String,String,String)
  in b ++ eatTags a
```

Все HTML и XHTML теги должны быть этой функцией вырезаны.

Упражнение: изменить функцию так, чтобы тег выразался не бесследно, а заменялся содержимым его аттрибута alt.

Теперь осталось лишь взять первые n предложений. Возьмём вначале одно:

```
takeSentence s =
  let ends = ".?!;"
      (first,rest) = break (`elem` ends) s
  in if not (null rest)
     then (first ++ [head rest],tail rest)
     else (first,[])
```

Тут я обошёлся без регулярных выражений, просто задав список разделителей (ends) и раскусывая строку по символу из их числа (break (`elem` ends)). Напоследок присоединяю разделитель, если он есть, к «откушенному» предложению (break прикрепляет его к «остатку»).

Осталось лишь взять первые n штук:

```
takeSentences n s
  | n > 0     = let (s',r) = takeSentence s
                in s' ++ takeSentences (n-1) r
  | otherwise = ""
```

Теперь любая запись может быть представлена так, как мы захотим. Обновляем список обработанных записей Записи получены, отобраны, отформатированы, отправлены. Осталось только обновить список обработанных. Вначале сохраним предыдущую версию файла (переименованием), а потом запишем на его место новый список:

```
  renameFile sentfile (sentfile ++ "~")
  writeFile sentfile $ unlines (sent_ids ++ new_ids)
```

Здесь использована функция renameFile из System.Directory.

Заключение

Вот вроде и всё. Можно вызывать получившийся скрипт:

```
$ runhaskell Feed2Lj.hs URL-вашего-фида
```

Пробовал пока только с GHC, но, думаю, и с Hugs должно работать. Я, кстати, осознал, что у интерпретатора Hugs есть важное преимущество перед GHC: установка GHC тянет около 100 МБ, а Hugs — всего порядка 10 МБ. Так что как разберусь с Hugs, буду стараться проверять свои скрипты и на нём.

В целом впечатления от опыта «написать на Хаскеле» очень положительные. Во-первых, очень приятно, когда удаётся написать полезную функцию в одну-две строчки. Во-вторых, интересно думать о программе иначе, писать более декларативно. В третьих, очень приятно, когда раз — и работает! (Ну это с любым языком). В четвёртых, мне нравится «математичный» синтаксис Хаскеля, он, по-моему, очень выразителен. Поначалу, пока не знакомо, конечно долго и непривычно, но когда входишь во вкус, получается быстрее и легче.

Кроме, понятно, гугла, большой подмогой является Hoogle. Сообщения GHC довольно подробные и понятные (разбирать ошибки C++-компиляторов про шаблоны гораздо труднее). Радует, что уже сейчас коллекция библиотек весьма богата (кажется, сопоставима с набором библиотек Python в то время, когда я с ним впервые познакомился). С уникодом, опять же, никаких проблем.

Есть и всякие «но»: но в коде других людей мне ещё далеко не всё понятно, но пихать ввод-вывод в любую точку кода в Хаскеле неудобно и не нужно (сделано намеренно, для отладки служит trace из Debug.Trace), но представить порядок ленивых вычислений не всегда легко, но документированы библиотеки в Hackage весьма лаконично (строго, по делу, но не так доходчиво и очевидно для новичков, как, например в Python), но cabal до сих пор нет ни в Debian, ни в Ubuntu.

Но всё равно, мне понравилось. Буду рад замечаниям и вопросам. Уверен, что-то можно было написать лучше (короче, понятнее и выразительнее). Что-то, наверное, забыл объяснить.

Revision #2
Created 6 April 2024 08:04:50 by gasick
Updated 6 April 2024 08:13:00 by gasick