```haskell
module Homework1 where

import Prelude hiding (Num)

{---------------------- 1A ---------------------}

data Cmd = Pen Mode
         | MoveTo Pos Pos
         | Def Name Pars Cmd
         | Call Name Vals
         | Seq [Cmd]
         deriving Show

data Mode = Up | Down

data Pos = M Num | N Name

type Pars = [Name]

type Vals = [Num]

-- Terminal Symbol types
type Num = Int
type Name = String

-- Show implementations
instance Show Mode where
    show Up   = "up"
    show Down = "down"

instance Show Pos where
    show (M a) = show a

{---------------------- 1B ---------------------}
-- Vector macro
-- Assume the pen is down...
--  def vector (x1, y1, x2, y2)
--      pen up
--      moveto (x1, y1)
--      pen down
--      moveto (x2, y2)
--      pen up

```

```
vector = Def "vector"
             ["x1","y1","x2","y2"]
             (Seq [Pen Up,
                   MoveTo (N "x1") (N "y1"),
                   Pen Down,
                   MoveTo (N "x2") (N "y2"),
                   Pen Up])

{---------------------- 1C --------------------}
steps :: Int -> Cmd
steps a | a <= 1  = Seq [Call "vector" [0, 0, 0, 1],
                         Call "vector" [0, 1, 1, 1]]
steps a           = Seq [steps (a-1),
                                Seq [Call "vector" [a-1, a-1, a-1, a],
                                       Call "vector" [a-1, a, a, a]]]


{---------------------- 2A --------------------}
data GateFn = And | Or | Xor | Not
              deriving Show

data Pair = Pair Int Int

data Gates = Gate Int GateFn Gates | GNone

data Links = Link Pair Pair Links | LNone

data Circuit = Circuit Gates Links

{---------------------- 2B --------------------}

halfadder = Circuit (Gate 1 Xor
                    (Gate 2 And
                     GNone))
                    (Link (Pair 1 1) (Pair 2 1)
                    (Link (Pair 1 2) (Pair 2 2)
                     LNone))

{---------------------- 2C --------------------}

printPair :: Pair -> String
printPair (Pair x y) = show x ++ "." ++ show y

printGates :: Gates -> String
```

```haskell
printGates GNone = ""
printGates (Gate i gfnc gates) = show i ++ ":" ++ show gfnc ++ ";\n" ++
                                 printGates gates

printLinks :: Links -> String
printLinks LNone = ""
printLinks (Link pair1 pair2 links) = "from " ++ printPair pair1 ++ " to " ++
                                      printPair pair2 ++ ";\n" ++ printLinks links

prettyPrint :: Circuit -> String
prettyPrint (Circuit gates links) = printGates gates ++ printLinks links


{--------------- Exercise 3 --------------------}

-- Original Syntax:
data Expr = I Int
          | Plus Expr Expr
          | Times Expr Expr
          | Neg Expr

-- Alternative Syntax:
data Op = Add | Multiply | Negate
        deriving Show

data Exp = Num Int
         | Apply Op [Exp]
         deriving Show

{---------------------- 3A --------------------}

theExpression = Apply Multiply [ Apply Negate [Apply Add [Num 4, Num 3]], Num 7]

{---------------------- 3B --------------------

The alternative syntax can apply an operator to an arbitrary list of
expressions, whereas the original syntax can only add two expressions,
multiply two expressions, or negate one expression. Thus, although the
alternative syntax is more flexible for the currently defined operators, it
would be more difficult to implement division in a way that makes sense.

---------------------- 3C --------------------}

translate :: Expr -> Exp
translate (I x) = (Num x)
```

```
138   translate (I x) = (Num x)
139   translate (Plus x y) = Apply Add [(translate x), (translate y)]
140   translate (Times x y) = Apply Multiply [translate x, translate y]
141   translate (Neg x) = Apply Negate [translate x]
142
```