COPYRIGHT

EXAMPLE λTEX OUTPUT

This module defines a number of functions for managing and formatting syntactic and semantic error diagnostic messages. The library is built around the DIAG ADT. DIAG objects can be constructed using one of the two functions errorMsg and warningMsg, and messages are stored in a lazily-computed sequence of one or more MSG objects. At the end of the program, any generated messages are emitted to stderr using the emitDiagnostics IO action, as described below.

```
module DIAG (
  DIAG, errorMsg, warningMsg, diag, ok, emitDiagnostics,
  fatalError,
  noInputFiles,
  notEnoughCmdlineArgs,
  tooManyCmdlineArgs,
  invalidCmdlineFlag,
  missingCmdlineFlagArg,
  incompatibleCmdlineFlags,
  repeatedCmdlineFlag,
) where

import IO (hPutStr, hPutStrLn, stderr)
import POSN (POSN, lineno, colno, findLine)
import SYSTEM (exitFailure, getProgName, getEnv)
```

THE MESSAGE TYPE

First, we must define MSG as a (*severity*,*text*,*position*) tuple:

```
data SEVERITY = E | W
data MSG = M SEVERITY STRING POSN
```

Because, in the course of translating a source file, most functions can generate any number of error messages, the data type I use for storing messages is a list. However, I hide this from the user as there are better (but much more complicated) ways of doing things.

```
data DIAG = DIAG [MSG]
```

I also define a pair of simple constructor functions:

```
errorMsg, warningMsg :: STRING → POSN → DIAG

errorMsg s p = DIAG [M E s p]
warningMsg s p = DIAG [M W s p]
```

The diag function concatenates two message lists:

```
diag :: DIAG → DIAG → DIAG
diag (DIAG a) (DIAG b) = DIAG (a ++ b)
```

The ok function emits a diagnostics object containing no messages:

```
ok :: DIAG
ok = DIAG []
```

THE emitDiagnostics FUNCTION

emitDiagnostics is called from *main* to print any errors and warnings accumulated during processing of a single source file. If it encounters any true errors in the error list, it will abort the program with exitFailure.

The content of the file, as fetched by readFile, must be supplied to emitDiagnostics to allow it to annotate the error message with the offending line in the file. While we're at it, we also supply the filename so it doesn't have to be stored in each token position.

Finally, emitDiagnostics can be used in one or two modes. In "strict" mode, all warnings are treated as errors. Otherwise, warnings are displayed but are otherwise ignored in the error count.

```
emitDiagnostics :: DIAG → BOOL → STRING → STRING → IO ()
emitDiagnostics (DIAG msgs) strict filename contents =
  do
    abort ← showDiag msgs strict contents filename maxErrors
    if abort then exitFailure else return ()
```

maxErrors represents the maximum number of error messages printed before aborting the program.

```
maxErrors :: INT
maxErrors = 5
```

showDiag prints accumulated error messages to stderr. All warnings are emitted but at most maxErrors error messages are printed before giving up. The function returns a flag indicating if any errors have been encountered.

```
showDiag :: [MSG] → BOOL → STRING → STRING → INT → IO BOOL
showDiag [] _ _ _ _ = return FALSE
showDiag ((M W posn msg):rest) FALSE contents filename remaining =
  do
    showMsg "Warning" posn msg contents filename
    abort ← showDiag rest FALSE filename contents remaining
    return abort
showDiag _ _ _ _ 0 =
  do
    fatalError "Too many errors to continue."
    return TRUE
showDiag (M _ posn msg:rest) strict contents filename remaining =
```

```
    do
      showMsg "Error" posn msg contents filename
      showDiag rest strict contents filename (remaining − 1)
      return TRUE
```

Next, I define a function that prints a formatted error message. The messages are in the format:

```
[progname] Error/Warning near line N in "file":
           The actual error message.
-------------------------------------------------------------
the line of code with the error position indicated by "^"
-----^-------------------------------------------------------
```

where the screen width is given by getColumns. Error printing is performed directly from *main*, so *formatMessage* is supplied the content of the file and its name.

```
showMsg :: STRING → POSN → STRING → STRING → STRING → IO ()
showMsg severity p msg contents filename =
  do
    columns ← getColumns
    let
      line0 = severity ++ " on line " ++ show line
                       ++ " in " ++ show filename ++ ": "
      line1 = replicate columns '-'
      line2 = take columns (takeWhile (not.isbreak) (drop (lineoff p) contents))
      line3
        | col1 < columns = replicate col1 '-' ++ "^" ++ replicate col2 '-'
        | otherwise = line1
line = lineno p     — starts at 1
col1 = colno p − 1     — starts at 0
col2 = columns − 1 − col1     — any columns after the error marker
    do
      putMsg (line0 : lines msg)
      hPutStrLn stderr line1
      hPutStrLn stderr line2
      hPutStrLn stderr line3
```

Print the first two lines of a message. The first line is prefixed with the message prefix, and the remaining lines are indented by an appropriate number of spaces.

```
putMsg :: STRINGS → IO ()
putMsg (line:rest) =
  do
    prefix ← getPrefix
    hPutStrLn stderr (prefix ++ line)
    putMsg (map (λ_ → ' ') prefix) rest


putMsg :: STRING → STRINGS → IO ()
putMsg _ [] = return ()
putMsg prefix (line:rest) =
  do
    hPutStrLn stderr (prefix ++ line)
    putMsg prefix rest
```

TERMINAL INTERFACE

To format the message as described aboce, we need to obtain the base component of our program name and the screen width from the *COLUMNS* variable, if set.

```
getPrefix :: IO String
getPrefix =
  do
    progname ← getProgName
    let pretty = show (basename progname) in
      return ("[" ++ tail (take (length pretty − 1) pretty) ++ "] ")

getColumns :: IO Int
getColumns = catch getColumns (λerr → return defaultWidth)
  where
    getColumns =
      do
        env ← getEnv "COLUMNS"
        let
          (cols, rest) = read env
          cols′ = if cols < minWidth ∨ rest ≠ "" then defaultWidth else cols
        do
          return cols′
    minWidth = 30
    defaultWidth = 80
```

FATAL ERRORS

In the remainder of this module I define a bunch of common fatal errors. Each error is an I/O action that formats and prints the supplied messages and aborts the program with exitFailure.

```
fatalError :: String → IO ()
fatalError msg =
  do
    putMsg (lines msg)
    exitFailure

noInputFiles :: IO ()
noInputFiles = fatalError msg
  where
    msg =
      "No input files specified.\n\
      \Use the -? option to see detailed synopsis."

notEnoughCmdlineArgs :: Int → IO ()
notEnoughCmdlineArgs n = fatalError msg
  where
    msg =
      "Insufficient number of command-line arguments.\n\
      \(" ++ show n ++ " expected.)\n\
      \Use the -? option to see detailed synopsis."
```

```
tooManyCmdlineArgs :: INT → STRINGS → IO ()
tooManyCmdlineArgs n args = fatalError msg
  where
    msg =
      "Unexpected command-line argument " ++ show (args !! n) ++ "\n\
      \(" ++ expected ++ ")\n\
      \Use the -? option to see detailed synopsis."
    expected =
      if n == 0 then
        "no arguments expected"
      else if n == 1 then
        "only one argument expected"
      else
        "only " ++ show n ++ " arguments expected"

invalidCmdlineFlag :: STRING → IO ()
invalidCmdlineFlag f = fatalError msg
  where
    msg =
      "Invalid command-line flag " ++ f' ++ "\n\
      \Use the -? flag to see detailed synopsis."
    f' = '-' : drop 2 (take (length f'' - 1) f'')
    f'' = show f

missingCmdlineFlagArg :: STRING → IO ()
missingCmdlineFlagArg f = fatalError msg
  where
    msg =
      "Command-line flag " ++ f ++ " requires an argument.\n\
      \Use the -? flag to see detailed synopsis."

incompatibleCmdlineFlags :: STRING → STRING → IO ()
incompatibleCmdlineFlags e f = fatalError msg
  where
    msg =
      "Command-line flags " ++ e ++ " and " ++ f ++
           " are mutually exclusive.\n\
      \Use the -? flag to see detailed synopsis."

repeatedCmdlineFlag :: STRING → IO ()
repeatedCmdlineFlag f = fatalError msg
  where
    msg =
      "Command-line flag " ++ f ++ " cannot be specified more than once.\n\
      \Use the -? flag to see detailed synopsis."
```