pandoc-minted

A pandoc filter to render LaTeX code blocks using minted

Usage

```
pandoc [OPTIONS] --filter pandoc-minted [FILES]
```

Source

As usual, declare a module *Main*...

```
module Main where
```

... and `import` some useful definitions:

- `intercalate` from *Data.List*,

```
import           Data.List           (intercalate)
```

- `getArgs` from *System.Environment*,

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

- `topDown` from *Text.Pandoc.Generic*,

```
import           Text.Pandoc.Generic (topDown)
```

- and everything from *Text.Pandoc.JSON*.

```
import           Text.Pandoc.JSON
```

The *Minted* Data Type

Define a data type *Minted* to more expressively handle inline code and code blocks.

```
data Minted
  = MintedInline (String, String) String
  | MintedBlock (String, String) String
```

Define a *Show* instance for *Minted*, in order to generate LaTeX code.

```
instance Show Minted where
  show (MintedInline (attrs, language) contents) =
      "\\mintinline[" ++ attrs ++ "]{" ++ language ++ "}{" ++ contents ++ "}"
  show (MintedBlock (attrs, language) contents) =
      unlines [ "\\begin" ++ "{minted}[" ++ attrs ++ "]{" ++ language ++ "}"
              , contents
              , "\\end" ++ "{minted}"
              ]
```

The `main` Function

Run `minted` as a JSON filter.

```
main :: IO ()
main = toJSONFilter . go =<< getArgs
  where
    go :: [String] -> (Pandoc -> Pandoc)
    go ["latex"] = minted
    go _         = id
```

The `minted` Filter

```
minted :: Pandoc -> Pandoc
minted = topDown (concatMap mintinline) .
         topDown (concatMap mintedBlock)
```

Handle Inline Code

Transform `Code` into a `\mintinline` call, otherwise return a given `Inline`.

```
mintinline :: Inline -> [Inline]
mintinline (Code attr contents) =
    let
        latex = show $ MintedInline (unpackCode attr "text") contents
    in
        [ RawInline (Format "latex") latex ]
mintinline x = [x]
```

Handle Code Blocks

Transform a `CodeBlock` into a `minted` environment, otherwise return a given `Block`.

```
mintedBlock :: Block -> [Block]
mintedBlock (CodeBlock attr contents) =
    let
        latex = show $ MintedBlock (unpackCode attr "text") contents
    in
        [ RawBlock (Format "latex") latex ]
mintedBlock x = [x]
```

Helper Functions

Given a triplet of `Attributes` (identifier, language(s), and key/value pairs) and a default language, return a pair of `minted` attributes and language.

```
unpackCode :: Attr -> String -> (String, String)
unpackCode (_, [], kvs) defaultLanguage =
    (unpackAttrs kvs, defaultLanguage)
unpackCode (identifier, "sourceCode" : "literate" : language : _, kvs) _ =
    (unpackAttrs kvs, language)
unpackCode (identifier, "sourceCode" : language : _, kvs) _ =
    (unpackAttrs kvs, language)
unpackCode (_, language : _, kvs) _ =
    (unpackAttrs kvs, language)
```

Given a list of key/value pairs, return a string suitable for `minted` options.

```
unpackAttrs :: [(String, String)] -> String
unpackAttrs kvs = intercalate ", " [ k ++ "=" ++ v  | (k, v) <- kvs ]
```