Template Haskell Tutorial

short, illustrated examples
from Illustrated Haskell <http://illustratedhaskell.org>

Motivating example

- `fst(x,_)       = x`
- `fst3(x,_,_)    = x`
- `fst4(x,_,_,_) = x`
- …

- `print $ fst3 ("hello world", 1, 2)`
- `print $ fst4 ("hello world", 1, 2, 3)`

So repetitive!

Template Haskell to the rescue!

Usage:

```
{-# LANGUAGE TemplateHaskell #-}
print $ $(fstN 3) ("hello world", 1, 2)
print $ $(fstN 4) ("hello world", 1, 2, 3)
```

How to write it?

```
-- FstN.hs
{-# LANGUAGE TemplateHaskell #-}
module FstN where
import Language.Haskell.TH

fstN :: Q Exp
fstN n = do
    x <- newName "x"
    return $ LamE [TupP $
        VarP x : replicate (n-1) WildP] (VarE x)
```

OK, how about explaining it?

- Every time you want to write something in TH, you start with:

```
runQ [| ... |]
```

- GHC will tell you how to write it. For example, if we wanted to write a splice that will produce `\(x,_,_) -> x`

```
$ ghci -fth
> :m +Language.Haskell.TH
> runQ [| \(x,_,_) -> x |]
LamE [TupP [VarP x_1,WildP,WildP]] (VarE x_1)
> :t it
it :: Exp
```

- That's it, no need to remember anything! Just ask GHC!

Writing `fst3` in TH

- So we already have an `Exp`, how about those x_1?

```
LamE [TupP [VarP x_1,WildP,WildP]] (VarE x_1)
```

```
> :t (VarP, VarE)
(VarP, VarE) :: (Name -> Pat, Name -> Exp)
```

- So, `VarP` and `VarE` takes a Name. Let's see how we can satisfy them:

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

- A ha! So we can just plug it into the expression GHC gave us:

```
fst3 = do
    x <- newName "x"
    LamE [TupP [VarP x,WildP,WildP]] (VarE x)
```

Evolving `fst3` into `fstN`

- The following corresponds to the expression `\(x,_,_) -> x`

```
fst3 = do
    x <- newName "x"
    LamE [TupP [VarP x,WildP,WildP]] (VarE x)
```

- Not surprisingly, to make `fst4`, we just need to make 3 `WildP`:

```
fst4 = do
    x <- newName "x"
    LamE [TupP [VarP x,WildP,WildP,WildP]] (VarE x)
```

- And we can easily generalize it into `fstN`

```
fstN n = do
    x <- newName "x"
    LamE [TupP (VarP x : replicate (n-1) WildP)] (VarE x)
```

Using `fstN`

- For technical reasons, splices must be defined in a separate module.
- So we need to create a new module to use the splice we defined:

```
-- TestFstN.hs
main = print $ $(fstN 3) ("hello world", 1, 3)
```

Quasi Quotes

Quasi Quotes

- The `[| ... |]` notation that you just used is the quasi quotes for Haskell expression.
- The contents within quasi quotes will be parsed at compile time.
- Example: in `Data.Array.Repa.Stencil`, you could define a stencil like this

```
[stencil2|  0 1 0
            1 0 1
            0 1 0 |]
```

- It is converted to:

```
makeStencil2 (Z:.3:.3)
   (\ix -> case ix of
             Z :. -1 :.  0  -> Just 1
             Z :.  0 :. -1  -> Just 1
             Z :.  0 :.  1  -> Just 1
             Z :.  1 :.  0  -> Just 1
             _              -> Nothing)
```

Quasi Quotes

- When you do `[| \x -> x |]`, the string inside the brackets is parsed by the Haskell compiler and gives you back the AST (Abstract Syntax Tree)

Let's do a simple example

- We will build a structure to represent HTML documents
- For simplicity, we omit attributes, self closing tags, etc.

```
-- HTML.hs
{-# LANGUAGE TemplateHaskell, QuasiQuotes #-}
module HTML where
data Node = Tag String [Node] -- tag name, children
          | Text String
          deriving Show
```

- Our target is to use quasi quotes to build a document tree:

```
-- HTMLTest.hs
import HTML
doc :: Node
doc = [html|<html>Hello, <strong>TH</strong> world!</html>]
   -- Node "html" [Text "Hello, ",
                   Tag "strong" [Text "TH"],
                   Text " world!"]
```

First, a simple HTML parser

- We'll sidetrack a bit and make a dead simple HTML parser using Parsec
- Our focus here isn't Parsec so we can just skim over this function that does the right thing

```
-- HTML.hs
textNode :: Parser Node
textNode = fmap Text $ many1 $ satisfy (/='<')

tagNode :: Parser Node
tagNode = do
    tagName <- char '<' *> many1 letter <* char '>'
    children <- many $ try tagNode <|> textNode
    string "</" >> string tagName >> char '>'
    return $ Tag tagName children
```

A simple test for our parser

```
$ ghci HTML.hs
> parseTest tagNode "<html>Hello, <strong>TH</strong> world!</html>"
Tag "html" [Text "Hello, ",Tag "strong" [Text "TH"],Text " world!"]
```

- It works!

Now we can write our QuasiQuoter

```
-- HTML.hs
Html :: QuasiQuoter
Html = QuasiQuoter
    htmlExpr
    undefined
    undefined
    undefined

htmlExpr :: String -> Q Exp
htmlExpr = undefined
```

- The `QuasiQuoter` takes 4 parameters. Each will be called when the quasi quote is being invoked to create:
  - An expression
    - `foo = [html| ... |]`
  - A pattern (for pattern matching)
    - `bar [html| ... |] = 3`
  - A type
  - A top-level declaration

- We will do *expression* and *pattern* in this example
- For more information consult GHC's documentation

- `htmlExpr` is supposed to parse the contents within `[html| … |]` and give back an `Exp`

```
htmlExpr :: String -> Q Exp
htmlExpr str = do
    filename <- loc_filename `fmap` location
     case parse tagNode filename str of
         Left err -> undefined
         Right tag -> [| tag |]
```

- As easy as that, `loc_filename` and `location` will give us the filename of the user.

Let's compile it!

```
$ ghc HTML.hs
Error: No instance for (Lift Node) arising from
arising of `tag'…
```

- What is that? Well maybe we can satisfy it by implementing the Lift instance for Node, as instructed:

```
instance Lift Node where
    lift (Text t) = [| Text t |]
    lift (Tag name children) = [| Tag name children |]
```

Let's try it out

```
-- HTMLTest.hs
{-# LANGUAGE TemplateHaskell , QuasiQuotes #-}
import HTML
main = print [html|<html>Hello, <strong>TH</strong>
world!</html>|]
```

```
$ ghci HTMLTest.hs
> main
Tag "html" [Text "Hello, ",Tag "strong" [Text "TH"],Text
" world!"]
```

- It works!

Quasi quoting for patterns

- Now let's try to do some operations on our HTML structure
- In this example we will convert an HTML tree into Markdown
- Markdown is a simple wiki syntax

- Example Markdown:

```
Let's **rock** and _roll_!
```

- Corresponding HTML:

```
<html>Let's <strong>rock</strong> and <em>roll</em></html>
```

- Usually people convert Markdown to HTML. We will do it the other way here.

Let's make a simple converter

```
-- HTMLTest.hs
markdown (Tag "strong" children) = "**" ++ concatMap markdown children ++ "**"
markdown (Tag "em" children)     = "_" ++ concatMap markdown children ++ "_"
markdown (Tag _ children)        = concatMap markdown children
markdown (Text t)                = t
```

- That's some normal Haskell. For fun, we're going to turn it into:

```
-- HTMLTest.hs
markdown [html|<strong>|] = "**" ++ concatMap markdown children ++ "**"
markdown [html|<em>|]     = "_" ++ concatMap markdown children ++ "_"
markdown [html|<_>|]      = concatMap markdown children
markdown [html|#text|]    = text
```

- Are we gaining anything? Honestly not much. But we'll do it for the sake of an TH example.

Add a pattern parser to our QuasiQuoter

```haskell
-- HTML.hs
html :: QuasiQuoter
html = QuasiQuoter htmlExpr htmlPat
  undefined undefined

htmlPat :: String -> Q Pat
htmlPat "<_>"      = [p| Tag _ children |]
htmlPat "#text"    = [p| Text text |]
htmlPat ('<':rest) = undefined -- ...
```

Asking GHC again…

- Now how do we write the "rest" case? Let's ask GHC

```
> runQ [p| Tag "strong" chlidren |]
ConP HTML.Tag [ LitP (StringL "strong")
              , VarP chlidren]
```

- So there we have almost had it.
- We just need to use Name at appropriate places and follow the types:

```
htmlPat ('<':rest) = return $
    ConP (mkName "HTML.Tag")
        [ LitP (StringL (init rest))
        , VarP (mkName "children")]
```

Some explanations

```
htmlPat ('<':rest) = return $
    ConP (mkName "HTML.Tag")
        [ LitP (StringL (init rest))
        , VarP (mkName "children")]
```

- Here we see mkName intead of newName
- mkName "foo" will translate into an identifier "foo" literally
- mkName "foo" will become something like "foo_1".
- You'll use this when you want to avoid name collisions

Let's test it out!

```haskell
-- HTMLTest.hs
{-# LANGUAGE TemplateHaskell , QuasiQuotes #-}
import HTML

doc = [html|<html>Hello, <strong>TH</strong> world!</html>|]

markdown [html|<_>|]      = concatMap markdown children
markdown [html|#text|]    = text
markdown [html|<strong>|] =
    "**" ++ concatMap markdown children ++ "**"

main = print . markdown $ doc
```

- `$ runhaskell HTMLTest.hs`
- `"Hello, **TH** world!"`

Summary

- Use `runQ` to have GHC write the splice for you
- Then just fix it up by following the type