This program reads in a textual representation of a comprehensive state characterization (CSC) and calls a model counter on each of its lower bounds and upper bounds. The output is the sum of the lower bound count ($\underline{I}_\psi$), the maximum counting time in seconds, and the count of each partition-with-a-gap (delimited by colons).

⟨*⟩≡
  ⟨*Libraries*⟩
  ⟨*Types*⟩
  ⟨*Functions*⟩

```
main :: IO ()
main = do
```
    ⟨*Parse argument*⟩
    ⟨*Collect partitions*⟩
    ⟨*Call model counter on each partition*⟩
```
    return ()
```

We just expect a single argument—the name of the CSC file we would like to count. We abort if there's not exactly one argument or if the file doesn't exist.

⟨*Parse argument*⟩≡
```
args <- getArgs
if ((length args) /= 1)
  then error "Expecting one arg. Usage: count CSC_FILE"
  else return ()

let cscFile = head args
fileExists <- doesFileExist cscFile
if (not fileExists)
  then error $ "The file "++cscFile++" does not exist."
  else return ()
```

⟨*Libraries*⟩≡
```
import System.Environment (getArgs)
import System.Directory (doesFileExist)
```

Each CSC file will contain partitions delimited by a blank line. Let's collect those string chunks by splitting on a double newline, and then make partitions out of each chunk.

⟨*Collect partitions*⟩≡
```
fileStr <- readFile cscFile
let partitionStrs = splitOn "\n\n" fileStr
partitions <- mapM makePartition partitionStrs
let nonoverlapParts = negatePrecedingPartitions partitions
```

⟨*Functions*⟩≡
```
negatePrecedingPartitions :: [Partition] -> [Partition]
negatePrecedingPartitions ps =
  if (all isExact ps)
    then ps
    else
      let idPs = zip [0..] ps
      in map (negateWorker ps) idPs

isExact :: Partition -> Bool
isExact TruePartition = True
isExact (Partition up lb) =
  if (length lb /= 1)
    then False
    else up == (head lb)

negateWorker :: [Partition] -> (Int,Partition) -> Partition
negateWorker _ (_,TruePartition) = TruePartition
negateWorker ps (i, (Partition p lbs)) =
  let negations = map negateP (take i ps)
      p' = makeConjunctionExpr ([p]++negations)
  in Partition p' lbs

negateP :: Partition -> CExpr
negateP (Partition p _) = CUnary CNegOp p undefNode
negateP _ = error "Should not be negating a TruePartition"
```

⟨*Libraries*⟩+≡
```
import Data.List.Split (splitOn)
import Data.Maybe (catMaybes)
```

Each exact partitions will have a characterizing formula (given as a C expression), and possibly some assumptions. We use the `CExpr` type from the `Language.C` library.

⟨*Types*⟩≡
```
data Partition = Partition CExpr [CExpr]
  | TruePartition
  deriving (Show,Eq)
```

⟨*Libraries*⟩+≡
```
import Language.C
```

The textual form of each partition, with indexed lines, is:

0. `Partition:`

1. `Upper Bound:`

2. `...`

3. `Lower Bound:`

4. `...`

We first split this string chunk into an array of lines, to correspond to the above numbering. To check if the bounds are exact, we just see if the line 2 is the same as line 4.

⟨*Functions*⟩+≡
```
makePartition :: String -> IO Partition
makePartition s = do
  let ls = filter (not . null) (lines s)
  if (length ls) < 3
    then return TruePartition
    else do
      let up = ls !! 2
          lo = ls !! 4
      if up /= lo
        then do
          upF <- parseToExpr (cleanUp up)
          let ls' = drop 4 ls
              ls'' = takeWhile (\l -> not $ "Assumptions:" `isInfixOf` l) ls'
          lowers <- mapM parseToExpr ls''
          return $ Partition upF lowers
        else do
          let up' = cleanUp up
          if null up'
            then return $ TruePartition
            else do
              formula <- parseToExpr up'
              return $ Partition formula [formula]

cleanUp :: String -> String
cleanUp s = trim $ head $ splitOn "MINUS" s
```

We import the function `parseToExpr` to parse the given legal C expression strings into a `Language.C` instance of a `CExpr`.

⟨*Libraries*⟩+≡
```
import Reading (parseToExpr)
import Data.Char (isSpace)
```

We call a model counter on each partition to find out how many satisfying solutions there are to a given formula.

⟨*Call model counter on each partition*⟩≡

```
results <- countPartitions nonoverlapParts
let psiCount = sum $ map (\(PartitionCount _ ls)->sum $ map (\(_,r)->r) ls) results
let upperTimes = map (\(PartitionCount (ut,_) _) -> ut) results
let lowerTimes = concat $ map (\(PartitionCount _ ls) -> map (\(t,_)->t) ls) results
let maxTime = maximum $ upperTimes++lowerTimes
let gapCounts = map gatherGapCount results

-- output: #(psi), #-time, #(gap1):#(gap2)...
putStrLn $ (display psiCount)++","++(show maxTime)++","++(gapCountStr gapCounts)
```

⟨*Functions*⟩+≡
```
gapCountStr :: [Rational] -> String
gapCountStr rs =
  let rs' = filter (>0) rs
      rs'' = map display rs'
  in
    if null rs''
      then "no-gap"
      else intercalate ":" rs''

gatherGapCount :: PartitionCount -> Rational
gatherGapCount (PartitionCount (_,uCount) ls) =
  let lCounts = map (\(_,c)->c) ls
      lSum = sum lCounts
      gapCount = uCount - lSum
  in gapCount

displayPartitionCount :: PartitionCount -> IO ()
displayPartitionCount (PartitionCount uCount lCounts) = do
  putStrLn $ "\nPartition:"
  putStrLn $ "  Upper:"
  displayCount uCount
  putStrLn $ "  Lower:"
  mapM_ displayCount lCounts

displayCount :: (Int,Rational) -> IO ()
displayCount (t,c) = putStrLn $ "     time: "++(show t)++", count:"++(display c)

countPartitions :: [Partition] -> IO [PartitionCount]
countPartitions ps = do
  let hasTruePC = any (\p -> p == TruePartition) ps
  if hasTruePC
    then return $ [PartitionCount (0,1 % 1) []]
    else mapM countPartition ps

data PartitionCount = PartitionCount {
  upperCount :: (Int,Rational),
  lowerCount :: [(Int,Rational)]
}

countPartition :: Partition -> IO PartitionCount
countPartition p = do
  upC <- modelCountUp p
  lowC <- modelCountLow p
  return $ PartitionCount upC lowC
countPartition _ = error "should have already processed the TruePartitions"
```

First we need to get the C expression(s) into SMTLib2 form—the format expected by PCP, our model counter. And to do this, we create a query using the SBV library.

⟨*Libraries*⟩+≡
```
import SolverLib (makeConjunctionExpr)
import System.Process (readProcessWithExitCode)
import Data.Ratio
import Numeric
import Data.Time
```

⟨*Functions*⟩+≡

```
modelCountUp :: Partition -> IO (Int,Rational)
modelCountUp TruePartition = return $ (0,1 % 1)
modelCountUp (Partition p _) = do
  r <- count p
  return r

modelCountLow :: Partition -> IO [(Int,Rational)]
modelCountLow TruePartition = return $ [(0,1 % 1)]
modelCountLow (Partition _ lowers) = do
  rs <- mapM count lowers
  return rs

pcpPath :: String
pcpPath =
  "/home/mitch/work/tools/PathConditionsProbability/startPCP"

count :: CExpr -> IO (Int,Rational)
count e = do
  if (show $ pretty e) == "1"
    then return (0,1 % 1)
    else do
      smtStr <- extractSMTQuery e
      let query = sanitizeSMT smtStr
          query' = addPcpHeader query
      writeFile "pcp.query" query'
      start <- getCurrentTime
      (_,out,_) <- readProcessWithExitCode pcpPath [] query
      end <- getCurrentTime
      let runtime = ceiling $ realToFrac $ diffUTCTime end start
      result <- parsePcpOut out
      return (runtime, result)

addPcpHeader :: String -> String
addPcpHeader s =
  "(set-option :non-linear-counter \"qcoral\")\n"++
  "(set-option :partitioning true)\n"++
  "(set-option :inclusion-exclusion false)\n"++
  "(set-option :linear-counter \"barvinok\")\n"++
  "(set-option :simplify \"z3\")\n"++s

display :: Rational -> String
display r = (show $ numerator r)++"/"++(show $ denominator r)

parsePcpOut :: String -> IO Rational
parsePcpOut o = do
```

```
  let ls = lines o
      (Just result) = find (isPrefixOf "(exact:") ls
  let f = init $ head $ tail $ words result
      [numer,denom] = splitOn "/" f
  return $ (read numer::Integer) % (read denom::Integer)
```

There are some acrobatics to go through in order to extract the SMT query. First we need to interpret this `CExpr` as an SBV expression. SBV is an interface to SMT solvers. We can then take this SBV predicate and generate an SMTLib2 script. (In the function `generateSMTBenchmark`, the first boolean argument controls whether this is a SAT instance, i.e., translate the query directly, or a PROVE instance, i.e., translate the negated query.)

⟨*Libraries*⟩+≡

```
import SolverLib (makePredicate)
import Data.SBV (generateSMTBenchmark)
```

⟨*Functions*⟩+≡

```
extractSMTQuery :: CExpr -> IO String
extractSMTQuery e = do
  generateSMTBenchmark True (makePredicate e)
```

We have to change some things in order to make SBV's auto-generated SMT query ready for `PCP`. Given the assertion:

$$
\begin{gathered}
X + 1 \leq 0 \wedge \\
X - 1073741823 \leq 0 \wedge \\
0 \leq X + 1073741823 \wedge \\
Y - 1073741823 \leq 0 \wedge \\
0 \leq Y + 1073741823
\end{gathered}
$$

SBV generates the below script:

```
; Automatically created by SBV on 2019-05-02 19:28:40.156551 EDT
(set-option :smtlib2_compliant true)
(set-option :diagnostic-output-channel "stdout")
(set-option :produce-models true)
(set-logic ALL) ; has unbounded values, using catch-all.
; --- uninterpreted sorts ---
; --- tuples ---
; --- literal constants ---
(define-fun s2 () Int 1)
(define-fun s4 () Int 0)
(define-fun s6 () Int 1073741823)
; --- skolem constants ---
(declare-fun s0 () Int) ; tracks user variable "X_int_0[0]"
(declare-fun s1 () Int) ; tracks user variable "X_int_1[0]"
; --- constant tables ---
; --- skolemized tables ---
; --- arrays ---
; --- uninterpreted constants ---
; --- user given axioms ---
; --- formula ---
(define-fun s3 () Int (+ s0 s2))
(define-fun s5 () Bool (<= s3 s4))
(define-fun s7 () Int (- s0 s6))
(define-fun s8 () Bool (<= s7 s4))
(define-fun s9 () Int (+ s0 s6))
(define-fun s10 () Bool (<= s4 s9))
(define-fun s11 () Bool (and s8 s10))
(define-fun s12 () Int (- s1 s6))
(define-fun s13 () Bool (<= s12 s4))
(define-fun s14 () Bool (and s11 s13))
(define-fun s15 () Int (+ s1 s6))
(define-fun s16 () Bool (<= s4 s15))
(define-fun s17 () Bool (and s14 s16))
(define-fun s18 () Bool (and s5 s17))
(assert s18)
(check-sat)
```

Here are Mateus' suggestions from an email:
"There are two things that need to be changed (well, 3 if you count replacing `(check-sat)` with `(count)`):

1. `PCP` doesn't support `define-fun`. It uses `declare-var` and `declare-const` instead:

   ```
   (declare-var x (Int -1000 1000))
   ```

```
;;declares a variable named 'x' which
;;is an integer in the range [-1000,1000]
(declare-const y Int 0)
;;declares a constant named 'y' which
;;corresponds to the integer 0
```

Boolean datatypes are implemented in PCP, but I must admit that I've never tested them much (int/floats are much more popular in the subjects I analysed during development). Send me a message if something behaves badly.

2. A sequence of `define-fun`s like below should be replaced by a sequence of assert expressions without "temporary" variables, i.e. replace each "temporary" variable recursively until you hit the actual variables (called 'skolem constants' in your file) and the constants.

```
(define-fun s3 () Int (+ s0 s2))
(define-fun s5 () Bool (<= s3 s4))
(define-fun s7 () Int (- s0 s6))
(define-fun s8 () Bool (<= s7 s4))
(define-fun s9 () Int (+ s0 s6))
(define-fun s10 () Bool (<= s4 s9))
(define-fun s11 () Bool (and s8 s10))
(define-fun s12 () Int (- s1 s6))
(define-fun s13 () Bool (<= s12 s4))
(define-fun s14 () Bool (and s11 s13))
(define-fun s15 () Int (+ s1 s6))
(define-fun s16 () Bool (<= s4 s15))
(define-fun s17 () Bool (and s14 s16))
(define-fun s18 () Bool (and s5 s17))
(assert s18)
```

"

⟨*Functions*⟩+≡

```
sanitizeSMT :: String -> String
sanitizeSMT s =
  let ss = lines s
      ⟨Split ss into constants, variables, and formula chunks⟩
      cs = transformConstants constants
      vs = transformVariables variables
      fs = transformFormula formula
  in unlines (cs++vs++fs)

slice from to xs = take (to - from + 1) (drop from xs)
```

The transformation of constants will be of the form:

```
(define-fun s2 () Int 1)
```

to

```
(declare-const s2 Int 1)
```

⟨*Functions*⟩+≡

```
transformConstants :: [String] -> [String]
transformConstants ss = map toConstant ss

toConstant :: String -> String
toConstant "; --- literal constants ---" =
  "; --- literal constants ---"
toConstant s =
  let tokens = splitOn " " s
  in
    if ((length tokens) > 3)
      then
        let varName = tokens !! 1
            ty = tokens !! 3
            suffix = last tokens
        in "(declare-const "++varName++" "++ty++" "++suffix
      else error "toConstant assumption violation"
```

The transformation of variables will be of the form:

```
(declare-fun s0 () Int)
```

to

```
(declare-var s0 (Int -1073741823 1073741823))
```

We use the constant bounds of 1073741823 because this is what is in CIVL. (The standard maximum value for a 32-bit integer in C is not given, because this was reaping edge-case havoc with CPAchecker and over/underflow. And within our SBV translation, the type is always a symbolic integer, so we'll just hardcode that type.

⟨*Functions*⟩+≡

```
transformVariables :: [String] -> [String]
transformVariables ss = map toVariable ss

toVariable :: String -> String
toVariable "; --- skolem constants ---" =
  "; --- skolem constants ---"
toVariable s =
  if (length (splitOn " " s)) > 1
    then
      let varName = (splitOn " " s) !! 1
      in "(declare-var "++varName++" (Int -1073741823 1073741823))"
    else error "toVariable assertion violation"

transformFormula :: [String] -> [String]
transformFormula ss =
  let table = foldr processLine Map.empty ss
      (Just assertIdx) = findIndex (\l-> "(assert " `isPrefixOf` l) ss
      aLine = ss !! assertIdx
      kernel = init $ (splitOn " " aLine) !! 1
      assertion = "(assert "++(expandExpr kernel table)++")"
  in ["; --- formula ---", assertion, "(count)"]

expandExpr :: String -> Map.Map String (Op, Var, Var) -> String
expandExpr expr m =
  case Map.lookup expr m of
    Nothing -> expr
    Just ("distinct", v1, v2) ->
      "(not (= "++(expandExpr v1 m)++" "++(expandExpr v2 m)++"))"
    Just (op, v1, "UNARY") ->
      "("++op++" "++(expandExpr v1 m)++")"
    Just (op, v1, v2) ->
      "("++op++" "++(expandExpr v1 m)++" "++(expandExpr v2 m)++")"

processLine :: String -> LookupTable -> LookupTable
processLine s m =
```

```
    if (not ("(define-fun " `isPrefixOf` s))
      then m
      else Map.insert (grabKey s) (makeEntry s) m

  grabKey :: String -> String
  grabKey s =
    if (length (splitOn " " s)) > 0
      then
        let ss = splitOn " " s
        in (ss !! 1)
      else error "grabKey assertion violation"

  makeEntry :: String -> (Op, Var, Var)
  makeEntry s =
    if (length (splitOn " " s)) > 6
      then
        let ss = splitOn " " s
            op = tail $ ss !! 4
            v1 = ss !! 5
            v2 = ss !! 6
            v2' = takeWhile (/=')') v2
        in (op, v1, v2')
      else
        let ss = splitOn " " s
            op = tail $ ss !! 4
            v1 = ss !! 5
            v1' = takeWhile (/=')') v1
        in (op, v1', "UNARY")

  trim = dropWhileEnd isSpace . dropWhile isSpace
```

⟨*Types*⟩+≡

```
  type Op = String
  type Var = String
  type LookupTable = Map.Map String (Op, Var, Var)
```

⟨*Libraries*⟩+≡

```
  import qualified Data.Map.Strict as Map
  import Data.List (findIndex, isPrefixOf, find, any, dropWhileEnd, isInfixOf, intercala
```

⟨*Split* `ss` *into* `constants`*,* `variables`*, and* `formula` *chunks*⟩≡
```
(Just constantIdx) = elemIndex "; --- literal constants ---" ss
(Just variableIdx) = elemIndex "; --- skolem constants ---" ss
(Just varEndIdx) = elemIndex "; --- constant tables ---" ss
(Just formulaIdx) = elemIndex "; --- formula ---" ss
formEndIdx = length ss

constants = slice (constantIdx) (variableIdx-1) ss
variables = slice (variableIdx) (varEndIdx-1) ss
formula = slice (formulaIdx) (formEndIdx-1) ss
```

⟨*Libraries*⟩+≡
```
import Data.List (elemIndex)
```