Template Haskell

Наводящие соображения, препроцессинг

Стандартные возможности

При обработке Haskell-кода у нас есть возможность в некоторой степени управлять поведением компилятора ghc с помощью прагм и специальных вставок. Например, прагма в заголовке файла

```
{-# OPTIONS_GHC <flags>   #-}
```

даёт возможность управлять флагами компилятора ghc из кода (об этом ниже подробнее). Или прагма `INLINE`

```
mySimpleFunction :: ...
mySimpleFunction = ...

{-# INLINE mySimpleFunction #-}
```

даёт явное указание для встраивания функций.

Также есть возможность взаимодействия с препроцессором Си. Например,

```
{-# LANGUAGE CPP #-}

main :: IO ()
main = do
  print "hello"
#ifdef APP_DEBUG
  print "test"
#endif
```

при комплияции (или запуске `runghc`) мы можем дополнительно указывать флаг:

```
>runghc -DAPP_DEBUG testCPP.hs
"hello"
"test"
```

Собственный препроцессор

Для Haskell-кода мы можем организовать свой препроцессор.

Допустим, мы хотим захардкодить время компиляции программы. Или её версию, или ещё что-нибудь — не суть важно. Напишем небольшой скрипт на Perl:

```
#!/usr/bin/env perl

$localtime = localtime();
($fname, $input, $output) = @ARGV;

# print "working for: ".$fname.' '.$input.' '.$output;

open $fin,  '<', $input  or die $!;
open $fout, '>', $output or die $!;
```

```
while(<$fin>) {
    s/__LOCALTIME__/$localtime/ge;
    print $fout $_;
}

close $fin;
close $fout;
```

Сохраним его где-нибудь в `$PATH` с именем `localtime_replace` и скажем `chmod u+x`. Как видите, скрипт принимает три параметра — имя оригинального файла, имя входного файла и имя выходного файла. Скрипт открывает входной файл, заменяем в нём строки `__LOCALTIME__` на текущее время и сохраняет результат в выходном файле. Два первых аргумента обычно будут совпадать.

Затем создадим файл `test.hs` следующего содержания:

```
{-# OPTIONS_GHC -F -pgmF localtime_replace #-}

main = putStrLn "Localtime: __LOCALTIME__"
```

Директива `OPTIONS_GHC` позволяет указать ключи, которые будут переданы GHC при компиляции программы. Описание всех ключей можно найти в Glasgow Haskell Compiler User's Guide. Интересующие нас в данный момент ключи описаны в разделе 5.11.4. Options affecting a Haskell pre-processor. Флаг `-F` говорит, что нужно использовать препроцессор. При вызове препроцессор `cmd` получает по крайней мере три аргумента в своей командной строке: первый аргумент — это имя исходного файла, второй — имя файла, содержащего входные данные (обычно, 1-й и 2-й аргумент совпадают), а третий — имя файла, в который `cmd` должен записать свой вывод. Ключ `-pgmF` указывает программу, которую следует использовать в качестве препроцессора. Также имеется ключ `-optF`, позволяющий передать препроцессору дополнительные аргументы. Эти аргументы будут переданы после имени выходного файла.

Проверяем:

```
$ ghci test.hs
ghci> main
Localtime: Thu Aug 15 22:50:04 2013
ghci> :q
```

К сожалению, под Windows данный приём не работает. Необходимо создавать «батник»-прокладку, файл `runme.cmd`:

```
perl localtime_replace
```

Сохраняем теперь все три файла в текущей директории. А в файле `test.hs` указываем:

```
{-# OPTIONS_GHC  -F -pgmF runme.cmd #-}

main = putStrLn "Localtime: __LOCALTIME__"
```

Запускаем:

```
runghc test.hs
```

Получаем:

```
perl localtime_replace "test.hs" "test.hs"
 "C:\Users\me\AppData\Local\Temp\ghc5744_0\ghc_1.hspp"
Localtime: Sun Mar 17 15:51:38 2019
```

Препроцессинг открывает перед нами много новых возможностей. Например, он активно используется в шаблонизаторе HSP, позволяющем использовать синтаксис XML прямо в исходном коде на Haskell. Однако, в силу понятных причин, использовать препроцессинг нужно с большой осторожностью.

Прогоняем код на Haskell через собственный препроцессор

What is the difference between the first two arguments given to a Haskell custom pre-processor when invoked using -F -pgmF?

Решение без препроцессинга…

Давайте посмотрим, как теперь справится с той же задачей Template Haskell:

```
module LocaltimeTemplate where

import Language.Haskell.TH
import Data.Time

localtimeTemplate :: Q Exp
localtimeTemplate = do
  t <- runIO getCurrentTime
  return $ LitE ( StringL (show t) )
```

Определённый здесь шаблон localtimeTemplate имеет тип Q Exp. Тип Exp определён в модуле Language.Haskell.TH.Syntax и представляет собой абстрактное синтаксическое дерево (AST) кода на Haskell. Этот тип оборачивается в монаду цитирования Q, позволяющей генерировать уникальные имена переменных и функций в теле шаблона. Для вызова функций с побочными эффектами из монады Q предназначена функция runIO:

```
runIO :: IO a -> Q a
```

В данном примере с помощью runIO и getCurrentTime мы узнаем текущее время и генерируем AST, соответствующий строке с этим временем.

Рассмотрим программу, использующую этот шаблон:

```
{-# LANGUAGE TemplateHaskell #-}

import LocaltimeTemplate

main = putStrLn $
  "Localtime: " ++ $(localtimeTemplate)
```

Смотрите, что происходит. Во время компиляции программы будет выполнен шаблон localtimeTemplate. Он вернёт AST, представляющий строку с текущим временем, т.е. временем компиляции программы. С помощью вклейки (кода $(localtimeTemplate), кстати, скобочки здесь не обязательны) этот AST будет подставлен на место шаблона. В результате мы словно скомпилируем следующую программу:

```
main = putStrLn $ "Localtime: " ++
   "2013-09-21 07:33:13.101347 UTC"
```

Тут нужно обратить внимание на несколько тонких моментов. Во-первых, шаблоны не могут вклеиваться в тех же модулях, в которых они объявляются. Дело в том, что при компиляции модуля, вклеивающего шаблон, шаблон должен быть уже скомпилирован, чтобы его можно было выполнить. Во-вторых, между символом `$` и скобочками или именем шаблона не должно быть пробела. Иначе во время компиляции мы получим ошибку `parse error on input `$'`, поскольку GHC ошибочно примет доллар за функцию `($) :: (a -> b) -> a -> b`.

Мои первые эксперименты с Template Haskell

GHC: insert compilation date

Введение в TH

> `Template Haskell` (далее TH) — это расширение языка Haskell предназначенное для мета-программирования. Оно даёт возможность алгоритмического построения программы на стадии компиляции. Это позволяет разработчику использовать различные техники программирования, недоступные в самом Haskell'е, такие как макро-подобные расширения, направляемые пользователем оптимизации (например inlining), обобщённое программирование (polytypic programming), генерация вспомогательных структур данных и функций из имеющихся.

Примеры

К примеру, код

```
yell file line =
  fail ($(printf "Error in file %s line %d") file line)
```

может быть преобразован с помощью TH в

```
yell file line =
  fail ((\x1 x2 -> "Error in file " ++ x1 ++
    " line " ++ show x2) file line)
```

Другой пример, код

```
data T = A Int String | B Integer | C
$(deriveShow ''T)
```

может быть преобразован в

```
data T = A Int String | B Integer | C

instance Show T
    show (A x1 x2) = "A " ++ show x1 ++ " "
      ++ show x2
    show (B x1)    = "B " ++ show x1
    show C         = "C"
```

Bulat Ziganshin. Template Haskell reference.

Введение в Template Haskell. Часть 1. Необходимый минимум.

Мотивация

По мнению Марка Карпова в качестве мотивационной части можно рассмотреть следующие причины использования `TH`:

- Автоматическое получение экземпляров классов типов по-прежнему является, пожалуй, наиболее распространённым вариантом использования `TH`. Поэтому `TH` предпочтительный метод автоматического вывода экземпляров в таких библиотеках, как `Aeson` и `Lens`.
- Создание средствами `TH` языков DSL, которые интегрированы в системы, встроенные в Haskell. Примерами таких DSL являются язык описания моделей, используемый в пакете persistent для сериализации различных данных, и разнообразные другие мини-языки, используемые, напр., в веб-среде: `yesod.`
- Конструкция уточнения значений особых «улучшенных типов» во время компиляции, которая превращает неверные входные данные в ошибки компиляции.
- Загрузка во время компиляции и обработка данных из внешних файлов, что иногда очень полезно. Несмотря на то, что это требует запуска ввода-вывода во время компиляции, это вполне оправданный случай использования в целом опасной функции.

Недостатков тоже хватает, основными являются отсутствие прозрачности и ясности кода («чёрная магия») — таким образом, возникает острая нужда в подробной документации. И возникает много несвойственных Haskell ограничений, например, вплоть до порядка определений в файлах, где используется `TH`.

Представление Haskell-кода в шаблонах

В `Template Haskell` фрагменты Haskell-кода представляются с помощью обычных алгебраических типов данных. Эти типы построены в соответствии с синтаксисом Haskell и представляют абстрактное синтаксическое дерево (AST — abstract syntax tree) конкретного кода. Есть тип `Exp` для представления выражений, `Pat` — для образцов, `Lit` — для литералов, `Dec` — для объявлений, `Type` — для типов и т.д. Определения всех этих типов можно посмотреть в документации модуля Language.Haskell.TH.Syntax. Они взаимосвязаны в соответствии с правилами синтаксиса Haskell, так что, используя их, можно сконструировать значения, представляющие любые фрагменты Haskell-кода. Вот несколько простых примеров:

- `varx = VarE (mkName "x")` представляет выражение `x`, т.е. простую переменную `x`
- `patx = VarP (mkName "x")` представляет образец `x`, т.е. ту же переменную `x`, использованную в образце
- `str = LitE (StringL "str")` представляет выражение-константу `"str"`
- `tuple = TupE [varx, str]` представляет выражение-пару (кортеж) `(x,"str")`

- `LamE [patx] tuple` представляет лямбда-выражение `\x -> (x,"str")`

  Чтобы упросить нам жизнь, имена всех конструкторов типа `Exp` оканчиваются на `E`, имена конструкторов типа `Pat` — на `P` и т.д. Функция `mkName`, использованная выше, создаёт значение типа `Name` (представляющего идентификаторы) из обычной строки (**`String`**), с её содержанием в качестве имени.

  Итак, чтобы создать Haskell-код, `TH`-функция должна просто сконструировать и вернуть значение типа `Exp` (можно ещё `Dec`, `Pat` или `Type`), которое является представлением для данного фрагмента кода. На самом деле, вам не нужно досконально изучать устройство этих типов, чтобы знать, как представить в них нужный Haskell-код, — далее, будет рассказано, как можно получить TH-представление конкретного фрагмента Haskell-кода.

…

Значение типа `Q a` нужны только для того, чтобы каким-либо образом использовать значение `a` в программе на Haskell. `a` может быть чем угодно в промежуточных монадических выражениях, но когда мы собираемся вставить сгенерированный код в исходный файл Haskell, есть только четыре варианта:

- Декларация `Dec`, которая включает в себя такие вещи верхнего уровня, как определения функций и типов данных. Фактически, мы хотели бы иметь возможность генерировать несколько объявлений одновременно, поэтому тип, который фактически используется (и ожидается интерполяционным механизмом): `[Dec]`.
- Выражение `Exp`, такое как `x+1` или `\x -> x+1`. Это, вероятно, самая распространённое использование.
- Тип `Type`, например **`Int`** или **`Maybe Int`**, или просто **`Maybe`**. Тип не обязательно должен быть полным (т.е. может иметь любой вид), так что это может быть почти всё, что можно встретить на уровне типа.
- Паттерн `Pat`, который мы используем для сопоставления с образцом.

Используя типы данных, неспешно, «через боль и страдание», мы действительно можем построить выражение :)

Использование шаблонов

Тем не менее шаблоны не являются чистыми функциями, возвращающими простое значение типа `Exp`. Вместо этого они являются вычислениями в специальной монаде `Q` (называемой монадой цитирования — “quotation monad”), которая позволяет автоматически генерировать уникальные имена для переменных с помощью монадической функции

```
newName :: String -> Q Name
```

При каждом её вызове генерируется новое уникальное имя с данным префиксом. Это имя может быть использовано как часть образца (с помощью конструктора `VarP :: Name -> Pat`) или выражения (`VarE :: Name -> Exp`).

Давайте напишем простой пример-шаблон `tupleReplicate`, который, будучи использован следующим образом: `$(tupleReplicate n) x`, вернёт n-местный кортеж с элементом `x` на всех позициях (аналогично функции **`replicate`** для списков). Обратите вни-

мание на то, что `n` — аргумент шаблона, а `x` — аргумент сгенерированной анонимной функции (лямбда-выражения).

Ниже код модуля, содержащего определение этого шаблона (модуль Language.Haskell.TH предоставляет весь инструментарий, необходимый для работы с `TH`):

```
module TupleReplicate where

import Language.Haskell.TH

tupleReplicate :: Int -> Q Exp
tupleReplicate n =
    do id <- newName "x"
       return $ LamE [VarP id]
                     (TupE $ replicate n $ VarE id)
```

К примеру вызов `tupleReplicate 3` вернёт значение `Exp` эквивалентное Haskell-выражению `(\x -> (x,x,x))`.

Мотивация существования монады `Q`

Генерация кода требует, чтобы нам были доступны определённые свойства и возможности:

- Возможность генерировать новые уникальные имена, которые не могут быть захвачены.
- Возможность извлекать информацию о программной сущности по её названию. Обычно мы хотим знать о функциях и типах, но есть также способы узнать о модуле, получить коллекцию экземпляров класса определённого типа и т.д.
- Возможность поставить и получить какое-то пользовательское состояние, которое затем используется во всём `TH`-коде в одном модуле.
- Возможность запуска ввода-вывода во время компиляции, поэтому мы можем, например, читать что-то из файла.

Эти функции обычно достигаются с помощью монад в Haskell, и поэтому неудивительно, что есть специальная монада, называемая `Q` (сокращение от «цитата»), которая содержит все функции, предоставляемые `TH`.

Как сгенерированный TH-код вставляется в программу

Вклейка (*splice*) записывается в виде `$somename`, где `somename` — идентификатор, или в виде `$(...)`, где троеточие подразумевает соответствующее выражение. Важно, чтобы не было пробела между символом `$` и идентификатором или скобками. Такое использование `$` переопределяет значение этого символа в качестве инфиксного оператора, так же как квалифицированное имя `M.x` переопределяет значение оператора композиции функций `.`. Если нужен именно оператор, то символ нужно окружить пробелами.

Вклейка может появляться в

- выражении; вклеиваемое выражение должно иметь тип `Q Exp`

- объявлениях верхнего уровня; вклеиваемое выражение должно иметь тип `Q [Dec]`. Объявления, сгенерированные вклейкой, имеют доступ только к тем идентификаторам, которые объявлены в коде раньше (что нетипично для обычных программ на Haskell'е, в которых порядок объявлений не играет роли);
- типе; вклеиваемое выражение должно иметь тип `Q Type`.

Также следует знать, что при запуске GHC нужно использовать флаг `-XTemplateHakell`, чтобы разрешить специальный синтаксис TH; или можно включить в исходник директиву

```
{-# LANGUAGE TemplateHaskell #-}
```

Вы можете использовать шаблон только извне, т.е. нельзя определить в одном модуле шаблон и тут же использовать его (вклеить) (это связанно с тем, что шаблон ещё не скомпилирован к этому моменту).

Пример модуля, который использует шаблон `tupleReplicate`:

```
{-# LANGUAGE TemplateHaskell #-}

module Test where

import TupleReplicate

main = do print ($(tupleReplicate 2) 1)
  -- напечатает (1,1)
          print ($(tupleReplicate 4) "x")
  -- напечатает ("x","x","x","x")
```

Цитирующие скобки

Построение значений `Exp`, представляющих абстрактное синтаксическое дерево — трудоёмкая и скучная работа. Но к счастью, в `Template Haskell` есть цитирующие скобки (ещё называемые «*оксфордские скобки*»), которые преобразуют конкретный Haskell-код в структуру, представляющую его.

Они бывают четырёх типов:

- `[e| ... |]` или `[| ... |]` для выражений (`:: Q Exp`)
- `[d| ... |]` для объявлений (`:: Q [Dec]`)
- `[t| ... |]` для типов (`:: Q Type`)
- `[p| ... |]` для образцов (паттернов) (`:: Q Pat`)

Соответственно внутри скобок должно быть синтаксически корректное выражение/объявление/тип/образец.

Например, цитата `[| \_ -> 0 |]` представляет собой структуру

```
(return $ LamE [WildP] (LitE (IntegerL 0)))
```

Цитата имеет тип `Q Exp` (а не просто `Exp`), так что она должна быть вычислена внутри монады цитирования, что позволяет `Template Haskell` заменить все идентификаторы, появляющиеся внутри цитаты, на уникальные, сгенерированные с помощью `newName`.

Например, цитата `[| \x -> x |]` будет преобразована в такой код:

```
do id <- newName "x"
   return $ LamE [VarP id] (VarE id)
```

Это, как и пример выше можно узнать следующим образом (First stab at Template Haskell): запускаем ghci с расширением `TH`:

```
> ghci -XTemplateHaskell
ghci> :m + Language.Haskell.TH
```

И там смотрим:

```
ghci> runQ [| \x -> 1 |]
LamE [VarP x_0] (LitE (IntegerL 1))
```

Тип последнего вычисленного выражения (`it` в ghci):

```
ghci> :t it
it :: Exp
```

Можно делать такие трюки:

```
ghci> runQ [| 1 + 2 |]
InfixE (Just (LitE (IntegerL 1))) (VarE GHC.Num.+)
 (Just (LitE (IntegerL 2)))
ghci> $(return it)
3
```

и даже вложения:

```
ghci> runQ [| 1 + $([| 2 |]) |]
InfixE (Just (LitE (IntegerL 1))) (VarE GHC.Num.+)
 (Just (LitE (IntegerL 2)))
```

Как мы видим в последнем примере, внутри цитирующих скобок мы можем использовать вклейку (сплайсинг), так что получается, что `TH` выступает в роли макро-препроцессора, обрабатывающего часть кода, написанного явно, и часть кода — сгенерированного. Например, цитата `[| 1 + $(f x) |]` вычислит `(f x)`, которая должна иметь тип `Q Exp`, выражение (структуру типа `Exp`), получившееся в результате представит в виде обычного Haskell-кода и подставит (вклеит) его на место `$(f x)`, а потом продолжит цитирование — преобразование получившегося кода в структуру, представляющую его. Благодаря автоматическому переименовыванию (собственно, для этого всё и делается внутри монады `Q`), внутри цитаты не будет конфликтов имён локальных переменных между разными вклейками одного и того же кода. Следующее определение хорошо это демонстрирует:

```
summ :: Int -> Q Exp
summ n = summ' n [| 0 |]

summ' :: Int -> Q Exp -> Q Exp
summ' 0 code = code
```

```
summ' n code = [| \x ->
     $(summ' (n-1) [| $code + x |])
  |]
```

Этот шаблон генерирует лямбда-выражение с `n` параметрами, которое их суммирует. Например, `$(summ 3)` преобразуется в

```
(\x1 -> \x2 -> \x3 -> 0 + x1 + x2 + x3)
```

Обратите внимание, на то, что в сгенерированном коде используются разные идентификаторы для аргументов вложенных лямбда-выражений, хотя в шаблоне имя одно: `[| \x -> ... |]`. Как видно на этом примере, вложенность цитат и вклеек может быть любой, но важно чтобы они чередовались — нельзя цитировать внутри цитаты и вклеивать внутри вклейки.

Вклейка и цитирование — это взаимно обратные операции: одна преобразует структуру `Exp` в Haskell-код, а другая — Haskell-код в структуру `Exp`, поэтому они взаимно аннигилируются:

`$( [|` выражение `|] )` ≡ выражение

`[| $(` структура `) |]` ≡ структура

Это позволяет, разрабатывая TH-программы, мыслить только в терминах генерируемого Haskell-кода и не думать о внутренних структурах, представляющих его синтаксис.

(тем не менее, есть ряд правил квазицитирования, которые при этом необходимо соблюдать!)

Рассмотрим для примера вычисление вклейки `$(summ 3)`. Просто будем заменять использование шаблона на его определение:

```
$(summ 3)
$(summ' 3 [| 0 |])
$([| \x -> $(summ' (3-1) [| $([| 0 |]) + x |]) |])
```

Теперь мы можем убрать лишние скобки `$([| ... |])`, заменяя по ходу дела `x`, на уникальный идентификатор:

```
  \x1 -> $(summ' (3-1) [| 0 + x1 |])
```

Снова подставляем определение `summ'`:

```
\x1 -> $([| \x -> $(summ' (2-1) [| $([| 0 + x1 |]) + x |]) |])
```

Далее будем повторять последние два шага, пока это возможно:

```
\x1 -> \x2 -> $(summ' (2-1) [| 0 + x1 + x2 |])
\x1 -> \x2 -> $([| \x -> $(summ' (1-1) [| $([| 0 + x1 + x2 |]) + x
|]) |])
\x1 -> \x2 -> \x3 -> $(summ' (1-1) [| 0 + x1 + x2 + x3 |])
\x1 -> \x2 -> \x3 -> $([| 0 + x1 + x2 + x3 |])
\x1 -> \x2 -> \x3 -> 0 + x1 + x2 + x3
```

Интересно, что это в этом определении левая часть лямбда-выражения `(\x1 -> \x2 -> ...)` рекурсивно строится по ходу разворачивания рекурсии, а правая часть `(0 + x1 + ...)`

в то же время аккумулируется в оставшейся части шаблона. Такая же техника используется в примере шаблона `printf` далее.

Для использования создаём два файла: `Summ.hs` с уже известным содержанием:

```
{-# LANGUAGE TemplateHaskell #-}

module Summ where

import Language.Haskell.TH

summ :: Int -> Q Exp
summ n = summ' n [| 0 |]

summ' :: Int -> Q Exp -> Q Exp
summ' 0 code = code
summ' n code = [| \x ->
     $(summ' (n-1) [| $code + x |])
  |]
```

и `UseSumm.hs`, который использует введённую нами функцию:

```
{-# LANGUAGE TemplateHaskell #-}

import Summ

main = putStrLn $
  "Summ: " ++ (show $ $(summ 3) 1 2 3)
```

Пример: printf

Теперь мы разберём определение шаблона `printf`, который упоминался в первой части статьи. Далее приводится код с пояснениями, а также модуль `Main`, использующий его. Скомпилировать его можно командой `ghc -XTemplateHaskell --make Main.hs`

`Main.hs`:

```
{-# LANGUAGE TemplateHaskell #-}

module Main where

-- Импортируем наш шаблон printf
import Printf (printf)

-- Оператор $( ... ) развернёт шаблон с данным параметром
-- в обычный Haskell-код во время компиляции и
-- вклеит его на то же место – как аргумент putStrLn

main = putStrLn ( $(printf "Error in file %s line %d: %s") "io.cpp"
 325 "printer not found" )
```

`Printf.hs`:

```
{-# LANGUAGE TemplateHaskell #-}

module Printf where

-- Импортируем инструментарий Template Haskell
import Language.Haskell.TH

-- Описание строки форматирования
data Format = D         -- представляет  S – представляет  L String –
представляет остальной текст (L от Literally)

-- Парсер строки форматирования – преобразовывает её в структуру Format
parse :: String -> String -> [Format]
parse          ""  rest  = [L rest]
parse ('%':'d':xs) rest  =  L rest : D : parse xs ""
parse ('%':'s':xs) rest  =  L rest : S : parse xs ""
parse       (x:xs) rest  =  parse xs (rest++[x])

-- Генератор Haskell-кода, подставляющий вместо элементов
-- форматирования соответствующие лямбда-выражения
gen :: [Format] -> ExpQ -> ExpQ
gen        []  code = code
gen (D   : xs) code = [| \x -> $(gen xs [| $code ++ show x |]) |]
gen (S   : xs) code = [| \x -> $(gen xs [| $code ++ x |]) |]
gen (L s : xs) code = gen xs [| $code ++ s |]

-- Шаблон, который берёт на вход строку форматирования
-- парсит её и генерирует соответствующий код
printf :: String -> ExpQ
printf s = gen (parse s "") [| "" |]
```

И есть альтернативное, более простое решение.

Создание собственных квазиквоттеров

Помимо перечисленных выше 4-х типов оксфордских скобок, возможно создание собственных вариантов. Мы с таким сталкивались, когда, например, изучали работу регулярных выражений (в лекции-10):

```
...
import Text.Regex.PCRE.Heavy

str = "We are testing this address: mister.twister@gmail.com,\
\ and this one: hacker@yahoo.com, too."

str2 = gsub [re|@[A-z]+\.[A-z]+|] "@mail.ru" str
```

Как можно делать такие обработчики самостоятельно?

Разрозненная документация немного проливает свет на это:

1. wiki.haskell: Quasiquotation

2. Language.Haskell.TH.Quote
3. 6.13.7. Template Haskell Quasi-quotation
4. How to call the quasiquoter for haskell syntax explicitly?
5. Template Haskell: Is there a function (or special syntax) that parses a String and returns Q Exp?

Но, лучше рассмотрим пару игрушечных примеров:

1. Quasi-quoting DSLs for free
2. Brief Intro to Quasi-Quotation

Итак, документация Language.Haskell.TH.Quote говорит, что если нам необходимо использовать `[q| ... string to parse ... |]`, нам необходимо самим создать парсер-обработчик строки и использовать четыре варианта конструктора `QuasiQuoter` (точнее, те из них, что нам будут нужны, но каждый со своим парсером):

```
data QuasiQuoter = QuasiQuoter { quoteExp  :: String -> Q Exp,
                                 quotePat  :: String -> Q Pat,
                                 quoteType :: String -> Q Type,
                                 quoteDec  :: String -> Q [Dec] }
```

6.13.7. Template Haskell Quasi-quotation

Вот минимальный рабочий пример, создаём модуль `TrivQQ.hs`:

```
{-# LANGUAGE TemplateHaskell #-}

module TrivQQ where

import Language.Haskell.TH
import Language.Haskell.TH.Quote

qq1 :: QuasiQuoter
qq1 = QuasiQuoter {
          quoteExp  = stringE
        , quotePat  = undefined
        , quoteType = undefined
        , quoteDec  = undefined
        }
```

В общем-то, парсера никакого и нет, объявляем строку строкой в терминах абстрактного дерева Haskell. Вот соответствующий пример использования, в файле `useQQ.hs` определим:

```
{-# LANGUAGE QuasiQuotes #-}

import TrivQQ

ex :: String
ex = [qq1|Hello|]
```

Вывод будет какой заказывали:

```
*Main> ex
"Hello"
```

И ещё один пример по теме прошлой лекции о грамматиках и парсерах. Пусть у нас есть грамматика

```
S = 'c' | 'a', S, 'a' | 'b', S, 'b';
```

для распознавания палиндрома в алфавите `{a,b,c}` с явно указанной серединой (символом `c`).

Опишем парсер и функции-утилиты для проверки строк:

```
import Text.ParserCombinators.Parsec

ruleS :: Parser ()
ruleS = (char 'c'  >> return ())
        <|>
        (char 'a' >> ruleS >> char 'a' >> return ())
        <|>
        (char 'b' >> ruleS >> char 'b' >> return ())


acbReady :: Parser Bool
acbReady = ruleS >> eof >> return True

parser = parse acbReady ""

go str = either ((error . show)) id res
               where res = parser str

goBool str = either (\_ -> False) id res
               where res = parser str
```

Теперь оформим модуль с парсером и добавим его в обработчик наших скобок:

```
{-# LANGUAGE TemplateHaskell #-}

module MyPars where

import Language.Haskell.TH
import Language.Haskell.TH.Quote

import Text.ParserCombinators.Parsec

ruleS :: Parser ()
ruleS = ...

acbReady :: Parser Bool
acbReady = ...

parser = parse ...
```

```
goBool str = ...

qq1 :: QuasiQuoter
qq1 = QuasiQuoter {
          quoteExp  = \str -> do
            let b = goBool str
            dataToExpQ (const Nothing) b
        , quotePat  = undefined
        , quoteType = undefined
        , quoteDec  = undefined
        }
```

(здесь полученное булево выражение будет превращено в необходимое представление с помощью функции `dataToExpQ`).

Во втором файле будем вызывать его в тестах:

```
{-# LANGUAGE QuasiQuotes #-}

import MyPars

t1 :: Bool
t1 = [qq1|aca|]

t2 = [qq1|c|]
t3 = [qq1|abcba|]
t4 = [qq1|abca|]
```

И при запуске в ghci получим:

```
*Main> t1
True
*Main> t2
True
*Main> t3
True
*Main> t4
False
```

Добавление более сложной функциональности, например в виде кода программ на каком-либо алгоритмическом языке:

```
prog1 :: Prog
prog1 = [prog|
    var x ;
    x := read ;
    write (x + x + 1)
  |]
```

подробно рассмотрено в статье “Quasi-quoting DSLs for free”.

Реальный пример использования TH

При проверке 3-го месячного задания, когда студенты сдают одинаково именованные модули AnGeo, Lines, LinesPlanes с одинаковыми экспортируемыми функциями, возникает необходимость автоматически учесть число сделанных конкретным студентом функций. Помимо прогонки тестов, есть необходимость на стадии компиляции получить предварительную информацию, какие функции реально импортированы, а какие собственно так и остались закомментированны. При этом, не хотелось бы, чтобы компилятор, обнаружив недостающую функцию, тут же закончил работу с ошибкой. Желательно, чтобы в этом случае использовалась некоторая функция по умолчанию, т.е. «заглушка».

Например, логика работы может быть выражена так:

```
import M

 f2 = if (defined f1) then f1
                      else {smth. by default}
```

Для удобства зададим обеспечивающие (provide..) модуль и функцию, которые с помощью lookupValueName из модуля Language.Haskell.TH определяют наличие заданного имени во время компиляции. Если имя не найдено, выводятся некоторые предоставленные объявления. В соответствии с правилами работы TH вынесем это в отдельный модуль:

```
module ProvideCommand where

import Language.Haskell.TH

provideCommand :: String -> Q [Dec] -> Q [Dec]
provideCommand nam defn = do
  mval <- lookupValueName nam
  case mval of
    Just _ -> return []
    Nothing -> defn
```

Теперь, в основном программном блоке применим созданную выше функцию provideCommand, которая в случае отсутствия в импортируемом модуле M нужной нам функции double обеспечит некоторую минимально необходимую «заглушку»:

```
{-# LANGUAGE TemplateHaskell #-}

import M
import ProvideCommand

provideCommand "double" [d| double x = x * 2 |]

main = print (double 15)
```

Ещё пример использования TH

Вопрос, заданный студентами, и реализация ответа на него: Equality by data constructors template

Пусть есть тип фигур

```
data Figure = Circle Double | Square Double
```

Мы хотим, чтобы он был инстансом класса **Eq**, а реализация сравнения была следующая: фигуры равны тогда и только тогда, когда их конструкторы данных равны (производные типы не должны учитываться), т.е. круги равны кругам, квадраты квадратам, а их размеры значения не имеют:

```
(Circle 1) == (Square 1) = False
(Circle 2) == (Circle 3) = True
```

Но при этом мы хотели бы более 100 видов фигур и не хотели бы делать инстансы в ручном режиме для проверок :) (решение только касается автоматизации проверок)

Haskell-код внутри Haskell-кода

Интересным вопросом и продолжением данной темы является создание аналога функции `eval(str)` из JavaScript и других скриптовых языков, которая принимает строку, представляющую корректный код данного языка, парсит его, преобразует в AST и исполняет.

Возможности `TH` позволяют это сделать, но как указывают в StackOverflow — напрямую это не слишком просто.

Template Haskell: Is there a function (or special syntax) that parses a String and returns Q Exp?

How to call the quasiquoter for haskell syntax explicitly?

Для облегчения задачи можно использовать различные готовые решения. Ближайшим для `TH` является использование пакета haskell-src-meta, который обеспечивает подходящие парсеры:

```
parsePat  :: String -> Either String Pat
parseExp  :: String -> Either String Exp
parseType :: String -> Either String Type
parseDecs :: String -> Either String [Dec]
```

Вот минимальный рабочий пример на базе уже рассмотренного. Определим модуль с шаблоном, генерируемым из строки:

```
module TupleReplicateStr where

import Language.Haskell.TH
import Language.Haskell.Meta.Parse

str = "\\x -> (x,x,x)"

getRight (Right b) = b

tupleReplicate :: Q Exp
tupleReplicate = do
    return $ getRight (parseExp str)
```

и используем его:

```
{-# LANGUAGE TemplateHaskell #-}

import TupleReplicateStr

main = do print $ $(tupleReplicate) 2
```

получим:

```
(2,2,2)
```

Разумеется есть и другие решения, напр.:

- Плагин System.Eval.Haskell
- Встраиваемый интерпретатор hint: hint: Runtime Haskell interpreter (GHC API wrapper) или github: hint

Литература по теме

1. *Geoffrey B. Mainland*. Why It's Nice to be Quoted: Quasiquoting for Haskell.
2. Template Haskel.
3. *Sheard T., Jones S.P.* Template Meta-programming for Haskell.
4. 6.13. Template Haskell
5. 6.13.7. Template Haskell Quasi-quotation
6. wiki.haskell: Quasiquotation.
7. A look at QuasiQuotation
8. Basic Tutorial of Template Haskell.
9. First stab at Template Haskell.
10. Quasi-quoting DSLs for free.
11. wiki.haskell: A practical Template Haskell Tutorial.
12. Мои первые эксперименты с Template Haskell.
13. *Greg Weber*. Code that writes code and conversation about conversations.
14. Введение в Template Haskell. Часть 1. Необходимый минимум.
15. Template Haskell 101.
16. Quasiquotation 101.
17. *Matt Parsons*. Template Haskell Is Not Scary.
18. *Mark Karpov*. Template Haskell tutorial.
19. 24 Days of GHC Extensions: Template Haskell.
20. *Geoffrey Mainland*. Type-Safe Runtime Code Generation with (Typed) Template Haskell.