Typst-Unlit

Write literate Haskell programs in Typst

tangled.org/@oppi.li/typst-unlit

Serves: 1 **Prep Time: 10min** **Compile Time: 10ms**

A literate program is one where comments are first-class citizens, and code is explicitly demarcated, as opposed to a regular program, where comments are explicitly marked, and code is a first-class entity.

GHC supports literate programming out of the box, by using a preprocessor to extract code from documents. This preprocessor is known as *unlit*[1]. GHC also supports *custom* preprocessors, which can be passed in via the `-pgmL` flag. This very document you are reading, is one such preprocessor program that allows extracting Haskell from Typst code (although it has been rendered to HTML, PDF or markdown depending on where you are reading it)[2]!

This recipe not only gives you a fish (the typst-unlit preprocessor), but also, teaches you how to fish (write your own preprocessors).

Ingredients

To write your own preprocessor:

- GHC: the Glorious Haskell Compiler
- Typst: to generate PDFs
- And thats it! No stacking, shaking or caballing here.

To compile this very document:

- The bootstrap program
- GHC: to produce an executable program
- Typst: to produce a readable PDF

Pro Tip: If you're missing any ingredients, your local nixpkgs should stock them!

Instructions

The idea behind the unlit program is super simple: iterate over the lines in the supplied input file and replace lines that aren't Haskell with an empty line! To detect lines that are Haskell, we look for the ```` ```haskell ```` directive and stop at the end of the code fence. Simple enough! Annoyingly, Haskell requires that imports be declared at the top of the file. This results in literate Haskell programs always starting with a giant block of imports:

```
-- So first we need to get some boilerplate and imports out of the way.
```

— Every literate programmer

Oh gee, if only we had a tool to put the important stuff first. Our preprocessor will remedy this wart, with the `haskell-top` directive to move blocks to the top. With that out of the way, lets move onto the program itself!

[1] https://gitlab.haskell.org/ghc/ghc/-/tree/master/utils/unlit

[2] This document needs itself to compile itself! This is why a bootstrap program is included.

Step 1: The maincourse

I prefer starting with `main` but you do you. Any program that is passed to `ghc -pgmL` has to accept exactly 4 arguments:

- `-h`: ignore this for now
- `<label>`: ignore this for now
- `<infile>`: the input lhaskell source code
- `<outfile>`: the output Haskell source code

Invoke the runes to handle CLI arguments:

```
main = do
  args <- getArgs
  case args of
    ["-h", _label, infile, outfile] -> process infile outfile
    _ -> die "Usage: typst-unlit -h <label> <source> <destination>"
```

You will need these imports accordingly (notice how I am writing my imports *after* the main function!):

```
import System.Environment (getArgs)
import System.Exit        (die)
```

Now, we move onto defining `process`:

Step 2: The processor

`process` does a bit of IO to read from the input file, remove comments, and write to the output file, `removeComments` is a pure function however:

```
process :: FilePath -> FilePath -> IO ()
process infile outfile = do
    ls <- lines <$> readFile infile
    writeFile outfile $ unlines $ removeComments ls
```

Step 3: Removing comments

We will be iterating over lines in the file, and wiping clean those lines that are not Haskell. To do so, we must track some state as we will be jumping in and out of code fences:

```
data State
  = OutsideCode
  | InHaskell
  | InHaskellTop
  deriving (Eq, Show)
```

To detect the code fences itself, we can define a few matcher functions, here is one for the ```` ```haskell ```` pattern:

```
withTag :: (String -> Bool) -> String -> Bool
withTag pred line = length ticks > 2 && pred tag
    where (ticks, tag) = span (== '`') line

isHaskell :: String -> Bool
isHaskell = withTag (== "haskell")
```

You will notice that this will also match ```` ````haskell ````, and this is intentional. If your text already contains 3 backticks inside it, you will need 4 backticks in the code fence and so on.

We do the same exercise for `haskell-top`:

```
isHaskellTop = withTag (== "haskell-top")
```

And for the closing code fences:

```
isCodeEnd = withTag null
```

`removeComments` itself, is just a filter, that takes a list of lines and removes comments from those lines:

```
removeComments :: [String] -> [String]
removeComments ls = go OutsideCode ls [] []
```

Finally, `go` is a recursive function that starts with some `State`, a list of input lines, and two more empty lists that are used to store the lines of code that go at the top (using the `haskell-top` directive), and the ones that go below, using the `haskell` directive:

```
go :: State -> [String] -> [String] -> [String] -> [String]
```

When the input file is empty, we just combine the `top` and `bottom` stacks of lines to form the file:

```
go _ [] top bot = reverse top ++ reverse bot
```

Next, whenever, we are `OutsideCode`, and the current line contains a directive, we must update the state to enter a code block:

```
go OutsideCode (x : rest) top bot
  | isHaskellTop x = go InHaskellTop rest top ("" : bot)
  | isHaskell x = go InHaskell rest top ("" : bot)
  | otherwise = go OutsideCode rest top ("" : bot)
```

When we are already inside a Haskell code block, encountering a triple-tick should exit the code block, and any other line encountered in the block is to be included in the final file, but below the imports:

```
go InHaskell (x : rest) top bot
  | isCodeEnd x = go OutsideCode rest top ("" : bot)
  | otherwise = go InHaskell rest top (x : bot)
```

And similarly, for blocks that start with the `haskell-top` directive, lines encountered here go into the `top` stack:

```
go InHaskellTop (x : rest) top bot
  | isCodeEnd x = go OutsideCode rest top ("" : bot)
  | otherwise = go InHaskellTop rest (x : top) bot
```

And thats it! Gently tap the baking pan against the table and let your code settle. Once it is set, you can compile the preprocessor like so:

```
ghc -o typst-unlit typst-unlit.hs
```

And now, we can execute our preprocessor on literate Haskell files!

Serving

To test our preprocessor, first, write a literate Haskell file containing your typst code:

````
= Quicksort in Haskell
The first thing to know about Haskell's syntax is that parentheses
are used for grouping, and not for function application.

```haskell
quicksort :: Ord a => [a] -> [a]
quicksort []     = []
quicksort (p:xs) = (quicksort lesser) ++ [p] ++ (quicksort greater)
    where
        lesser  = filter (< p) xs
        greater = filter (>= p) xs
```

The parentheses indicate the grouping of operands on the
right-hand side of equations.
````

Remember to save that as a `.lhs` file, say `quicksort.lhs`. Now you can compile it with both `ghc` ...

```
ghci -pgmL ./typst-unlit quicksort.lhs
GHCi, version 9.10.3: https://www.haskell.org/ghc/  :? for help
[1 of 2] Compiling Main             ( quicksort.lhs, interpreted )
Ok, one module loaded.
ghci> quicksort [3,2,4,1,5,4]
[1,2,3,4,4,5]
```

... and `typst`:

```
typst compile quicksort.lhs
```

And there you have it! One file that can be interpreted by `ghc` and rendered beautifully with `typst` simultaneously.

Notes

This entire document is just a bit of ceremony around writing preprocessors, the Haskell code in this file can be summarized in this shell script:

```
#!/usr/bin/env bash

# this does the same thing as typst-unlit.lhs, but depends on typst and jq
# this script does clobber the line numbers, so users beware

typst query "$3" 'raw.where(lang: "haskell-top")' | jq -r '.[].text' > "$4"
typst query "$3" 'raw.where(lang: "haskell")' | jq -r '.[].text' >> "$4"
```

This document mentions the word “Haskell” 60 times.