prfchk: Proof Checking for Haskell (v0.9.0)

Andrew Butterfield

November 20, 2018

Contents

Chapter 1

prfckh Application

```
module Main where

import Data.List
import Data.Maybe

import System.Environment
import System.Directory
import System.FilePath
import Control.Exception

import Utilities
import REPL
import AST
import Matching
import HParse
import Theory
import Check

version = "0.9.0.0"

main :: IO ()
main
 = do args <- getArgs
      case args of
        [] ->  repl
        [nm]  -> batch nm
        _ -> putStrLn
               $ unlines [ "usage: prfchk [name]"
                         , " name is of .thr file in /data"
                         , "If no name given, the command line interface runs"
                         , "if name is given,"
                         , " that theory is loaded and all theorems are checked"
                         ]

batch :: String -> IO ()
batch nm
  = do hreqs <- loadTheory [nm] hreqs0
       case currThry hreqs of
         Nothing    -> putStrLn ( "Failed to load data/"++nm++".thr")
         Just thry
           -> do let hms = hmods hreqs
```

```
                 let hth = hthrys hreqs ++ [thry]
                 putStrLn "STARTING BATCH CHECK...\n"
                 sequence_ $ map (showReport  . checkTheorem hms hth) $ thTheorems
                 putStrLn "\nFINISHED BATCH CHECK"

repl :: IO ()
repl
  = do runREPL hreqWelcome hreqConfig hreqs0
       return ()

hreqWelcome = unlines
 [ "Welcome to Proof Check v"++version
 , "To run in batch mode, give name of theory file when invoking from shell."
 , "Type '?' for help."
 ]

hreqConfig
  = REPLC
      hreqPrompt
      hreqEOFreplacmement
      hreqParser
      hreqQuitCmds
      hreqQuit
      hreqHelpCmds
      hreqCommands
      hreqEndCondition
      hreqEndTidy


data HReqState
  = HReq { hmods :: [Mdl]
         , hthrys :: [Theory]
         , currThry :: Maybe Theory
         }
  deriving Show

hmods__ f hrs = hrs{ hmods = f $ hmods hrs} ; hmods_ h = hmods__ $ const h
hthrys__ f hrs = hrs{ hthrys = f $ hthrys hrs} ; hthrys_ h = hthrys__ $ const h
currThry__ f hrs = hrs{ currThry = f $ currThry hrs}
currThry_ h = currThry__ $ const h

hreqs0 = HReq [] [] Nothing

type HReqCmd       =  REPLCmd      HReqState
type HReqCmdDescr  =  REPLCmdDescr HReqState
type HReqExit      =  REPLExit     HReqState
type HReqCommands  =  REPLCommands HReqState
type HReqConfig    =  REPLConfig   HReqState

hreqPrompt :: Bool -> HReqState -> String
hreqPrompt _ _ = "prfchk> "

hreqEOFreplacmement = [nquit]

hreqParser = wordParse

hreqQuitCmds = [nquit] ; nquit = "q"
```

```
hreqQuit :: HReqExit
hreqQuit _ hreqs = putStrLn "\nGoodbye!\n" >> return (True, hreqs)

hreqHelpCmds = ["?"]


-- we don't use these features in the top-level REPL
hreqEndCondition _ = False
hreqEndTidy _ hreqs = return hreqs

hreqCommands :: HReqCommands
hreqCommands = [ cmdShowState
               -- , showTheoryFiles
               , cmdShowLaws
               -- , cmdLoadHaskell -- deprecated for now.
               , cmdLoadTheory
               , cmdCheckTheorem
               -- , cmdParseHaskell
               ]

cmdShowState :: HReqCmdDescr
cmdShowState
  = ( "state"
    , "show state"
    , "show short summary of state contents"
    , showState )

showState _ hreqs
  = do showHModNames   $ hmods    hreqs
       showTheoryNames $ hthrys   hreqs
       showCurrThry    $ currThry hreqs
       return hreqs

showHModNames [] = putStrLn "No Haskell Modules"
showHModNames hms = putStrLn ("Haskell Modules: " ++ shlist (map mname hms))

showTheoryNames [] = putStrLn "No Required Theories"
showTheoryNames thrys
  = putStrLn ("Required Theories: "++ shlist (map theoryName thrys))

showCurrThry Nothing = putStrLn "\nNo Current Theory"
showCurrThry (Just thry) = putStrLn ("\nCurrent Theory: "++theoryName thry)

shlist strs = intercalate ", " strs

showTheoryFiles :: HReqCmdDescr
showTheoryFiles
  = ( "tf"
    , "show theory files"
    , "show list of *.thr in /data."
    , showTFiles )

showTFiles _ hreq
  = do listing <- getDirectoryContents "./data"
       let thrFiles = filter isThr listing
       putStrLn $ unlines thrFiles
```

```
       return hreq

isThr fp = takeExtension fp == ".thr"

cmdShowLaws :: HReqCmdDescr
cmdShowLaws
  = ( "laws"
    , "'law' names"
    , "show all law and definition names"
    , showLaws )

showLaws _ hreqs
  = do sequence_ $ map showHModLaws $ hmods hreqs
       putStrLn ""
       sequence_ $ map showTheoryLaws $ hthrys hreqs
       putStrLn ""
       case currThry hreqs of
         Nothing    -> putStrLn "No Current Theory"
         Just thry  -> do showTheoryLaws thry
                          showTheorems thry
       return hreqs

showHModLaws hmod
 = do putStrLn ("Laws in Haskell source '"++mname hmod++"'")
      sequence_ $ map showDecl $ topdecls hmod

showDecl (Fun []) = putStrLn "  !dud function definition!"
showDecl (Fun (m:_))  =  putStrLn ("  " ++ fname m)
showDecl (Bind (Var n) _ _) = putStrLn ("  " ++ n)
showDecl _ = putStrLn "  ??"

showTheoryLaws thry
  = do putStrLn ("Laws in Theory '"++theoryName thry++"'")
       sequence_ $ map showLaw $ thLaws thry

showLaw law = putStrLn ("  "++ lawName law)

showTheorems thry
  = do putStrLn ("Theorems in Theory '"++theoryName thry++"'")
       sequence_ $ map showTheorem $ thTheorems thry

showTheorem thrm = putStrLn ("  "++ thmName thrm)

-- deprecated for now
cmdLoadHaskell :: HReqCmdDescr
cmdLoadHaskell
  = ( "lh"
    , "load Haskell source"
    , unlines
        [ "lh <fname>  -- parse and dump AST for data/<fname>.hs"
        ]
    , loadSource )

loadSource [] hreqs = putStrLn "no file given" >> return hreqs
loadSource (fnroot:_) hreqs
  = do  mdl <- readHaskell fnroot
        putStrLn "Module AST:\n"
```

```
        let aststr = show mdl
        putStrLn aststr
        writeFile ("data/"++fnroot++".ast") aststr
        -- return $ hmods__ (++[mdl]) hreqs
        return hreqs

readHaskell fnroot
  = do let fname = fnroot ++ ".hs"
       modstr <- readFile ("data/"++fname)
       parseHModule fname modstr

cmdParseHaskell :: HReqCmdDescr
cmdParseHaskell
  = ( "ph"
    , "parse Haskell"
    , "ph <haskell-expr> -- parse haskell expression on command line"
    , parseHaskell )

parseHaskell args hreqs
 = do case hParseE (ParseMode "ph") [] [(1,estr)] of
        But msgs -> putStrLn $ unlines msgs
        Yes (hsexp,_)
          -> do putStrLn "haskell-src parse:"
                putStrLn $ show hsexp
                let expr = hsExp2Expr preludeFixTab hsexp
                putStrLn "simple AST version:"
                putStrLn $ show expr
      return hreqs
 where estr = unwords args


cmdLoadTheory :: HReqCmdDescr
cmdLoadTheory
  = ( "load"
    , "load Theory source"
    , unlines
        [ "load <fname>  -- load data/<fname>.thr"
        , " -- also loads all haskell modules and theories that it imports"
        ]
    , loadTheory )

loadTheory [] hreqs = putStrLn "no file given" >> return hreqs
loadTheory (fnroot:_) hreqs
  = do res <- readTheory fnroot
       case res of
         Nothing -> return hreqs
         Just theory
           -> do putStrLn ("\nLoaded Theory '"++fnroot++"'")
                 loadDependencies theory hreqs



readTheory fnroot
  = do let fname = fnroot ++ ".thr"
       thrystr <- readFile ("data/"++fname)
       case parseTheory (ParseMode fname) thrystr of
         But msgs  ->  do putStrLn $ unlines msgs
```

```
                          return Nothing
         Yes thry  ->  return $ Just thry

loadDependencies theory hreqs
  = do hms <- loadModDeps $ hkImports theory
       ths <- loadThryDeps $ thImports theory
       putStrLn "Theory dependencies loaded.\n"
       return $ currThry_ (Just theory)
              $ hthrys_ ths
              $ hmods_ hms
              $ hreqs

loadModDeps []  = return []
loadModDeps (n:ns)
  = do m <- readHaskell n
       ms <- loadModDeps ns
       return (m:ms)

loadThryDeps [] = return []
loadThryDeps (t:ts)
  = do res <- readTheory t
       case res of
         Nothing  -> loadThryDeps ts
         Just thry -> do thrys <- loadThryDeps ts
                         return (thry:thrys)

cmdCheckTheorem :: HReqCmdDescr
cmdCheckTheorem
  = ( "check"
    , "check theorem"
    , "check <name> -- check theorem called name"
    , theoremCheck )

theoremCheck [] hreqs
  = do putStrLn "no theorem specified"
       return hreqs

theoremCheck (n:_) hreqs
  = do case currThry hreqs of
         Nothing
           ->  putStrLn "no current theory"
         Just thry
           ->  case findTheorem n $ thTheorems thry of
                 Nothing   ->  putStrLn ("Theorem not found: "++n)
                 Just thm  ->  showReport $
                     checkTheorem (hmods hreqs) (hthrys hreqs ++ [thry]) thm
       return hreqs
```

Chapter 2

prfchk Libraries

2.1 Haskell Parser

```
Copyright  Andrew Butterfield (c) 2017--18

LICENSE: BSD3, see file LICENSE at reasonEq root
```

```
module HParse
( Line, Lines, Parser
, parseHModule
, parseExpr, hParseE
, parseEqual
, hs42
, ParseMode(..), ParseResult(..), SrcLoc(..), pFail
)
where

import Data.Char
import qualified Data.Map as M

import Language.Haskell.Parser
import Language.Haskell.Pretty
import Language.Haskell.Syntax

import Utilities
import AST

import Debug.Trace
dbg msg x = trace (msg ++ show x) x
mdbg msg x = return $! dbg msg x
```

2.1.1 Monadic Failure

A polymorphic, monadic parser type:

```
type Line = (Int,String)
type Lines = [Line]
type Parser m a  = Lines -> m (a,Lines)
```

A SrcLoc-based monadic failure:

```
pFail :: Monad m => ParseMode -> Int -> Int -> String -> m a
pFail pmode lno colno msg
  = fail (parseFilename pmode ++ ':':show lno++ ":"++show colno++" "++msg)
```

2.1.2 Parser Top-Level

```
parseHModule :: Monad m => String -> String -> m Mdl
parseHModule fname modstr
 = case parseModuleWithMode pmode modstr of
     ParseFailed loc msg -> pFail pmode (srcLine loc) (srcColumn loc) msg
     ParseOk hsmod -> return $ hsModule2Mdl hsmod
 where pmode = ParseMode fname
```

2.1.3 Parsing Expressions

```
parseExpr :: Monad m => ParseMode -> Lines -> Parser m Expr
parseExpr pmode restlns chunk
 = do (hsexp,lns') <- hParseE pmode restlns chunk
      return (hsExp2Expr preludeFixTab hsexp,lns')

hParseE :: Monad m => ParseMode -> Lines -> Parser m HsExp
hParseE pmode restlns [] = pFail pmode 0 0 "no expression!"
hParseE pmode restlns chunk@((lno,_):_)
  = case parseModuleWithMode pmode (mkNakedExprModule chunk) of
      ParseFailed _ msg  -> pFail pmode lno 1 msg
      ParseOk hsmod -> do hsexp <- getNakedExpr hsmod
                          return (hsexp, restlns)
```

```
mkNakedExprModule [(_,str)]
  = unlines [ "module NakedExpr where"
            , "nakedExpr = "++str ]
mkNakedExprModule chunk
  = unlines ( [ "module NakedExpr where"
              , "nakedExpr = " ]
              ++ map snd chunk )
```

```
getNakedExpr :: Monad m => HsModule -> m HsExp
getNakedExpr
 (HsModule _ _ _ _ [ HsPatBind _ _ (HsUnGuardedRhs hsexp) [] ])
    = return hsexp
getNakedExpr _ = fail "can't find the naked expression"

hs42 = LInt 42
```

2.1.4 Parsing Equivalences

```
parseEqual :: Monad m => ParseMode -> Lines -> Parser m (Expr, Expr)
parseEqual pmode restlns [] = pFail pmode 0 0 "no equivalence!"
parseEqual pmode restlns chunk@((lno,_):_)
  = case parseModuleWithMode pmode (mkNakedExprModule chunk) of
      ParseFailed _ msg  -> pFail pmode lno 1 msg
      ParseOk hsmod -> return (getNakedEqual hsmod, restlns)
```

```
getNakedEqual :: HsModule -> (Expr,Expr)
getNakedEqual
 (HsModule _ _ _ _ [ _, HsPatBind _ _ (HsUnGuardedRhs hsexp) [] ])
   = case hsexp of
       (HsInfixApp e1 (HsQVarOp (UnQual (HsSymbol "=="))) e2)
          -> ( hsExp2Expr preludeFixTab e1
             , hsExp2Expr preludeFixTab e2)
       _                -> (hs42,hs42)
getNakedEqual _  =   (hs42,hs42)
```

2.2 Abstract Syntax Tree

```
Copyright  Andrew Butterfield (c) 2017--18

LICENSE: BSD3, see file LICENSE at reasonEq root
```

```
{-# LANGUAGE PatternSynonyms #-}
module AST
(
  Expr(..), Match(..), Decl(..), Mdl(..), FixTab
, hsModule2Mdl, hsDecl2Decl, hsExp2Expr
, pattern InfixApp, pattern Equal
  -- special variables:
, eNull, eCons
, eEq
, pWild, pAs
, preludeFixTab
)
where

import Language.Haskell.Parser
import Language.Haskell.Pretty
import Language.Haskell.Syntax

import Data.Map (Map)
import qualified Data.Map as M

import Debug.Trace
dbg msg x = trace (msg ++ show x) x
mdbg msg x = return $! dbg msg x
```

We need a simplified AST for haskell. We don't need any source-locs, and we really don't need to distinguish identifiers from symbols, handle qualified names, or treat patterns differently to general expressions. Wildcard patterns (or even irrefutable ones) can be handled using names.

2.2.1 Simplified Haskell AST

We simplify things dramatically. First, expressions:

```
data Expr
  = LBool Bool | LInt Int | LChar Char
  | Var String
  | App Expr Expr
  | If Expr Expr Expr
  | GrdExpr [(Expr,Expr)]
  | Let [Decl] Expr
  | PApp String [Expr]
  deriving (Eq,Show)
```

Next, matchings:

```
data Match = Match { fname ::  String  -- function name
                   , lhspat :: [Expr]  -- LHS patterns
                   , rhs :: Expr     -- RHS outcome
                   , ldecls :: [Decl]  -- local declarations
                   }
           deriving (Eq, Show)
```

Then, declarations:

```
data Decl
  = Fun [Match]
  | Bind Expr Expr [Decl]
  | Fixity String Int Assoc
  | Type String -- just noting name for now - to be addressed later
  deriving (Eq, Show)
```

Associativity: left, right or none:

```
data Assoc = ANone | ALeft | ARight deriving (Eq,Show)
```

We want to be able to record fixity information:

```
type FixTab = Map String (Int,Assoc)
```

Finally, modules (ignoring exports)

```
data Mdl = Mdl { mname :: String
               , imps :: [Import]
               , topdecls :: [Decl]
               }
         deriving Show

data Import = Imp { imname :: String
                  , asnmame :: Maybe String
                  }
            deriving Show
```

2.2.2 Simplifying Strings

```
hsName2Str :: HsName -> String
hsName2Str (HsIdent str)  = str
hsName2Str (HsSymbol str) = str

hsOpName :: HsOp -> String
hsOpName (HsVarOp hn) = hsName2Str hn
hsOpName (HsConOp hn) = hsName2Str hn

hsSpcCon2Str :: HsSpecialCon -> String
hsSpcCon2Str HsUnitCon  =  "()"
hsSpcCon2Str HsListCon  =  "[]"
hsSpcCon2Str HsFunCon   =  "->"
hsSpcCon2Str HsCons     =  ":"
hsSpcCon2Str (HsTupleCon i)  = "("++replicate (i-1) ','++")"

hsQName2Str :: HsQName -> String
hsQName2Str (Qual (Module m) nm) = m ++ '.':hsName2Str nm
hsQName2Str (UnQual nm) = hsName2Str nm
hsQName2Str (Special hsc) = hsSpcCon2Str hsc

hsQOp2Str :: HsQOp -> String
```

```
hsQOp2Str (HsQVarOp hsq)  = hsQName2Str hsq
hsQOp2Str (HsQConOp hsq)  = hsQName2Str hsq

hsExp2Str :: HsExp -> String
hsExp2Str (HsVar qnm)  = hsQName2Str qnm
hsExp2Str (HsCon qnm)  = hsQName2Str qnm
hsExp2Str hse = error ("hsExp2Str invalid for "++show hse)

hsPat2Str :: HsPat -> String
hsPat2Str (HsPVar pnm) = hsName2Str pnm
hsPat2Str hsp = error ("hsPat2Str invalid for "++show hsp)
```

2.2.3 Simplifying Literals

```
hsLit2Expr :: HsLiteral -> Expr
hsLit2Expr (HsInt i)  = LInt $ fromInteger i
hsLit2Expr (HsChar c) = LChar c
hsLit2Expr lit = error ("hsLit2Expr NYIf "++show lit)
```

2.2.4 Simplifying Parsed Expressions

```
hsExp2Expr :: FixTab -> HsExp -> Expr
hsExp2Expr _ (HsVar hsq)  =  Var $ hsQName2Str hsq
hsExp2Expr _ (HsCon hsq)  =  Var $ hsQName2Str hsq
hsExp2Expr _ (HsLit lit)  =  hsLit2Expr lit
hsExp2Expr fixtab iapp@(HsInfixApp _ _ _)  =  hsInfix2Expr fixtab iapp
hsExp2Expr ftab (HsApp e1 e2)
  =  App (hsExp2Expr ftab e1) (hsExp2Expr ftab e2)
hsExp2Expr ftab (HsIf hse1 hse2 hse3)
  = If (hsExp2Expr ftab hse1) (hsExp2Expr ftab hse2) (hsExp2Expr ftab hse2)
hsExp2Expr fixtab (HsParen hse)  =  hsExp2Expr fixtab hse
hsExp2Expr ftab (HsList hses)  =  hsExps2Expr ftab hses
hsExp2Expr _ hse  =  error ("hsExp2Expr NYIf "++show hse)
```

We want to match and build infix operators as simple unary applications:

```
eEq = Var "=="
pattern InfixApp e1 op e2 = App (App (Var op) e1) e2
pattern Equal e1 e2       = App (App (Var "==") e1) e2
```

```
eNull = Var "[]"
eCons = Var ":"
hsExps2Expr :: FixTab -> [HsExp] -> Expr
hsExps2Expr _ []            =  eNull
hsExps2Expr ftab (hse:hses)
  =  InfixApp (hsExp2Expr ftab hse) ":" (hsExps2Expr ftab hses)
```

Fixing Infix Parses

The `haskell-src` package does a very lazy parsing of infix operators that ignores operator precedence and treats every operator as left-associative. So

$$\mathtt{e} = e_1 \otimes_1 e_2 \otimes_2 e_3 \otimes_3 \cdots \otimes_{n-2} e_{n-1} \otimes_{n-1} e_n$$

where e_1 is not an infix application, parses as[1]

$$e_? = (\ldots((e_1 \otimes_1 e_2) \otimes_2 e_3) \otimes_3 \cdots \otimes_{n-2} e_{n-1}) \otimes_{n-1} e_n$$

This needs to be fixed. It also explains why there is a `HsParen` constructor in `HsExp`!

The first consequence is that the second argument for each operator can be independently converted, while the longest chain formed as long as first arguments are infix operators needs special handling. Let the function converting `HsExp` h into `Expr` a be denoted by $[\![]\!]$, so that $a = [\![h]\!]$. So the example above should first be transformed into two lists as follows:

$$
\begin{array}{ll}
 & [\![e_?]\!] \\
= & \text{“expand } e_?\text{”} \\
 & [\![(\ldots((e_1 \otimes_1 e_2) \otimes_2 e_3) \otimes_3 \cdots \otimes_{n-2} e_{n-1}) \otimes_{n-1} e_n]\!] \\
= & \text{“2nd arguments convert independently”} \\
 & (\ldots(([\![e_1]\!] \otimes_1 [\![e_2]\!]) \otimes_2 [\![e_3]\!]) \otimes_3 \cdots \otimes_{n-2} [\![e_{n-1}]\!]) \otimes_{n-1} [\![e_n]\!] \\
= & \text{“split out longest 1st-argument chain of operators”} \\
 & (\ \langle \otimes_1, \otimes_2, \otimes_3 \cdots \otimes_{n-2}, \otimes_{n-1} \rangle\ ,\ \langle [\![e_1]\!], [\![e_2]\!], [\![e_3]\!], \ldots, [\![e_{n-1}]\!], [\![e_n]\!] \rangle\ ) \\
= & \text{“fuse adjacent terms of operators into sub-expression, highest precedence first.”} \\
 & \text{“top-level list will be shorter, with lowest precedence operators”} \\
 & (\ \langle \otimes_a, \otimes_b, \otimes_3 \cdots \otimes_x, \otimes_y \rangle\ ,\ \langle [\![e_a]\!], [\![e_b]\!], [\![e_c]\!], \ldots, [\![e_y]\!], [\![e_z]\!] \rangle\ ) \\
= & \text{“bottom-up, for each right-associate operator, twist the tree.”} \\
 & e \quad \text{“ — the true form of } \mathtt{e}\text{”}
\end{array}
$$

We will describe “tree-twisting” below.

```haskell
hsInfix2Expr :: FixTab -> HsExp -> Expr
-- this is usually called with iapp being a HsInfixApp
hsInfix2Expr fixtab iapp
 =  e
 where
   (ops,es) = split fixtab iapp
   prcf = fst . readFixTab fixtab
   (ops',es') = pfusing prcf 9 (ops,es)
   assf = snd . readFixTab fixtab
   e = twist prcf assf $ head $ es' -- won't be empty
```

[1] So `x:y:z:[]` parses as $((x : y) : z) : []$!

Split We use `split` to perform the 2nd argument conversion and splitting

$$
\begin{aligned}
& [\![(\dots((e_1 \otimes_1 e_2) \otimes_2 e_3) \otimes_3 \dots \otimes_{n-2} e_{n-1}) \otimes_{n-1} e_n]\!] \\
= \; & \text{“2nd arguments convert independently”} \\
& (\dots(([\![e_1]\!] \otimes_1 [\![e_2]\!]) \otimes_2 [\![e_3]\!]) \otimes_3 \dots \otimes_{n-2} [\![e_{n-1}]\!]) \otimes_{n-1} [\![e_n]\!] \\
= \; & \text{“split out longest 1st-argument chain of operators”} \\
& (\ \langle \otimes_1, \otimes_2, \otimes_3 \cdots \otimes_{n-2}, \otimes_{n-1} \rangle\ ,\ \langle [\![e_1]\!], [\![e_2]\!], [\![e_3]\!], \dots, [\![e_{n-1}]\!], [\![e_n]\!] \rangle\ )
\end{aligned}
$$

```
split :: FixTab -> HsExp -> ( [String], [Expr] )

-- split (B e1 op e2) = ( ops ++ [op] , es ++ [e2]) where (ops,es) = split e1
-- !!!!! if hse1 is a HsParen, then we might need to leave it alone!!!
split ftab (HsInfixApp hse1 hsop hse2)
  = (ops++[op],es++[hsExp2Expr ftab hse2])
  where
    op        =  hsQOp2Str hsop
    (ops,es)  =  split ftab hse1

-- split a@(A _) = ( [], [a] )
-- !!!! if hsexp is HsParen then that is stripped off !!!
split ftab hsexp = ([],[hsExp2Expr ftab hsexp])
```

Fuse We then proceed to fuse together operators of highest precedence with their neighbouring expressions, and keep repeating until the lowest precedence have themselves been fused.

$$
\begin{aligned}
& (\ \langle \otimes_1, \otimes_2, \otimes_3 \cdots \otimes_{n-2}, \otimes_{n-1} \rangle\ ,\ \langle [\![e_1]\!], [\![e_2]\!], [\![e_3]\!], \dots, [\![e_{n-1}]\!], [\![e_n]\!] \rangle\ ) \\
= \; & \text{“fuse adjacent terms of operators into sub-expression, highest precedence first.”} \\
& \text{“top-level list will be shorter, with lowest precedence operators”} \\
& (\ \langle \otimes_a, \otimes_b, \otimes_3 \cdots \otimes_x, \otimes_y \rangle\ ,\ \langle [\![e_a]\!], [\![e_b]\!], [\![e_c]\!], \dots, [\![e_y]\!], [\![e_z]\!] \rangle\ )
\end{aligned}
$$

```
pfuse :: (String -> Int) -> Int -> [String] -> [Expr] -> ([String],[Expr])
pfuse _ p [] [e] = ([],[e])
pfuse prcf p [op] [e1,e2]
  | p == prcf op   =  ([],[InfixApp e1 op e2])
  | otherwise   =  ([op],[e1,e2])
pfuse prcf p (op:ops) (e1:e2:es)
  | p == prcf op   =  pfuse prcf p ops (InfixApp e1 op e2 : es)
  | otherwise      =  (op:ops',e1:es')
  where (ops',es') = pfuse prcf p ops (e2:es)

pfusing :: (String -> Int) -> Int -> ([String],[Expr]) -> ([String],[Expr])
pfusing _ (-1) oes = oes
pfusing prcf p (ops,es) = pfusing prcf (p-1) $ pfuse prcf p ops es
```

Twist We now get to the point were we look for trees built with right-associative operators, that will still be in left-associative form. We have to “twist” these trees into right-associative form. At the top-level, we process binary sub-expressions first, and then twist the top result.

```
-- -- we assume everything is left-infix to start.

-- InfixApp op e1 e2
twist :: (String -> Int) -> (String -> Assoc) -> Expr -> Expr
twist prcf assf (InfixApp e1 op e2)
  = twist' prcf assf (InfixApp (twist prcf assf e1) op (twist prcf assf e2))
twist prcf assf e = e

twist' prcf assf e@(InfixApp (InfixApp e1 op1 e2) op2 e3)
  | assf op1 == ARight && assf op2 == ARight && prcf op1 == prcf op2
    = InfixApp e1 op1 (insSE prcf assf op2 e2 e3 )
twist' _ _ e = e


insSE prcf assf op2 (InfixApp e4 op3 e5) e3
  | assf op3 == ARight && prcf op2 == prcf op3
    = InfixApp e4 op3 (insSE prcf assf op2 e5 e3)
insSE  _ _ op2 e2 e3 = InfixApp e2 op2 e3
```

For now, we view righthand-sides as expressions

```
hsRhs2Expr :: FixTab -> HsRhs -> Expr
hsRhs2Expr ftab (HsUnGuardedRhs hse)     =  hsExp2Expr ftab hse
hsRhs2Expr ftab (HsGuardedRhss grdrhss)
  =  GrdExpr $ map (hsGrdRHs2Expr2 ftab) grdrhss

hsGrdRHs2Expr2 :: FixTab -> HsGuardedRhs -> (Expr, Expr)
hsGrdRHs2Expr2 ftab (HsGuardedRhs _ grd rhs)
 = (hsExp2Expr ftab grd, hsExp2Expr ftab rhs)
```

For now, we view patterns as expressions

```
pWild = Var "_"
pAs   = Var "@"
hsPat2Expr :: HsPat -> Expr
hsPat2Expr (HsPVar hsn) = Var $ hsName2Str hsn
hsPat2Expr (HsPLit lit) = hsLit2Expr lit
hsPat2Expr (HsPList hspats) = hsPats2Expr hspats
hsPat2Expr (HsPParen hspat) = hsPat2Expr hspat
hsPat2Expr (HsPInfixApp p1 op p2)
  =  InfixApp (hsPat2Expr p1) (hsQName2Str op) (hsPat2Expr p2)
hsPat2Expr HsPWildCard = pWild
hsPat2Expr (HsPAsPat nm hspat)
 =  App (App pAs $ Var $ hsName2Str nm) $ hsPat2Expr hspat
hsPat2Expr (HsPApp qnm hspats)  = PApp (hsQName2Str qnm) $ map hsPat2Expr hspats

hsPat2Expr hsp = error ("hsPat2Expr NYIf "++show hsp)

hsPats2Expr :: [HsPat] -> Expr
hsPats2Expr []  = eNull
hsPats2Expr (hspat:hspats)
  = App (App eCons $ hsPat2Expr hspat) $ hsPats2Expr hspats
```

2.2.5 Simplifying Parsed Matches

```
hsMatch2Match :: FixTab -> HsMatch -> Match
hsMatch2Match fixtab (HsMatch _ nm pats rhs decls)
  = Match (hsName2Str nm)
          (map hsPat2Expr pats)
          (hsRhs2Expr fixtab rhs)
          (map (hsDecl2Decl fixtab) decls)
```

2.2.6 Simplifying Parsed Declarations

```
hsDecl2Decl :: FixTab -> HsDecl -> Decl
hsDecl2Decl fixtab (HsFunBind hsMatches)
  = Fun $ map (hsMatch2Match fixtab) hsMatches

hsDecl2Decl ftab (HsPatBind _ hspat hsrhs hsdecls)
 = Bind (hsPat2Expr hspat)
        (hsRhs2Expr ftab hsrhs)
        (map (hsDecl2Decl ftab) hsdecls)

-- ignore type signatures and declarations for now, just note name
hsDecl2Decl fixtab (HsTypeSig _ hsn _)        = Type "::"
hsDecl2Decl fixtab (HsTypeDecl _ hsn _ _) = Type $ hsName2Str hsn
hsDecl2Decl fixtab (HsDataDecl _ _ hsn _ _ _) = Type $ hsName2Str hsn
hsDecl2Decl fixtab(HsNewTypeDecl _ _ hsn _ _ _) = Type $ hsName2Str hsn

hsDecl2Decl fixtab (HsInfixDecl _ assoc p [op])
  = Fixity (hsOpName op) p (hsAssoc2Assoc assoc)
hsDecl2Decl fixtab hsd = error ("hsDecl2Decl NYIf "++show hsd)
```

```
hsAssoc2Assoc :: HsAssoc     ->  Assoc
hsAssoc2Assoc HsAssocNone    =   ANone
hsAssoc2Assoc HsAssocLeft    =   ALeft
hsAssoc2Assoc HsAssocRight   =   ARight
```

2.2.7 Simplifying Parsed Modules

```
hsModule2Mdl :: HsModule -> Mdl
hsModule2Mdl (HsModule _ (Module nm) _ imports decls)
  = Mdl nm
        (map hsImpDcl2Imp imports)
        (map (hsDecl2Decl fixtab) decls)
  where fixtab = buildFixTab preludeFixTab decls

hsImpDcl2Imp :: HsImportDecl -> Import
hsImpDcl2Imp hsID
 = Imp (hsMod2Str $ importModule hsID)
       (hsModAs2MStr $ importAs hsID)

hsMod2Str :: Module -> String
hsMod2Str (Module str) = str

hsModAs2MStr :: Maybe Module -> Maybe String
hsModAs2MStr Nothing = Nothing
hsModAs2MStr (Just m) = Just $ hsMod2Str m
```

2.2.8 Fixity Handling

Building a fixity table on top of a pre-existing table.

```
buildFixTab :: FixTab -> [HsDecl] -> FixTab
buildFixTab fixtab []  = fixtab
buildFixTab fixtab (HsInfixDecl _ assoc p [op] : decls)
  =  buildFixTab (M.insert (hsOpName op) (p,hsAssoc2Assoc assoc) fixtab) decls
buildFixTab fixtab (_ : decls)
  =  buildFixTab fixtab decls
```

Looking up a fixity table:

```
readFixTab :: FixTab -> String -> (Int,Assoc)
readFixTab fixtab op
  =  case M.lookup op fixtab of
       Nothing   ->  (9,ALeft) -- see H2010 Report,  4.4.2
       Just res  ->  res
```

Prelude Fixity Declarations

We need to setup the Prelude fixities:

```
preludeFixTab
 = M.fromList
       [ ("!!",(9,ALeft))  -- infixl 9  !!
       , (".",(9,ARight))  -- infixr 9  .

         -- infixr 8  ^, ^^, **
       , ("^",(8,ARight)), ("^^",(8,ARight)), ("**",(8,ARight))

         -- infixl 7  *, /, `quot`, `rem`, `div`, `mod`
       , ("*",(7,ALeft)), ("/",(7,ALeft))
       , ("quot",(7,ALeft)), ("rem",(7,ALeft))
       , ("div",(7,ALeft)), ("mod",(7,ALeft))

       , ("+",(6,ALeft)), ("-",(6,ALeft))     -- infixl 6  +, -

       , (":",(5,ARight))  -- infixr 5  :
       , ("++",(5,ALeft))  -- infixl 5  ++

         -- infix  4  ==, /=, <, <=, >=, >, `elem`, `notElem`
       , ("==",(4,ANone)), ("/=",(4,ANone))
       , ("<",(4,ANone)), ("<=",(4,ANone)), (">=",(4,ANone)), (">",(4,ANone))
       , ("elem",(4,ANone)), ("notElem",(4,ANone))

       , ("&&",(3,ARight))                     -- infixr 3  &&
       , ("||",(2,ARight))                     -- infixr 2  ||
       , (">>",(1,ALeft)), (">>=",(1,ALeft))   -- infixl 1  >>, >>=
       , ("=<<",(1,ARight))                    -- infixr 1  =<<

         -- infixr 0  $, $!, `seq`
       , ("$",(0,ARight)), ("$!",(0,ARight)), ("seq",(0,ARight))
       ]
```

2.3 Theory

```
Copyright  Andrew Butterfield (c) 2017--18

LICENSE: BSD3, see file LICENSE at reasonEq root
```

```
module Theory
( Theory(..), parseTheory
, Theorem(..), findTheorem
, Law(..), InductionScheme(..)
, Strategy(..), Calculation(..)
, Justification(..), JRel(..), JLaw(..), Usage(..), Focus(..)
)
where

import Data.Char

import Utilities
import AST
import HParse

import Debug.Trace
dbg msg x = trace (msg ++ show x) x
mdbg msg x = return $! dbg msg x
```

2.3.1 Theory Document Structure

Typically a keyword at the start of a line introduces something. We start with `THEORY` and zero or more imports:

```
THEORY <TheoryName>
IMPORT-THEORY <Name>
IMPORT-HASKELL <Name>
```

These are followed by zero or more entries that describe laws, induction schemes and theorems.

Laws are described by the following "one-liner" construct:

```
LAW <name> <br?> <expr>
```

Here, `<br?>` means that the following part is either entirely on this line, or else occupies a number of subsequent lines. There can be a blank line before it, and must be a blank line after it. The following part itself must not have blank lines embedded in it.

An induction-scheme is described by the following four lines:

```
INDUCTION-SCHEME <Type>
BASE <value>
STEP <var> --> <expr>
INJ <br?> <expr> == <expr>
```

A theorem has the following top-level structure:

```
THEOREM <name> <br?> <expr>
STRATEGY <strategy>
<strategy-body>
QED <name>
```

Strategies include:

```
ReduceAll
ReduceLHS
ReduceRHS
ReduceBoth
Induction<ind-var> ::  <type>
```

The choice of strategy will then determine the resulting structure:

ReduceAll `<calculation>`

ReduceLHS `<calculation>`

ReduceRHS `<calculation>`

ReduceBoth

```
LHS
<calculation>
RHS
<calculation>
```

Induction

```
BASE <val> <br!> <expr>
<one of the other four strategies>
QED BASE
STEP <expr>
ASSUME <br?> <expr>
SHOW <br?> <expr>
<one of the other four strategies>
QED STEP
```

Here, `<br!>` is similar to `<br?>`, except that a line break at this point is mandatory.

A calculation is a sequence of formulæ seperated by justification lines, which always start with an equal sign. Blank lines are allowed around justification lines.

```
<expr1>
= <justification1>
...
= <justificationN>
<exprN+1>
```

The justification format is as follows:

```
jrel law [usage] [focus]
```

```
1. jrel (mandatory):
 =  -- logical equality
    -- inequalities such as <, <=, ==>, .. may be supported later
2. law (mandatory):
  LAW name     -- name of law
  DEF name     -- defn of name
  DEF name.i   -- defn of name, identifying clause
  INDHYP       -- inductive hypothesis
  CASE         -- case assumption
  SIMP         -- Simplifier
3. usage (optional):
      -- if omitted for DEF, then l2r
      -- if omitted otherwise, then whole law
  l2r -- left-to-right (for laws of form lhs = rhs)
  r2l -- right-to-left (for laws of form lhs = rhs)
4. focus (optional):
            -- if omitted for DEF, focus is first occurrence of DEF name
            -- if omitted for anything else, the focus is at top-level
  @ name    -- the first occurrence of that name
  @ name i  -- ith in-order occurrence of name
```

2.3.2 Datatypes

```
THEORY <TheoryName>
IMPORT-THEORY <Name>
IMPORT-HASKELL <Name> ...
```

```
data Theory
 = THEORY {
     theoryName  :: String
   , thImports   :: [String]  -- Theory Names
   , hkImports   :: [String]  -- Haskell Module names
   , thLaws      :: [Law]
   , thIndScheme :: [InductionScheme]
   , thTheorems  :: [Theorem]
   }
 deriving Show

thImports__   f thry = thry{ thImports   = f $ thImports thry }
hkImports__   f thry = thry{ hkImports   = f $ hkImports thry }
thLaws__      f thry = thry{ thLaws      = f $ thLaws    thry }
thIndScheme__ f thry = thry{ thIndScheme = f $ thIndScheme thry }
thTheorems__  f thry = thry{ thTheorems  = f $ thTheorems thry }
```

```
LAW <name> <br?> <expr>
```

```
data Law
 = LAW {
     lawName :: String
   , lawEqn :: Expr
   }
 deriving Show
```

```
INDUCTION-SCHEME <Type>
BASE <value>
STEP <var> --> <expr>
INJ <br?> <expr> == <expr>
```

```
data InductionScheme
 = IND {
     indType :: String
   , indVar  :: String  -- generic induction variable
   , indBase :: Expr              -- base value
   , indStep :: Expr    -- induction var to step expression
   }
 deriving Show
```

```
THEOREM <name> <br?> <expr>
STRATEGY <strategy>
<strategy-body>
QED <name>
```

```
data Theorem
 = THEOREM {
     thmName :: String
   , theorem :: Expr
   , strategy :: Strategy
   }
 deriving Show
```

```
ReduceAll
ReduceLHS
ReduceRHS
ReduceBoth
```

```
data Strategy
 = ReduceAll Calculation
 | ReduceLHS Calculation
 | ReduceRHS Calculation
 | ReduceBoth Calculation Calculation
```

```
STRATEGY Induction <ind-var> ::  <type>
BASE <val> <br!> <expr>
<one of the other four strategies>
QED BASE
STEP <expr>
ASSUME <br?> <expr>
SHOW <br?> <expr>
<one of the other four strategies>
QED STEP
```

```
 | Induction { -- goal is what we are proving by induction
     iVar :: (String,String)   -- var :: type
   , baseVal :: Expr           -- base value
   , bGoal :: Expr             --  goal[baseVal/var]
   , baseStrategy :: Strategy
   , stepExpr :: Expr          -- expr
   , assume :: Expr            -- goal
   , iGoal :: Expr             -- goal[stepExpr/var]
   , stepStrategy :: Strategy
   }
 deriving Show
```

```
<expr1>
= <justification1>
...
= <justificationN>
<exprN+1>
```

```
data Calculation
 = CALC {
     goal :: Expr
   , calcs :: [(Justification,Expr)]
   }
 deriving Show
```

Justifications:

```
jrel law [usage] [focus]

1. jrel (mandatory):
 =  -- logical equality
    -- inequalities such as <, <=, ==>, .. may be supported later
2. law (mandatory):
  LAW name     -- name of law
  DEF name     -- defn of name
  DEF name.i   -- defn of name, identifying clause
  INDHYP       -- inductive hypothesis
  CASE         -- case assumption
  SIMP         -- Simplifier
3. usage (optional):
      -- if omitted for DEF, then l2r
      -- if omitted otherwise, then whole law
  l2r -- left-to-right (for laws of form lhs = rhs)
  r2l -- right-to-left (for laws of form lhs = rhs)
4. focus (optional):
            -- if omitted for DEF, focus is first occurrence of DEF name
            -- if omitted for anything else, the focus is at top-level
  @ name    -- the first occurrence of that name
  @ name i  -- ith in-order occurrence of name
```

```haskell
data Justification
 = BECAUSE {
     jrel :: JRel
   , law :: JLaw
   , usage :: Usage
   , focus :: Focus
   }
 deriving Show
data JRel = JEq deriving (Eq, Show)
data JLaw = L String | D String Int | IH | CS | SMP deriving (Eq, Show)
data Usage = Whole | L2R | R2L deriving (Eq, Show)
data Focus = Top | At String Int deriving (Eq, Show)
```

2.3.3 Parser Top-Level

We start by adding in an “empty” theory as an accumulating parameter, breaking input into numbered lines and starting the proper parsing.

```
parseTheory :: Monad m => ParseMode -> String -> m Theory
parseTheory pmode str
  = do (thry,_) <- theoryParser pmode theory0 $ zip [1..] $ lines str
       return thry

theory0 = THEORY { theoryName = "?", thImports = [], hkImports = []
                 , thLaws = [], thIndScheme = [], thTheorems = [] }
```

We start proper parsing by looking for THEORY <TheoryName> on the first line:

```
theoryParser :: Monad m => ParseMode -> Theory -> Parser m Theory
theoryParser pmode theory lns
 = do (thryNm,lns') <- requireKeyAndName "THEORY" lns
      parseBody pmode theory{theoryName = thryNm} lns'
```

```
parseBody :: Monad m => ParseMode -> Theory -> Parser m Theory
parseBody pmode theory [] = return (theory, [])
parseBody pmode theory (ln@(lno,str):lns)
 -- we skip empty lines here...
 | emptyLine str  =  parseBody pmode theory lns

 -- simple one-liners
 | gotImpTheory   =  parseBody pmode (thImports__ (++[thryName]) theory) lns
 | gotImpCode     =  parseBody pmode (hkImports__ (++[codeName]) theory) lns

 -- complex parsers
 | gotIndSchema = callParser (parseIndSchema pmode theory typeName lno)     lns
 | gotLaw       = callParser (parseLaw pmode theory lwName lno lrest)       lns
 | gotTheorem   = callParser (parseTheorem pmode theory thrmName lno trest) lns

 | otherwise      =  pFail pmode lno 1 $ unlines
                       [ "unexpected line:\n"++str
                       , "expecting IMPORT-X, LAW, INDUCTION-SCHEME, THEOREM" ]
 where
   (gotImpTheory, thryName)       =  parseKeyAndName "IMPORT-THEORY"    str
   (gotImpCode,   codeName)       =  parseKeyAndName "IMPORT-HASKELL"   str
   (gotLaw, lwName, lrest)        =  parseOneLinerStart "LAW"           str
   (gotIndSchema, typeName)       =  parseKeyAndName "INDUCTION-SCHEME" str
   (gotTheorem, thrmName, trest)  =  parseOneLinerStart "THEOREM"       str

   callParser parser lns
     = do (theory',lns') <- parser lns
          parseBody pmode theory' lns'
```

2.3.4 Parse Laws

LAW <name> <br?> <expr>

```
parseLaw :: Monad m => ParseMode -> Theory  -> String -> Int -> String
         -> Parser m Theory
parseLaw pmode theory lwName lno rest lns
  = case parseExprChunk pmode lno rest lns of
      But msgs
        -> pFail pmode lno 1 $ unlines msgs
      Yes (expr, lns')
```

```
        ->  return (thLaws__ (++[LAW lwName expr]) theory, lns')

parseExprChunk :: Monad m => ParseMode -> Int -> String -> Parser m Expr
parseExprChunk pmode lno rest lns
 | emptyLine rest  =  parseExpr pmode restlns chunk
 | otherwise       =  parseExpr pmode lns     [(lno,rest)]
 where (chunk,restlns) = getChunk lns
```

2.3.5 Parse Induction Schemata

```
INDUCTION-SCHEME <Type>
BASE <value>
STEP <var> --> <expr>
INJ <br?> <expr> == <expr>
```

```
parseIndSchema :: Monad m => ParseMode -> Theory -> String -> Int
               -> Parser m Theory
parseIndSchema pmode theory typeName lno (ln1:ln2:ln3:lns)
 | not gotBase  =  pFail pmode (lno+1) 1 "INDUCTION-SCHEME: missing BASE"
 | not gotStep  =  pFail pmode (lno+2) 1 "INDUCTION-SCHEME: missing STEP"
 | not gotInj   =  pFail pmode (lno+3) 1 "INDUCTION-SCHEME: missing INJ"
 | otherwise
    =  case parseEquivChunk pmode (lno+3) ln3rest lns of
        Nothing
          ->  pFail pmode lno 1 "INDUCTION-SCHEME: Injective law expected"
        Just ((e1,e2), lns')
          ->  parseBody pmode (thIndScheme__ (++[ind]) theory) lns'
 where
   (gotBase,bValue) = parseKeyAndValue pmode "BASE" $ snd ln1
   (gotStep,sVar,eStep) = parseKeyNameKeyValue pmode "STEP" "-->" $ snd ln2
   len = length "INJ"
   (ln3inj,ln3rest) = splitAt len $ snd ln3
   gotInj = ln3inj == "INJ"
   ind = IND typeName sVar bValue eStep
parseIndSchema pmode theory typeName lno _
  = pFail pmode lno 0 "INDUCTION-SCHEME: Incomplete"
```

Look for two expressions connected by ‘equality’.‘

```
parseEquivChunk :: Monad m => ParseMode -> Int -> String
                -> Parser m (Expr,Expr)
parseEquivChunk pmode lno rest lns
 | emptyLine rest  =  parseEqual pmode restlns chunk
 | otherwise       =  parseEqual pmode lns      [(lno,rest)]
 where (chunk,restlns) = getChunk lns
```

2.3.6 Parse Theorems

```
THEOREM <name> <br?> <expr>
STRATEGY <strategy>
<strategy-body>
QED <name>
```

```
parseTheorem :: Monad m => ParseMode -> Theory -> String -> Int -> String
             -> Parser m Theory
parseTheorem pmode theory thrmName lno rest lns
  = case parseExprChunk pmode lno rest lns of
      Nothing
        ->  pFail pmode lno 0 "Theorem expression expected"
      Just (goal, lns')
        -> do (strat,lns'') <- parseStrategy pmode lns'
              let thry = THEOREM thrmName goal strat
              let theory' = thTheorems__ (++[thry]) theory
              return (theory',lns'')
```

Parse Strategies

```
parseStrategy :: Monad m => ParseMode -> Parser m Strategy
parseStrategy pmode [] = pFail pmode maxBound 0 "STRATEGY: premature end of file"
parseStrategy pmode (ln:lns)
  | gotReduce     =  parseReduction pmode rstrat lns
  | gotInduction  =  parseInduction pmode vartyp lns
  | otherwise     =  pFail pmode (fst ln) 0 $ unlines
                       [ "Found: " ++ snd ln
                       , "when STRATEGY <strategy> expected." ]
  where
    (gotReduce,rstrat) = parseRedStratDecl $ snd ln
    (gotInduction,vartyp) = parseIndStratDecl $ snd ln
```

```
ReduceAll
ReduceLHS
ReduceRHS
ReduceBoth
```

```
parseRedStratDecl str
  | stratSpec == ["STRATEGY","ReduceAll"]   =  (True,ReduceAll  udefc)
  | stratSpec == ["STRATEGY","ReduceLHS"]   =  (True,ReduceLHS  udefc)
  | stratSpec == ["STRATEGY","ReduceRHS"]   =  (True,ReduceRHS  udefc)
  | stratSpec == ["STRATEGY","ReduceBoth"]  =  (True,ReduceBoth udefc udefc)
  | otherwise  =  (False,error "not a reduction strategy")
  where
    stratSpec = words str
    udefc = error "undefined reduce calculation"
```

```
parseReduction :: Monad m => ParseMode -> Strategy
               -> Parser m Strategy
```

```
ReduceAll| ReduceLHS| ReduceRHS
<Calculation>
```

```
-- single reductions end with "QED"
parseReduction pm (ReduceAll _) lns  =  parseReduction' pm "QED" ReduceAll lns
parseReduction pm (ReduceLHS _) lns  =  parseReduction' pm "QED" ReduceLHS lns
parseReduction pm (ReduceRHS _) lns  =  parseReduction' pm "QED" ReduceRHS lns
```

```
ReduceBoth
LHS
<calculation>
RHS
<calculation>
```

```
parseReduction pm (ReduceBoth _ _) lns
 = do (_,lns1) <- requireKey "LHS" lns
      -- first reduction ends with "RHS"
      (ReduceAll red1,lns2) <- parseReduction' pm "RHS" ReduceAll lns1
      -- second reduction ends with "QED"
      (ReduceAll red2,lns3) <- parseReduction' pm "QED" ReduceAll lns2
      return (ReduceBoth red1 red2,lns3)
```

```
parseReduction' pmode calcStop reduce lns
 = do (calc, lns') <- parseCalculation pmode calcStop lns
      -- expect calcStop
      completeCalc pmode calcStop reduce calc lns'

completeCalc pmode calcStop _ _ [] = pFail pmode 0 0 $ unlines
                                       [ "Premature end of file"
                                       , "Expecting: "++calcStop ]
completeCalc pmode calcStop reduce calc ((num,str):lns)
 | take 1 (words str) == [calcStop]  =  return (reduce calc,lns)
 | otherwise  =  pFail pmode num 0 $ unlines
                    [ "Improper calc end: "++str
                    , "Expecting: "++calcStop ]
```

```
STRATEGY Induction <ind-var> ::  <type>
```

```
parseIndStratDecl str
  = case words str of
      ("STRATEGY":"induction":indtvars)  ->  parseIndVars indtvars
      _ -> (False,error "not an induction strategy")

parseIndVars [] = (False,error "no induction variables defined.")
parseIndVars [var,"::",typ] = (True, (var,typ))
parseIndVars _ = (False, error "Expected var :: type")
```

```
BASE <val> <br!> <expr>
<one of the other four strategies>
QED BASE
STEP <expr>
ASSUME <br?> <expr>
SHOW <br?> <expr>
<one of the other four strategies>
QED STEP
```

```
parseInduction :: Monad m => ParseMode -> (String,String) -> Parser m Strategy
parseInduction pmode _ []
  = pFail pmode 0 0 "Induction proof: premature end-of-file"
parseInduction pmode vartyp lns
  = do (bval,lns1) <- requireKeyAndValue pmode "BASE" lns
       (bexpr,lns2) <- parseExprChunk pmode 0 [] lns1
       (bstrat,lns3) <- parseStrategy pmode lns2
       (sexpr,lns4) <- requireKeyAndValue pmode "STEP" lns3
       (_,lns5a) <- requireKey "ASSUME" lns4
       (ass,lns5) <- parseExprChunk pmode 0 [] lns5a -- FIX
       (_,lns6a) <- requireKey "SHOW" lns5
       (goal,lns6) <- parseExprChunk pmode 0 [] lns6a -- FIX
       (sstrat,lns7) <- parseStrategy pmode lns6
       (thnm,lns8) <- requireKeyAndName "QED" lns7
       return ( Induction { iVar = vartyp
                          , baseVal = bval
                          , bGoal = bexpr
                          , baseStrategy = bstrat
                          , stepExpr = sexpr
                          , assume = ass
                          , iGoal = goal
                          , stepStrategy = sstrat
                          }
              , lns8
              )
```

```
<expr1>
= <justification1>
...
= <justificationN>
<exprN+1>
```

```
type Steps = [(Line,Lines)]
```

This requires multiple “chunks” to be parsed. Blank lines are separators, as are lines beginning with a leading space followed by a single equal sign. A calculation is ended by a line starting with `calcStop`.

```
parseCalculation :: Monad m => ParseMode -> String -> Parser m Calculation
parseCalculation pmode calcStop lns
  = do (calcChunks,rest) <- takeLinesBefore calcStop lns
       ((fstChunk,sepChunks),_) <- splitLinesOn pmode isJustificationLn calcChunks
       (goalPred,_) <- parseExpr pmode [] fstChunk
       steps <- parseSteps pmode sepChunks
       return (CALC goalPred steps, rest)
```

Break line-list at the first use of a designated keyword, discarding empty lines along the way

```
takeLinesBefore :: Monad m => String -> Parser m Lines
takeLinesBefore _ [] = return ( [], [] )
takeLinesBefore key lns@(ln:lns')
 | null lnwords         =  takeLinesBefore key lns'
```

```
 | head lnwords == key   =  return ( [], lns )
 | otherwise             =  do (before,after) <- takeLinesBefore key lns'
                               return ( ln:before, after )
 where lnwords = words $ snd ln
```

A justification line has a first word that is an equals-sign (for now).

```
isJustificationLn :: Line -> Bool
isJustificationLn (_,str)  =  case words str of
                                []      ->  False
                                (w:_)   ->  w `elem` ["="]
```

Split into maximal chunks seperated by lines that satisfy `splitHere`:

```
splitLinesOn :: Monad m => ParseMode -> (Line -> Bool) -> Parser m (Lines,Steps)

-- we expect at least one line before split
splitLinesOn pmode splitHere [] = pFail pmode 0 0 "premature end of calc."
splitLinesOn pmode splitHere (ln:lns)
 | splitHere ln  = pFail pmode (fst ln) 0 $ unlines
                     [ "Cannot start with: " ++ snd ln
                     , "Expecting expression" ]
 | otherwise    =  splitLinesOn' pmode splitHere [ln] lns

-- seen initial chunk, looking for first split
splitLinesOn' pmode splitHere knuhc []  =  return ((reverse knuhc,[]),[])
splitLinesOn' pmode splitHere knuhc (ln:lns)
 | splitHere ln  =  splitLinesOn'' pmode splitHere (reverse knuhc) [] ln [] lns
 | otherwise    = splitLinesOn' pmode splitHere (ln:knuhc) lns

-- found split
-- accumulating post-split chunk
splitLinesOn'' pmode splitHere chunk0 spets split knuhc []
 | null knuhc   =  pFail pmode (fst split) 0 "premature end of calc."
 | otherwise    =  return ( ( chunk0
                          , reverse ((split, reverse knuhc):spets) )
                        , [] )
splitLinesOn'' pmode splitHere chunk0 spets split knuhc (ln:lns)
 | splitHere ln  =  splitLinesOn'' pmode splitHere
                                  chunk0 ((split, reverse knuhc):spets) ln [] lns
 | otherwise    = splitLinesOn'' pmode splitHere chunk0 spets split (ln:knuhc) lns
```

Parsing calculation steps:

```
parseSteps :: Monad m => ParseMode -> Steps -> m [(Justification,Expr)]
parseSteps pmode [] = return []
parseSteps pmode ((justify,chunk):rest)
  = do just <- parseJustification pmode justify
       (exp,_) <- parseExpr pmode [] chunk
       steps <- parseSteps pmode rest
       return ((just,exp):steps)
```

Parsing a justification.

```
jrel law [usage] [focus]

1. jrel (mandatory):
 =  -- logical equality
    -- inequalities such as <, <=, ==>, .. may be supported later
2. law (mandatory):
  LAW name     -- name of law
  DEF name     -- defn of name
  DEF name.i   -- defn of name, identifying clause
  INDHYP       -- inductive hypothesis
  CASE         -- case assumption
  SIMP         -- Simplifier
3. usage (optional):
      -- if omitted for DEF, then l2r
      -- if omitted otherwise, then whole law
  l2r -- left-to-right (for laws of form lhs = rhs)
  r2l -- right-to-left (for laws of form lhs = rhs)
4. focus (optional):
            -- if omitted for DEF, focus is first occurrence of DEF name
            -- if omitted for anything else, the focus is at top-level
  @ name    -- the first occurrence of that name
  @ name i  -- ith in-order occurrence of name
```

Parsing of whole line — need at least two words

```haskell
parseJustification :: Monad m => ParseMode -> Line -> m Justification
parseJustification pmode (lno,str)
 = case words str of
    (w1:w2:wrest) ->  do jr <- parseJRel w1
                         parseJustify pmode lno jr wrest w2
    _ ->  pFail pmode lno 0 "incomplete justification"
 where
    parseJRel "="  =  return JEq
    parseJRel  x   =  pFail pmode lno 1 ("unrecognised proof relation: "++x)
```

Parsing given at least two words, the first of which is OK. If we get a succesful parse, we ignore anything leftover.

```haskell
parseJustify :: Monad m => ParseMode -> Int -> JRel -> [String] -> String
             -> m Justification
parseJustify pmode lno jr wrest w2
 | w2 == "LAW"    = parseLawName pmode lno jr     wrest
 | w2 == "DEF"    = parseDef     pmode lno jr     wrest
 | w2 == "INDHYP" = parseUsage   pmode lno jr IH  wrest
 | w2 == "CASE"   = parseUsage   pmode lno jr CS  wrest
 | w2 == "SIMP"   = parseUsage   pmode lno jr SMP wrest
 | otherwise      = pFail        pmode lno  1
                          ("unrecognised law specification: "++w2)
```

Seen a `LAW`, expecting a `fname`

```haskell
parseLawName pmode lno jr []          =  pFail pmode lno 0 "LAW missing name"
parseLawName pmode lno jr (w:wrest)   =  parseUsage pmode lno jr (L w) wrest
```

Seen a `DEF`, expecting a `fname[.i]`

```
parseDef pmode lno jr [] = pFail pmode lno 0 "DEF missing name"
parseDef pmode lno jr (w:wrest) =  parseUsage pmode lno jr (mkD w) wrest

mkD w -- any error in '.loc' results in value 0
  | null dotloc        =  D w 0
  | null loc           =  D nm 0
  | all isDigit loc    =  D nm $ read loc
  | otherwise          =  D nm 0
  where
    (nm,dotloc) = break (=='.') w
    loc = tail dotloc
```

Seen law, looking for optional usage.

```
parseUsage pmode lno jr jlaw []
                       =  return $ BECAUSE jr jlaw (defUsage jlaw) (defFocus jlaw)
parseUsage pmode lno jr jlaw ws@(w:wrest)
  | w == "l2r"   =  parseFocus pmode lno jr jlaw L2R               wrest
  | w == "r2l"   =  parseFocus pmode lno jr jlaw R2L               wrest
  | otherwise    =  parseFocus pmode lno jr jlaw (defUsage jlaw) ws

defUsage (D _ _)  =  L2R
defUsage _        =  Whole
```

Seen law and possible usage, looking for optional focus. Expecting either `@ name` or `@ name i`

```
parseFocus pmode lno jr jlaw u []
                                 =  return $ BECAUSE jr jlaw u $ defFocus jlaw
parseFocus pmode lno jr jlaw u [w1,w2]
  | w1 == "@"                    =  return $ BECAUSE jr jlaw u $ At w2 1
parseFocus pmode lno jr jlaw u [w1,w2,w3]
  | w1 == "@" && all isDigit w3  =  return $ BECAUSE jr jlaw u $ At w2 $ read w3
parseFocus pmode lno jr jlaw u ws
     =  pFail pmode lno 0 ("invalid focus: "++unwords ws)

defFocus (D n _)  =  At n 1
defFocus _        =  Top
```

2.3.7 "One-Liner" Parsing

Speculative line-parses

The following line parsers check to see if a line has a particular form, returning a true boolean value that is so, plus extra information if required.

```
emptyLine :: String -> Bool
emptyLine str = all isSpace str || take 2 (dropWhile isSpace str) == "--"
```

We return a boolean that is true if the parse suceeds.

```
parseKeyAndName :: String -> String -> (Bool, String)
parseKeyAndName key str
  = case words str of
      [w1,w2] | w1 == key  ->  (True,  w2)
      _                    ->  (False, error ("Expecting '"++key++"' and name"))
```

```
parseKeyAndValue :: ParseMode -> String -> String -> (Bool, Expr)
parseKeyAndValue pmode key str
  = case words str of
      (w1:wrest) | w1 == key
        -> case parseExpr pmode [] [(0,unwords wrest)] of
            Nothing -> (False, error ("Bad value: "++ unwords wrest))
            Just (hsexp,_) ->  (True,  hsexp)
      _                    ->  (False, error ("Expecting '"++key++"' and value"))
```

```
parseKeyNameKeyValue :: ParseMode -> String -> String -> String
                     -> (Bool,String,Expr)
parseKeyNameKeyValue pmode key1 key2 str
  = case words str of
      (w1:w2:w3:wrest) | w1 == key1 && w3 == key2
        -> case parseExpr pmode [] [(0,unwords wrest)] of
            Nothing -> (False, "", error ("Bad value: "++ unwords wrest))
            Just (hsexp,_) ->  (True,  w2, hsexp)
      _                    ->  (False, "", error ("Expecting '"++key2++"' and value"))
```

```
parseOneLinerStart :: String -> String -> (Bool,String,String)
parseOneLinerStart key str
  = case words str of
      (w1:w2:rest) | w1 == key  ->  (True,  w2, unwords rest)
      _                         ->  ( False
                                    , error "parseOneLinerStart failed!"
                                    , str)
```

Mandatory one-liners

These parsers expect a specific form of line as the first non-empty line in the current list of lines, and fail with an error if not found.

```
requireKey :: Monad m => String -> Parser m ()
requireKey key [] = fail ("EOF while expecting "++key)
requireKey key (ln@(lno,str):lns)
 | emptyLine str  = requireKey key lns
 | otherwise
   = case words str of
       [w1] | w1 == key  ->  return ((),lns)
       _  ->  lFail lno ("Expecting '"++key++"', found:\n"++str)
```

Here, we expect something on the current line.

```
requireKeyAndName :: Monad m => String -> Parser m String
requireKeyAndName key [] = fail ("EOF while expecting "++key++" <name>")
requireKeyAndName key (ln@(lno,str):lns)
  | emptyLine str  =  requireKeyAndName key lns
  | otherwise
   = case words str of
       [w1,w2] | w1 == key  ->  return (w2,lns)
       _                    ->  lFail lno ("Expecting '"++key++"' and name")
```

Here we will pass over empty lines.

```
requireKeyAndValue :: Monad m => ParseMode -> String -> Parser m Expr
requireKeyAndValue pmode key [] = fail ("EOF while expecting "++key++" <expr>")
requireKeyAndValue pmode key (ln@(lno,str):lns)
  | emptyLine str  =  requireKeyAndValue pmode key lns
  | otherwise
   = case words str of
       (w1:wrest) | w1 == key
          ->  parseExpr pmode lns [(0,unwords wrest)]
       _  ->  fail ("Expecting '"++key++"' and expr")
```

```
lFail lno msg = fail ("Line:"++show lno++"\n"++msg)
```

2.3.8 Chunk Parser

A chunk is found by skipping over zero or more empty lines, to find a maximal run of one or more non-empty lines. A chunck is either followed by at least one empty line, or the end of all of the lines.

```
getChunk []        =  ([],[])

getChunk (ln@(_,str):lns)
 | emptyLine str   =  getChunk        lns
 | otherwise       =  getChunk' [ln] lns

getChunk' snl []   =  (reverse snl, [])

getChunk' snl (ln@(_,str):lns)
 | emptyLine str   =  (reverse snl,lns)
 | otherwise       =  getChunk' (ln:snl) lns
```

2.3.9 Theorem Utilities

```
findTheorem :: Monad m => String -> [Theorem] -> m Theorem
findTheorem _ [] = fail "theorem not found"
findTheorem nm (thm:thms)
 | nm == thmName thm   =  return thm
 | otherwise           =  findTheorem nm thms
```

2.4 Matching

```
Copyright  Andrew Butterfield (c) 2017--18

LICENSE: BSD3, see file LICENSE at reasonEq root
```

```
module Matching
( Binding
, eMatch
, buildReplacement
)
where

import Data.Map (Map)
import qualified Data.Map as M
import Data.Set (Set)
import qualified Data.Set as S

import AST

import Debug.Trace
dbg msg x = trace (msg ++ show x) x
mdbg msg x = return $! dbg msg x
```

2.4.1 Bindings

We bind (variable) names to expressions:

```
type Binding = Map String Expr
```

Our standard lookup is total, taking a name and searching for it and its associated expression. If not found, a vairable with that name is returned.

```
bGet :: Binding -> String -> Expr
bGet bind nm
  = case M.lookup nm bind of
      Nothing  ->  Var nm
      Just e   ->  e
```

Update is partial, as redefinition of an name already present is not allowed:

```
bSet :: Monad m => String -> Expr -> Binding -> m Binding
bSet nm e bind
  = case M.lookup nm bind of
      Nothing              ->  return $ M.insert nm e bind
      Just e0 | e0 == e    ->  return bind
      _                    ->  fail "conflicting defs."
```

2.4.2 Known Names

Some names only match themselves. Some we hardwire, as (re-)defining them in imported Haskell files is impossible/awkward:

```
hardPrelude :: Set String
hardPrelude = S.fromList
  [ "even","otherwise","ord","minimum","maximum"
  , "+","-","*","/","div","mod"
  , ">",">=","<","<="
  , "==","&&","||"
```

```
  , "++","[]",":"
  ]
```

We want to extend these with

2.4.3 Matching

Matching takes candidate and pattern expressions, along with a list of names defined in the context where the pattern is given, and establishes if the candidate is an instance of the pattern. If so, it returns a binding that maps the pattern to the candidate.

```
eMatch :: Monad m => [String] -> Expr -> Expr -> m Binding
eMatch knwnNms cand patn
 = mExpr (S.fromList knwnNms `S.union` hardPrelude) M.empty cand patn
```

Here we have an input binding, initially empty, that grows as matching proceeds.

```
mExpr  :: Monad m => Set String -> Binding -> Expr -> Expr -> m Binding
```

Literals only match themselves

```
mExpr known bind (LBool c) (LBool  p)
  =  if  c == p  then  return bind  else  fail "diff. bool"
mExpr known bind (LInt  c) (LInt   p)
  =  if  c == p  then  return bind  else  fail "diff. int"
mExpr known bind (LChar c) (LChar  p)
  =  if  c == p  then  return bind  else  fail "diff. char"
```

Variables match themselves, of course, but will also match anything if not “known”:

```
mExpr known bind (Var c) (Var p)
  | c == p                 =  return bind
mExpr known bind cand (Var p)
  | p == "_"               =  return bind
  | p `S.member` known     =  fail "not self."
  | otherwise              =  bSet p cand bind
```

Applications match if both function and argument expressions do:

```
mExpr known bind0 (App c1 c2) (App p1 p2)
  = do { bind1 <- mExpr known bind0 c1 p1
       ;          mExpr known bind1 c2 p2 }
```

If-then-else match if condition-, then- and else-expressions do.:

```
mExpr known bind0 (If c1 c2 c3) (If p1 p2 p3)
  = do { bind1 <- mExpr known bind0 c1 p1
       ; bind2 <- mExpr known bind1 c2 p2
       ;          mExpr known bind2 c3 p3 }
```

```
mExpr known bind cand patn = fail "no match found"
```

2.4.4 Building

For now, ignore local declarations

```
buildReplacement :: Binding -> [Decl] -> Expr -> Expr
buildReplacement bind _ b@(LBool _) = b
buildReplacement bind _ i@(LInt  _) = i
buildReplacement bind _ c@(LChar _) = c
buildReplacement bind _   (Var n)   = bGet bind n
```

```
buildReplacement bind ldcls (App e1 e2)
  = App (buildReplacement bind ldcls e1)
        (buildReplacement bind ldcls e2)
```

2.4.5 Stuff

```
data Expr
  = LBool Bool | LInt Int | LChar Char
  | Var String
  | App Expr Expr
  | If Expr Expr Expr
  | GrdExpr [(Expr,Expr)]
  | Let [Decl] Expr
  | PApp String [Expr]
  deriving (Eq,Show)

data Match = Match { fname ::  String  -- function name
                   , lhspat :: [Expr]  -- LHS patterns
                   , rhs :: Expr    -- RHS outcome
                   , ldecls :: [Decl]  -- local declarations
                   }
           deriving (Eq, Show)

data Decl
  = Fun [Match]
  | Bind Expr Expr [Decl]
  | Syntax -- not relevant to this tool !
  | Type String -- just noting name for now - to be addressed later
  deriving (Eq, Show)
```

2.5 Checking

```
Copyright  Andrew Butterfield (c) 2017--18

LICENSE: BSD3, see file LICENSE at reasonEq root
```

```
module Check
(Report, showReport, checkTheorem)
where

import AST
import Theory
import Matching

import Debug.Trace
dbg msg x = trace (msg ++ show x) x
mdbg msg x = return $! dbg msg x
```

```
type Report = [String]

showReport rep = putStrLn $ unlines rep

rep :: String -> Report
rep str = lines str

rjoin :: Report -> Report -> Report
r1 `rjoin` r2 = r1 ++ r2
```

```
checkTheorem :: [Mdl] -> [Theory] -> Theorem -> Report
checkTheorem mdls thrys thm
  = rep ("\nChecking theorem '"++thmName thm++"'")
      `rjoin` (checkStrategy mdls thrys dummyH (theorem thm) $ strategy thm)
```

Induction and case-based strategies require a hypothesis to be passed to the calculation checker, while the variations of reduction don't. In the latter case we pass in a dummy hypothesis:

```
dummyH = Var "??"
```

```
checkStrategy :: [Mdl] -> [Theory] -> Expr -> Expr -> Strategy -> Report
```

```
checkStrategy mdls thrys hyp goal (ReduceAll calc)
  = rep "\nStrategy: reduce all to True"
      `rjoin` checkFirst calc goal
      `rjoin` checkCalc mdls thrys hyp calc
      `rjoin` checkLast calc (LBool True)

checkStrategy mdls thrys hyp goal (ReduceLHS calc)
  = rep "\nStrategy: reduce LHS to RHS"
      `rjoin` checkFirst calc (lhsOf goal)
      `rjoin` checkCalc mdls thrys hyp calc
      `rjoin` checkLast calc (rhsOf goal)

checkStrategy mdls thrys hyp goal (ReduceRHS calc)
  = rep "\nStrategy: reduce RHS to LHS"
      `rjoin` checkFirst calc (rhsOf goal)
      `rjoin` checkCalc mdls thrys hyp calc
      `rjoin` checkLast calc (lhsOf goal)

checkStrategy mdls thrys hyp goal (ReduceBoth cLHS cRHS)
  = rep "\n Strategy: reduce RHS and LHS to same"
```

```
     `rjoin` checkBothStart goal cLHS cRHS
     `rjoin` rep "\nCheck LHS" `rjoin` checkCalc mdls thrys hyp cLHS
     `rjoin` rep "\nCheck RHS" `rjoin` checkCalc mdls thrys hyp cRHS
     `rjoin` checkSameLast cLHS cRHS

-- istrat must be (Induction ...)
checkStrategy mdls thrys hyp goal istrat
  = rep ( "\nStrategy: Induction in " ++ var ++ " :: "++ typ )
     `rjoin` checkIndScheme thrys goal bgoal igoal var typ
     `rjoin` rep "\nCheck Base Case..."
     `rjoin` checkStrategy mdls thrys hyp bgoal (baseStrategy istrat)
     `rjoin` rep "\nCheck Step Case..."
     `rjoin` checkStrategy mdls thrys (assume istrat) igoal (stepStrategy istrat)
     `rjoin` rep "\nInduction NYFI"
  where
    (var,typ) = iVar istrat
    bgoal = bGoal istrat
    igoal = iGoal istrat
```

```
checkFirst :: Calculation -> Expr -> Report
checkFirst (CALC e0 _) e
  | e0 == e    =  rep "OK: correct first expression."
  | otherwise =  rep "!!: incorrect first expression."
```

```
lastE :: Calculation -> Expr
lastE (CALC e [])      =  e
lastE (CALC _ steps)  =  snd $ last steps

checkLast :: Calculation -> Expr -> Report
checkLast calc e
  | (lastE calc) == e  =  rep "OK: correct last expression."
checkLast _ _          =  rep "!!: incorrect last expression."
```

```
checkBothStart :: Expr -> Calculation -> Calculation -> Report
checkBothStart goal (CALC gLHS _) (CALC gRHS _)
  | goal == equal gLHS gRHS  = rep "OK: goal lhs/rhs"
  | otherwise                = rep "!!: (lhs = rhs) is not goal"
```

```
checkSameLast :: Calculation -> Calculation -> Report
checkSameLast cLHS cRHS
 | lastE cLHS == lastE cRHS  =  rep "OK: last expressions are the same."
 | otherwise                 =  rep "!!: last expressions differ."
```

```
equal :: Expr -> Expr -> Expr
equal e1 e2 = App (App eEq e1) e2
```

```
lhsOf (App (App eq e1) _)
 | eq == eEq   =  e1
lhsOf e        =  e

rhsOf (App (App eq _) e2)
 | eq == eEq   =  e2
rhsOf e        =  e
```

```
checkIndScheme thrys goal bgoal igoal var typ
  = case findTheoryInds thrys typ of
      Nothing -> rep ("No Induction scheme for "++typ)
      Just inds
       ->  rep ("Ind Scheme '"++typ++"' valid")
```

```
        `rjoin`
        rep "checkIndScheme n.y.f.i."
    -- we need to check base = indscheme.base
```

We keep the best until last ...

```
checkCalc :: [Mdl] -> [Theory] -> Expr -> Calculation -> Report
checkCalc mdls thrys hyp (CALC goal [])  =  rep "!!: no steps to check"
checkCalc mdls thrys hyp (CALC goal steps)
  = checkSteps mdls thrys hyp goal steps
checkSteps _ _ _ _ []  = rep "check complete"
checkSteps mdls thrys hyp goal ((just,goal'):steps)
  = checkStep mdls thrys hyp goal just goal'
     `rjoin` checkSteps mdls thrys hyp goal' steps
```

This is where all the heavy lifting is done:

```
checkStep :: [Mdl] -> [Theory] -> Expr -> Expr -> Justification -> Expr
          -> Report

checkStep mdls thrys hyp goal (BECAUSE _ (D dnm i) howused what) goal'
-- need to modify this based on howused !!!!
 = case searchMods mdls dnm i of
   Nothing -> rep ("!!: Can't find definition "++dnm++"."++show i)
   Just defn
    -> case findAndApplyDEFN (mdlsKnown mdls) defn goal howused what of
       Nothing -> rep ("!!: Failed to apply "++show dnm++"."++show i)
       Just goal''
        -> if goal'' == goal'
           then rep ("OK: use of "++dnm++"."++show i++" is correct.")
           else rep $ unlines
                  [ "!!: use of "++dnm++"."++show i++" differs."
                  , "Expected:\n"++show goal'
                  , "Got:\n"++show goal''
                  ]

checkStep mdls thrys hyp goal (BECAUSE _ (L lnm) howused what) goal'
 = case findTheoryLaws thrys lnm of
     Nothing -> rep ("!!: Can't find law "++lnm)
     Just thelaw
       -> case findAndApplyLAW (mdlsKnown mdls) thelaw goal howused what of
            Nothing -> rep ("!!: Failed to apply "++lnm++" "++show howused)
            Just goal''
              -> if goal'' == goal'
                  then rep ("OK: use of "++lnm++" "++show howused++" is correct.")
                  else rep ("!!: use of "++lnm++" "++show howused++" differs.")

checkStep mdls thrys hyp goal (BECAUSE _ SMP _ _) goal'
  | exprSIMP goal == goal'  =  rep ("OK: use of SIMP is correct.")
  | otherwise               =  rep ("!!: use of SIMP differs.")

checkStep mdls thrys hyp goal (BECAUSE _ IH howused what) goal'
 = case findAndApplyLAW (mdlsKnown mdls) (LAW "IH" hyp) goal howused what of
     Nothing -> rep ("!!: Failed to apply IH "++show howused)
     Just goal''
       -> if goal'' == goal'
           then rep ("OK: use of IH "++show howused++" is correct.")
           else rep ("!!: use of IH "++show howused++" differs.")
```

We need all names defined in imported haskell files:

```
mdlsKnown = concat . map mdlKnown

mdlKnown mdl = getDefined $ topdecls mdl

getDefined [] = []
getDefined (Fun (m:_)         : tdcls)  = fname m : getDefined tdcls
```

```
getDefined (Bind (Var v) _ _ : tdcls)  = v        : getDefined tdcls
getDefined (_                : tdcls)  =            getDefined tdcls
```

```
type Definition = (Expr,Expr,[Decl])

searchMods [] dnm i = Nothing
searchMods (mdl:mdls) dnm i
  = case searchDecls (topdecls mdl) dnm i of
      Nothing   ->  searchMods mdls dnm i
      jdefn     ->  jdefn
```

```
searchDecls [] dnm i = Nothing
searchDecls (decl:decls) dnm i
  = case checkDecl dnm i decl of
      Nothing -> searchDecls decls dnm i
      jdefn -> jdefn
```

```
checkDecl :: String -> Int -> Decl -> Maybe Definition

checkDecl dnm i (Bind v@(Var vnm) defn ldcls)
  | dnm == vnm && i < 2   =   Just (v,defn,ldcls)
  -- only do simple  v = e where ... binds for now

checkDecl dnm i (Fun [match])
  | dnm == fname match && i < 2
                         = Just (mkLHS dnm match,rhs match, ldecls match)
checkDecl dnm i (Fun matches)
  | i < 1   =  Nothing
  | i > length matches   =  Nothing
  | dnm == fname match   = Just (mkLHS dnm match,rhs match, ldecls match)
  where
    match = matches !! (i-1)
checkDecl _ _ _ = Nothing

mkLHS dnm match = mkApp (Var dnm) $ lhspat match

mkApp f [] = f
mkApp f (a:as)  = mkApp (App f a) as
```

This does an in-order traverse of the `goal` looking for the sub-expression defined by `what`. Once found, it will use `defn` to rewrite that sub-expression.

```
findAndApplyDEFN :: [String] -> Definition -> Expr -> Usage -> Focus
                 -> Maybe Expr
findAndApplyDEFN knowns defn goal howused Top
  = applyDEFN knowns howused defn goal

findAndApplyDEFN knowns defn goal howused (At nm i)
  = case pathToIndicatedName goal nm i of
      Nothing -> Nothing
      Just path
       -> applyAtPathFocus (applyDEFN knowns howused defn) path goal
```

```
applyDEFN :: [String] -> Usage -> Definition -> Expr -> Maybe Expr

applyDEFN knowns L2R (lhs,rhs,ldcls) expr
  = case eMatch knowns expr lhs of
      Nothing -> Nothing
      Just bind -> Just $ buildReplacement bind ldcls rhs

applyDEFN knowns R2L (lhs,rhs,ldcls) expr
```

```
  = case eMatch knowns expr rhs of
      Nothing -> Nothing
      Just bind -> Just $ buildReplacement bind ldcls lhs
```

```
findTheoryLaws [] lnm = Nothing
findTheoryLaws (thry:thrys) lnm
  = case searchLaws (thLaws thry) lnm of
      Nothing  ->  findTheoryLaws thrys lnm
      jlaw     ->  jlaw

searchLaws [] lnm = Nothing
searchLaws (lw:laws) lnm
  | lawName lw == lnm  =  Just lw
  | otherwise  = searchLaws laws lnm
```

```
findTheoryInds [] typ = Nothing
findTheoryInds (thry:thrys) typ
  = case searchInds (thIndScheme thry) typ of
      Nothing  ->  findTheoryInds thrys typ
      inds     ->  inds

searchInds [] typ = Nothing
searchInds (inds:indss) typ
  | indType inds == typ  =  Just inds
  | otherwise  = searchInds indss typ
```

This does an in-order traverse of the `goal` looking for the sub-expression defined by `what`. Once found, it will use `thelaw`, according to `howused`, to rewrite that sub-expression.

```
findAndApplyLAW :: [String] -> Law -> Expr -> Usage -> Focus -> Maybe Expr

findAndApplyLAW knowns thelaw goal howused Top
 = applyLAW knowns howused (lawEqn thelaw) goal

findAndApplyLAW knowns thelaw goal howused (At nm i)
  = case pathToIndicatedName goal nm i of
      Nothing -> Nothing
      Just path
       -> applyAtPathFocus (applyLAW knowns howused $ lawEqn thelaw) path goal
```

```
applyLAW :: [String] -> Usage -> Expr -> Expr -> Maybe Expr

applyLAW knowns Whole thelaw expr
  = case eMatch knowns expr thelaw of
      Nothing -> Nothing
      Just _ -> Just $ LBool True

applyLAW knowns L2R (Equal lhs rhs) expr
  = case eMatch knowns expr lhs of
      Nothing -> Nothing
      Just bind -> Just $ buildReplacement bind [] rhs

applyLAW knowns R2L (Equal lhs rhs) expr
  = case eMatch knowns expr rhs of
      Nothing -> Nothing
      Just bind -> Just $ buildReplacement bind [] lhs
```

Focus Handling

Consider we are looking for the ith occurrence of name `f` in an expression, and it is found embedded somehere, and is a function name applied to several arguments: `.... f x y z .....` What we want returned is a pointer to that full application, and not just to `f`. However, this means that the location of `f` can be arbitrarily deep down the lefthand branch of an `App`, as the above application will parse as $@(@(@\ f\ x)\ y)\ z$. If the application has path ρ, then the path to the occurrence of f will be $\rho \frown \langle 1,1,1 \rangle$. So we can delete trailing ones to get up to the correct location in this case. However if `f` occurs in an if-expression (say), like `if f then x else y`, then if the if-expression has path ρ, then f has path $\rho \frown \langle 1 \rangle$, but this last one needs to remain. In effect we have to tag the indices to indicate if we are branching through an application (@) or some other kind of node (e.g., if).

```
-- we only care about App vs everything else right now
data ExprBranches = AppB | OtherB deriving (Eq, Show)
type Branch = (Int,ExprBranches)
type Path = [Branch] -- identify sub-expr by sequence of branch indices
findAllNameUsage :: String -> Path -> Expr -> [Path]
-- paths returned here are reversed, with deepest index first

findAllNameUsage nm currPath (App (Var v) e2)
  | nm == v  = currPath : findAllNameUsage nm ((2,AppB):currPath) e2

findAllNameUsage nm currPath (Var v) = if nm == v then [currPath] else []

findAllNameUsage nm currPath (App e1 e2)
  =  findAllNameUsage nm ((1,AppB):currPath) e1
  ++ findAllNameUsage nm ((2,AppB):currPath) e2

findAllNameUsage nm currPath (If e1 e2 e3)
  =  findAllNameUsage nm ((1,OtherB):currPath) e1
  ++ findAllNameUsage nm ((2,OtherB):currPath) e2
  ++ findAllNameUsage nm ((3,OtherB):currPath) e3

findAllNameUsage nm currPath (GrdExpr grds)
  = concat $ map (findGuardNameUsage nm currPath) $ zip [1..] grds

findAllNameUsage _ _ (LBool _) =  []
findAllNameUsage _ _ (LInt  _) =  []
findAllNameUsage _ _ (LChar _) =  []

findAllNameUsage nm currPath e = error ("findAllNameUsage NYIf "++show e)
```

```
findGuardNameUsage nm currPath (i,(grd,res))
  =    findAllNameUsage nm ((1,OtherB):cp') grd
    ++ findAllNameUsage nm ((2,OtherB):cp') res
  where cp' = (i,OtherB):currPath
```

```
getIth :: Int -> [a] -> Maybe a
getIth _ []       =  Nothing
getIth 1 (x:_)    =  Just x
getIth n (_:xs)   =  getIth (n-1) xs

replIth :: Int -> a -> [a] -> Maybe [a]
replIth _ _ []        =  Nothing
replIth 1 x' (x:xs)   =  Just (x':xs)
replIth n x' (x:xs)   =  do xs' <- replIth (n-1) x' xs
                            return (x:xs')
```

Given an expression (e), a name (n), and an integer i, locate the ith (inorder) "effective occurence" of n in e. By "effective occurrence" we mean that if the name is of an applied function then we want the sub-expression that corresponds to the application of that function to all its arguments. For example, given $(h\ f) + f\ x\ y + 1$, the first effective occurrence of f is just the f that is the argument to h, while the second effective occurrence is the whole application $f\ x\ y$.

```
pathToIndicatedName :: Expr -> String -> Int -> Maybe [Int]
pathToIndicatedName goal nm i
 = case findAllNameUsage nm [] goal of
      [] -> Nothing
      paths
         -> case getIth i paths of
             Nothing -> Nothing
             Just path -> Just $ reverse $ map fst $ dropWhile isApp1 path
  where
    isApp1 (1,AppB)   =  True
    isApp1 _          =  False
```

```
applyAtPathFocus :: (Expr -> Maybe Expr) -> [Int] -> Expr -> Maybe Expr
applyAtPathFocus replace []     goal = replace goal
applyAtPathFocus replace (i:is) (App e1 e2)
  | i == 1   =  do e1' <- applyAtPathFocus replace is e1
                   return $ App e1' e2
  | i == 2   =  do e2' <- applyAtPathFocus replace is e2
                   return $ App e1 e2'
applyAtPathFocus replace (i:is) (If e1 e2 e3)
  | i == 1   =  do e1' <- applyAtPathFocus replace is e1
                   return $ If e1' e2 e3
  | i == 2   =  do e2' <- applyAtPathFocus replace is e2
                   return $ If e1 e2' e3
  | i == 3   =  do e3' <- applyAtPathFocus replace is e3
                   return $ If e1 e2 e3'
applyAtPathFocus replace (i:j:is) (GrdExpr eps)
  = do (e1,e2) <- getIth i eps
       if      j == 1 then do e1'   <-  applyAtPathFocus replace is e1
                              eps'  <-  replIth i (e1',e2) eps
                              return $ GrdExpr eps'
       else if j == 2 then do e2'   <-  applyAtPathFocus replace is e2
                              eps'  <-  replIth i (e1,e2') eps
                              return $ GrdExpr eps'
       else Nothing
applyAtPathFocus replace (i:is) (Let dcls e)    =  Nothing
applyAtPathFocus replace (i:is) (PApp nm es)    =  Nothing

applyAtPathFocus replace (i:is) goal = Nothing
```

Builtin-simplifier

```
exprSIMP :: Expr -> Expr
exprSIMP (InfixApp e1 op e2)  =  applyOp op (exprSIMP e1) (exprSIMP e2)
exprSIMP (App e1 e2)          =  App (exprSIMP e1) (exprSIMP e2)
exprSIMP (If e1 e2 e3)        =  If (exprSIMP e1) (exprSIMP e2) (exprSIMP e3)
exprSIMP (GrdExpr eps)        =  GrdExpr $ map exprSIMP2 eps
exprSIMP (Let dcls e)         =  Let dcls $ exprSIMP e
exprSIMP (PApp nm es)         =  PApp nm $ map exprSIMP es
exprSIMP e                    =  e

exprSIMP2 (e1,e2)             =  (exprSIMP e1,exprSIMP e2)
```

The fun part:

```
applyOp "+"  (LInt x) (LInt y)  =  LInt  (x+y)
```

```haskell
applyOp "-"  (LInt x) (LInt y)  =  LInt  (x-y)
applyOp "*"  (LInt x) (LInt y)  =  LInt  (x*y)
applyOp "==" (LInt x) (LInt y)  =  LBool (x==y)
applyOp "/=" (LInt x) (LInt y)  =  LBool (x/=y)
applyOp "<"  (LInt x) (LInt y)  =  LBool (x<y)
applyOp "<=" (LInt x) (LInt y)  =  LBool (x<=y)
applyOp ">"  (LInt x) (LInt y)  =  LBool (x>y)
applyOp ">=" (LInt x) (LInt y)  =  LBool (x>=y)
applyOp op e1 e2 = InfixApp e1 op e2
```