The nox Source Manual

Linus Arver

ver. 0.1.0-4-gf123c9e

2014-01-07 16:34:14 -0800

Contents

1 Introduction

nox is a program that comments or uncomments text from STDIN and prints the result to STDOUT.

1.1 How to Read This Manual

The general format is to show the raw source code first, followed by commentary on what the just-shown block of code does. The idea is to try to read the source code first, and then have it explained in detail later. Whenever the commentary says "this block of code" or "here", it is referring to the block of code directly above it.

2 nox.lhs

```
{-# LANGUAGE RecordWildCards #-}

module Main where

import Control.Monad (when)
import qualified Data.Text.Lazy as T
import qualified Data.Text.Lazy.IO as T
import System.Exit
import System.IO

import NOX.Core
import NOX.Option
import NOX.Util
```

The above are some basic imports. The only interesting import is the `when` function, which allows us to write some succinct one-liners in `main`.

```
main :: IO ()
main = do
    hSetBuffering stdout NoBuffering
    hSetBuffering stderr NoBuffering
    opts <- getOpts
    (opts', argsErrNo) <- argsCheck opts
    when (argsErrNo > 0) $ do
        errMsg $ "code " ++ show argsErrNo
        exitWith $ ExitFailure argsErrNo
    text <- T.hGetContents stdin
    when (T.null text) . abort $ (,) "nothing from STDIN" 1
    T.putStr $ (if uncomment opts' then remCmt else makeCmt) opts' text
```

This is the main program section and handles all arguments passed to `nox`. The given options are first checked for sanity before `nox` does anything; if we encounter an error, we exit immediately.

3 NOX/Option.lhs

```
{-# LANGUAGE DeriveDataTypeable #-}
{-# LANGUAGE RecordWildCards #-}

module NOX.Option where

import Data.List (intercalate)
import qualified Data.Text.Lazy as T
import System.Console.CmdArgs.Implicit
```

```
import NOX.Language
import NOX.Meta
import NOX.Util

data Opts = Opts
    { lang :: Language
    , multi :: Bool
    , uncomment :: Bool
    , sline :: String
    } deriving (Data, Typeable, Show, Eq)

progOpts :: Opts
progOpts = Opts
    { lang = Shell &= typ "LANGUAGE" &= help ("language type; values are: "
        ++ names
        ++ "; default is `sh' for '#' comments")
    , multi = False
        &= help "multi-line comment style (default is single); if the target language\
            \ lacks multiline symbols, then the single-line symbol is used"
    , uncomment = False
        &= help "uncomment the text; you only need to specify the particular language\
            \ --- nox will take care of both the language's single and multiline\
            \ symbols"
    , sline = [] &= typ "STRING"
        &= help "custom single-line comment string; overrides `-l' option"
    }
    where
    names :: String
    names = intercalate ", "
        $ zipWith (\a b -> a ++ " " ++ b)
            (map (enclose sQuotes . ldExt) langs)
            (map (enclose parens . show) langs)

getOpts :: IO Opts
getOpts = cmdArgs $ progOpts
    &= versionArg [explicit, name "version", name "v", summary _PROGRAM_INFO]
    &= summary (_PROGRAM_INFO ++ ", " ++ _COPYRIGHT)
    &= help "comment/uncomment out blocks of code"
    &= helpArg [explicit, name "help", name "h"]
    &= program _PROGRAM_NAME
    &= details
        [ "Examples:"
        , ""
        , "  echo \"a\\nab\\nabc\" | nox -l c"
        , ""
        ,  "//a"
```

```
        ,  "//ab"
        ,  "//abc"
        , ""
        , "  echo \"a\\nab\\nabc\" | nox -l c -m"
        , ""
        ,  "/*"
        ,  "a"
        ,  "ab"
        ,  "abc"
        ,  "*/"
        ]
```

The options are defined by the `Opts` data type, and we write our version of it with `progOpts`. We add some more customizations to how `nox` will behave (e.g., `-h` and `-v` flags) with `getOpts`. This is standard practice for writing command line options with the `CmdArgs` library.

```
argsCheck :: Opts -> IO (Opts, Int)
argsCheck opts = return . (,) opts =<< argsCheck' opts
    where
    argsCheck' :: Opts -> IO (Int)
    argsCheck' Opts{..}
        | not (elem lang langs) = errMsg "unsupported language" >> return 1
        | multi && noMulti && noSingle
            = errMsg "--multi requested, but language supports neither multiline nor\
                \ single-line comments"
            >> return 1
        | otherwise = return 0
        where
        noMulti = T.null . uncurry T.append $ ldCmtM lang
        noSingle = T.null $ ldCmtS lang
```

Check for errors, and return an error code (a positive number) if any errors are found.

4 NOX/Language.lhs

```
{-# LANGUAGE DeriveDataTypeable #-}
{-# LANGUAGE OverloadedStrings #-}

module NOX.Language where

import Data.Data
import qualified Data.Text.Lazy as T

data Language
    = C
    | EmacsLisp
```

```
    | Haskell
    | HTML
    | Lilypond
    | Shell
    | TeX
    deriving (Data, Eq, Enum, Show, Typeable)
```

`Language` defines the language types recognized by `nox`.

```
langs :: [Language]
langs = enumFrom C

class LangDesc a where
    ldCmtS :: a -> T.Text
    ldCmtM :: a -> (T.Text, T.Text)
    ldExt :: a -> String

instance LangDesc Language where
    ldCmtS l = case l of
        C -> "//"
        EmacsLisp -> ";"
        Haskell -> "--"
        HTML -> x
        Lilypond -> "%"
        Shell -> "#"
        TeX -> "%"
        where
        x = T.empty
    ldCmtM l = case l of
        C -> ("/*", "*/")
        EmacsLisp -> x
        Haskell -> ("{-", "-}")
        HTML -> ("<!--", "-->")
        Lilypond -> ("%{", "%}")
        Shell -> x
        TeX -> x
        where
        x = (T.empty, T.empty)
    ldExt l = case l of
        C -> "c"
        EmacsLisp -> "el"
        Haskell -> "hs"
        HTML -> "html"
        Lilypond -> "ly"
        Shell -> "sh"
        TeX -> "tex"
```

The `LangDesc` class defines comment strings and also the extension type of a particular language.

5 NOX/Core.lhs

```
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE RecordWildCards #-}

module NOX.Core where

import qualified Data.Text.Lazy as T

import NOX.Language
import NOX.Option
import NOX.Parse

makeCmt :: Opts -> T.Text -> T.Text
makeCmt Opts{..}
    | multi = flip T.append mcbn . T.append mcan
    | otherwise = T.unlines . map (makeSingleComment sline) . T.lines
    where
    (mca, mcb) = ldCmtM lang
    mcan = T.append mca "\n"
    mcbn = T.append mcb "\n"
    makeSingleComment sline' l
        | T.null l = l
        | not (null sline') = T.append (T.pack sline') l
        | otherwise = T.append (ldCmtS lang) l
```

`makeCmt` comments out the given chunk of text. For single-line comments, simply prepend each non-empty line with the comment string. For multiline comments, we prepend and append lines containing just the multiline comment string pairs.

```
remCmt :: Opts -> T.Text -> T.Text
remCmt Opts{..} src
    | multi = case mlineCommentExists src $ ldCmtM lang of
        (True, removed) -> removed
        (False, _) -> src
    | otherwise = T.unlines . map (tryRemSlineComment sline) $ T.lines src
    where
    tryRemSlineComment :: String -> T.Text -> T.Text
    tryRemSlineComment sline' rawline = if scmtExists
        then removed
        else rawline
        where
        (scmtExists, removed) = slineCommentExists rawline $
            if null sline'
                then ldCmtS lang
                else T.pack sline'
```

`remCmt` uncomments a given chunk of (probably commented) text. For single-line mode, remove all single-line comment characters found on each line independently of other lines. For multiline comments, we find and remove the mulitline comment string pairs.

6 NOX/Parse.lhs

```
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE RecordWildCards #-}

module NOX.Parse where

import qualified Data.Text.Lazy as T
import Text.Parsec.Char
import Text.Parsec.Combinator
import Text.Parsec.Prim
import Text.Parsec.Text.Lazy

slineDetect :: String -> Parser String
slineDetect x = do
    leadSpace <- many $ oneOf " \t"
    _ <- string x
    rest <- manyTill anyChar eof
    return $ leadSpace ++ rest
```

`slineDetect` tries to find the given needle string `x`. We make sure to skip any leading whitespace, and then return the line without the (leading) comment characters.

```
mlineDetect :: (String, String) -> Parser String
mlineDetect (a, b) = do
    beg <- manyTill anyChar . lookAhead $ string a
    mid <- between (string a) (string b) (manyTill anyChar . lookAhead . try $ string b)
    end <- manyTill anyChar eof
    return $ beg ++ mid ++ end
```

This is just like `slineDetect`, but for multiline comment strings.

```
slineCommentExists :: T.Text -> T.Text -> (Bool, T.Text)
slineCommentExists src slineCmtStr
    | T.null slineCmtStr = (False, T.empty)
    | otherwise = case parse (slineDetect slineCmtStr') [] src of
        Left _ -> (False, T.empty)
        Right str -> (True, T.pack str)
    where
    slineCmtStr' = T.unpack slineCmtStr
```

Here we check if a single line comment exists; if so, we return the uncommented version of that line.

```
mlineCommentExists :: T.Text -> (T.Text, T.Text) -> (Bool, T.Text)
mlineCommentExists src (a, b)
    | T.null a && T.null b = (False, T.empty)
    | otherwise = case parse (mlineDetect (a', b')) [] src of
        Left _ -> (False, T.empty)
        Right str -> (True, T.pack str)
    where
    (a', b') = (T.unpack a, T.unpack b)
```

This is the multi-line counterpart of `slineCommentExists`.

7 NOX/Util.lhs

```
module NOX.Util where

import System.Exit
import System.IO

enclose :: (String, String) -> String -> String
enclose (a, b) w = a ++ w ++ b

enclose' :: (String, String) -> String -> String
enclose' pair = enclose sSpaces . enclose pair

dQuotes
    , sQuotes
    , parens
    , bracks
    , braces
    , sSpaces :: (String, String)
dQuotes = ("\"", "\"")
sQuotes = ("`", "'")
parens = ("(", ")")
bracks = ("[", "]")
braces = ("{", "}")
sSpaces = (" ", " ")
```

These two functions make it easier to surround text with a given pair of strings.

```
abort :: (String, Int) -> IO ()
abort (msg, eid) = do
    errMsg msg
    hPutStrLn stderr "operation aborted"
    exitWith $ ExitFailure eid

errMsg :: String -> IO ()
```

```
errMsg msg = hPutStrLn stderr $ "error: " ++ msg
```

These are some basic error-logging/exiting functions.

8 NOX/Meta.lhs

```
module NOX.Meta where

_PROGRAM_NAME, _PROGRAM_VERSION, _PROGRAM_INFO, _COPYRIGHT :: String
_PROGRAM_NAME = "nox"
_PROGRAM_VERSION = "0.1.0"
_PROGRAM_INFO = _PROGRAM_NAME ++ " version " ++ _PROGRAM_VERSION
_COPYRIGHT = "(C) Linus Arver 2013-2014"
```