```haskell
--------------------------
--Exercise 1. Mini Logo
--------------------------

--(a) Define the abstract syntax for Mini Logo as a Haskell data type.

type Name = String
type Numb = Int

data Pos = PI Numb | PS Name deriving Show
type PP = (Pos,Pos)

data Pars = Pa [Name] deriving Show
data Vals = Va [Numb] deriving Show
data Mode = Up | Down deriving Show

data Cmd = Pen Mode
         | Moto PP
         | Def Name Pars Cmds
         | Call Name Vals
         deriving Show
type Cmds = [Cmd]


--(b) Write a Mini Logo marco vector that draws a line from a given position (x1,y1) to a given position (x2,y2) and
--represent the marco in abstract syntax, that is, as a Haskell data type value.


--Concrete syntax:
--def vector (x1,y1,x2,y2) [(pen up),(moveto (x1,y1)),(pen down),(moveto (x2,y2))]

--Abstract syntax:
vector = Def "vector" (Pa ["x1","y1","x2","y2"]) [(Pen Up), (Moto (PS "x1",PS "y1")), (Pen Down), (Moto (PS "x2", PS "y2"))]

callVec = Call "vector" (Va [1,1,2,2])

--(c) Define a Haskell function steps :: Int -> Cmds that constructs a Mini Logo program which draws a stair of n steps.

steps :: Int -> Cmds
steps 0 = []
steps n  = steps (n-1)++[Call "vector" (Va [n-1,n-1,n-1,n]), Call "vector" (Va [n-1,n,n,n])]




---------------------------------------------------
--Exercise 2. Digital Circuit Design Language
---------------------------------------------------

--(a) Define the abstract syntax for the above language as a Haskell data type.

data Circuit = GL Gates Links deriving Show
type Numgafn = (Int, Gafn)
data Gates = GG Numgafn Gates | Nogate deriving Show
data Gafn = And|Or|Xor|Not deriving Show
type Gaport = (Int, Int)
data Links = From Gaport Gaport Links | Nolink deriving Show

--(b) Represent the half adder circuit in abstract syntax, that is, as a Haskell data type value.

```

```haskell
li = From (1,1) (2,1) (From (1,2) (2,2) Nolink)
ga = GG (1,Xor) (GG (2,And) Nogate)
halfAdder = GL ga li

--(c) Define a Haskell function that implements a pretty printer for the abstract syntax.

ppGafn :: Gafn -> String
ppGafn And = "and"
ppGafn Or = "or"
ppGafn Xor = "xor"
ppGafn Not = "not"

ppGates :: Gates -> String
ppGates Nogate = ""
ppGates (GG (a,b) c) = (show a)++":"++ppGafn b++";\n"++ppGates c

ppLinks :: Links -> String
ppLinks Nolink = ""
ppLinks (From (a,b) (c,d) e) = "from "++(show a)++"."++(show b)++" to "++(show c)++"."++(show d)++";\n"++ppLinks e

ppCircuit :: Circuit -> String
ppCircuit (GL a b) = ppGates a++ppLinks b



------------------------------------------
--Exercise 3. Design Abstract Syntax
------------------------------------------

data Expr = N Int
          | Plus Expr Expr
          | Neg Expr
          deriving Show

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

--(a) Represent the expression -(3+4)*7 in the alternative abstract syntax.

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

--(b) What are the advantages and disadvantages of either representation?

-- The definition of Expr is simpler than the combination of Op and Exp,
-- but we need to take care of the number of arguments for each operation.
-- The Op and Exp give us more freedom. For example, Op can be reused in
-- other definitions and [Exp] let us have arbitrary number of arguments.

--(c) Define a function translate :: Expr -> Exp that translates expressions
--given in the first abstract syntax into equivalent expressions in the second
--abstract syntax.

translate :: Expr -> Exp
translate (N a) = Num a
translate (Plus a b) = Apply Add [translate a, translate b]
translate (Neg a) = Apply Negate [translate a]

ta = translate (N 5)
tb = translate (Plus (N 3) (Neg (N 8)))
```