This is the main driver-program entry point for Corrode.

It brings together the C parser and preprocessor interface from language-c, Corrode's algorithms, and the pretty-printing library pretty, all while reporting errors in a consistent way.

```
import Control.Monad
import Control.Monad.Trans.Class
import Control.Monad.Trans.Except
import Data.List
import Language.C
import Language.C.System.GCC
import Language.C.System.Preprocess
import Language.Rust.Corrode.C
import Language.Rust.Corrode.CrateMap
import Language.Rust.Idiomatic
import System.Environment
import System.Exit
import System.FilePath
import Text.PrettyPrint.HughesPJClass
```

Corrode can produce reasonable single-module output using only the information that you would have passed to a C compiler. But with some guidance from the user, it can produce better output. Here we remove Corrode-specific command-line options; the rest will be passed to GCC.

```
newtype Options = Options
    { moduleMaps :: [(String, String)]
    }

defaultOptions :: Options
defaultOptions = Options
    { moduleMaps = []
    }

parseCorrodeArgs :: [String] -> Either String (Options, [String])
parseCorrodeArgs ("-corrode-module-map" : spec : rest) = do
    let spec' = case span (/= ':') spec of
            (crate, _ : specFile) -> (crate, specFile)
            (specFile, []) -> ("", specFile)
    (opts, other) <- parseCorrodeArgs rest
    return (opts { moduleMaps = spec' : moduleMaps opts }, other)
parseCorrodeArgs (arg : rest) = do
    (opts, other) <- parseCorrodeArgs rest
    return (opts, arg : other)
parseCorrodeArgs [] = return (defaultOptions, [])
```

There are lots of steps in this process, and several of them return an `Either`, which is used similarly to Rust's `Result` type. In Haskell we don't have the `try!` macro that Rust has; instead the `ExceptT` monad encapsulates the "return early on failure" pattern.

We layer `ExceptT` on top of the `IO` monad so that we're permitted to access files and command-line arguments, but we need to adapt various types of return values from different functions we'll call. For a function which performs pure computation and might fail, wrap the function call in `try`. If the function can also do I/O, wrap it in `tryIO` instead.

```
try :: Either e a -> ExceptT e IO a
try = tryIO . return
```

```
tryIO :: IO (Either e a) -> ExceptT e IO a
tryIO = ExceptT
```

We use one other function for dealing with errors. `withExceptT f` applies `f` to the error value, if there is one, which lets us convert different types of errors to one common error type.

Here's the pipeline:

```
main :: IO ()
main = dieOnError $ do
```

1. Extract the command-line arguments we care about. We'll pass the rest to the preprocessor.

```
let cc = newGCC "gcc"
cmdline <- lift getArgs
(options, cmdline') <- try (parseCorrodeArgs cmdline)
(rawArgs, _other) <- try (parseCPPArgs cc cmdline')
```

2. The user may have specified the `-o <outputfile>` option. Not only do we ignore that, but we need to suppress it so the preprocessor doesn't write its output where a binary was expected to be written. We also force-undefine preprocessor symbols that would indicate support for language features we can't actually handle, and remove optimization flags that make GCC define preprocessor symbols.

```
let defines = [Define "_FORTIFY_SOURCE" "0", Define "__NO_CTYPE" "1"]
let undefines = map Undefine ["__BLOCKS__", "__FILE__", "__LINE__"]
let warnings = ["-Wno-builtin-macro-redefined"]
let args = foldl addCppOption
        (rawArgs
            { outputFile = Nothing
            , extraOptions =
                  (filter (not . ("-O" `isPrefixOf`)) (extraOptions rawArgs)) ++
                  warnings
            })
        (defines ++ undefines)
```

3. Load any specified module-maps.

```
allMaps <- fmap mergeCrateMaps $ forM (moduleMaps options) $
    \ (crate, filename) -> tryIO $ do
        spec <- readFile filename
        return $ do
            crateMap <- parseCrateMap spec
            return (crate, crateMap)
let modName = takeBaseName (inputFile args)
let (currentModule, otherModules) = splitModuleMap modName allMaps
let allRewrites = rewritesFromCratesMap otherModules
```

4. Run the preprocessor—except that if the input appears to have already been preprocessed, then we should just read it as-is.

```
input <- if isPreprocessed (inputFile args)
    then lift (readInputStream (inputFile args))
    else withExceptT
        (\ status -> "Preprocessor failed with status " ++ show status)
        (tryIO (runPreprocessor cc args))
```

5. Get language-c to parse the preprocessed source to a `CTranslUnit`.

```
unit <- withExceptT show (try (parseC input (initPos (inputFile args))))
```

6. Generate a list of Rust items from this C translation unit.

```
items <- try (interpretTranslationUnit currentModule allRewrites unit)
```

7. Pretty-print all the items as a `String`.

```
let output = intercalate "\n"
        [ prettyShow (itemIdioms item) ++ "\n"
        | item <- items
        ]
```

8. Write the final string to a file with the same name as the input, except with any extension replaced by ".rs".

```
let rsfile = replaceExtension (inputFile args) ".rs"
lift $ do
    writeFile rsfile output
    putStrLn rsfile
    putStrLn $ case outputFile rawArgs of
        Just outfile -> outfile
        Nothing -> replaceExtension (inputFile args) ".o"
```

When the pipeline ends, we need to check whether it resulted in an error. If so, we call `die` to print the error message to `stderr` and exit with a failure status code.

```
dieOnError :: ExceptT String IO a -> IO a
dieOnError m = do
    result <- runExceptT m
    case result of
        Left err -> die err
        Right a -> return a
```