A Y86 Haskell Specification

```
data Y86Mach r a c =   RRMovl (r, r) (Y86Mach r a c)
                     | IRMovl (c, r) (Y86Mach r a c)
                     | Addl (r, r) (Y86Mach r a c)
                     | Subl (r, r) (Y86Mach r a c)
                     | Andl (r, r) (Y86Mach r a c)
                     | Xorl (r, r) (Y86Mach r a c)
                     | Jmp a (Y86Mach r a c)
                     | Je a (r, r) (Y86Mach r a c)
                     | Jne a (r, r) (Y86Mach r a c)
                     | Call a (Y86Mach r a c)
                     | Ret (Y86Mach r a c)
                     | Halt

type Reg = Int type Pos = Int

data Y86State = St { regs :: [Double],                  -- The
eight Y86 Registers
                     mach :: [Y86Operation Y86State ()] -- The entire Y86 Machine Progra
                   }

data Y86Operation s v = Op (s -> (v, s))

instance Monad (Y86Operation s)
   where return a     = Op (\x -> (a, x))
         (Op f) >>= g = Op (\s -> let (a, s2) = f s
                                      Op fun  = g a
                                  in fun s2)


-- Monadic Operations ------------------------------------------

opRRMovl :: Reg -> Reg -> (Y86Operation Y86State ()) opRRMovl r1
r2 = Op (\ps -> ((), St (set r2 (get r1 (regs ps)) (regs ps))
(mach ps)))

opIRMovl :: Double -> Reg -> (Y86Operation Y86State ()) opIRMovl c
r = Op (\ps -> ((), St (set r c (regs ps)) (mach ps)))


opAddl :: Reg -> Reg -> (Y86Operation Y86State ()) opAddl r1 r2 =
Op (\ps -> ((), St (set r2 (get r1 (regs ps) + get r2 (regs ps))
(regs ps)) (mach ps)))


opSubl :: Reg -> Reg -> (Y86Operation Y86State ()) opSubl r1 r2 =
Op (\ps -> ((), St (set r2 (get r1 (regs ps) - get r2 (regs ps))
(regs ps)) (mach ps)))

opJmp :: Pos -> (Y86Operation Y86State ()) opJmp a = Op (\ps ->
createJmp ps a)

createJmp :: Y86State -> Pos -> ((),Y86State) createJmp ps a =
((), snd nSt)
                 where jmach = (snd . splitAt (a-1)) (mach ps)
                       nSt   = exec (sequence jmach) ps

opJe :: Pos -> Reg -> Reg -> (Y86Operation Y86State ()) opJe a r1
r2 = Op (\ps -> createJe ps a r1 r2)

createJe :: Y86State -> Pos -> Reg -> Reg -> ((),Y86State)
createJe ps a r1 r2 = ((), if (get r1 rs == get r2 rs) then snd
nSt
                                                        else ps)
                      where jmach = (snd . splitAt (a-1)) (mach ps)
                            nSt   = exec (sequence jmach) ps
                            rs    = regs ps

opJne :: Pos -> Reg -> Reg -> (Y86Operation Y86State ()) opJne a
r1 r2 = Op (\ps -> createJne ps a r1 r2)

createJne :: Y86State -> Pos -> Reg -> Reg -> ((),Y86State)
createJne ps a r1 r2 = ((), if (get r1 rs /= get r2 rs) then snd
nSt
                                                         else ps)
                       where jmach = (snd . splitAt (a-1)) (mach ps)
                             nSt   = exec (sequence jmach) ps
                             rs    = regs ps


-- Aux Funcs ----------------------------------------------------

get :: Int -> [a] -> a get i = last . take i
```

```
set :: Int -> a -> [a] -> [a] set i a l = let (l1, l2) = splitAt
(i-1) l
            in l1 ++ a : tail l2

exec (Op f) = f


-- Show Instances ---------------------------------------------

instance Show Y86State where
  showsPrec _ t = showString " Y86Registers -> " . shows (regs t)


-- Tests ------------------------------------------------------

initSt :: [Y86Operation Y86State ()] -> Y86State initSt = St
(replicate 8 0)

prog1 :: [Y86Operation Y86State ()]

prog1 = [opIRMovl 4.0 1, opIRMovl 5.0 2, opAddl 1 2]

prog2 = [opIRMovl 1.0 1, opIRMovl 5.0 2, opIRMovl 8.0 3, opAddl 1
2, opJne 4 2 3]

exec1 = exec (sequence prog1) (initSt prog1)

exec2 = snd  (exec (sequence prog2) (initSt prog2))
```