stateful computation

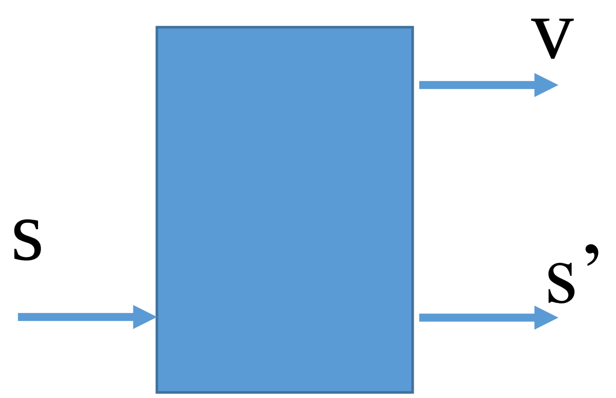
continuation

State Monad

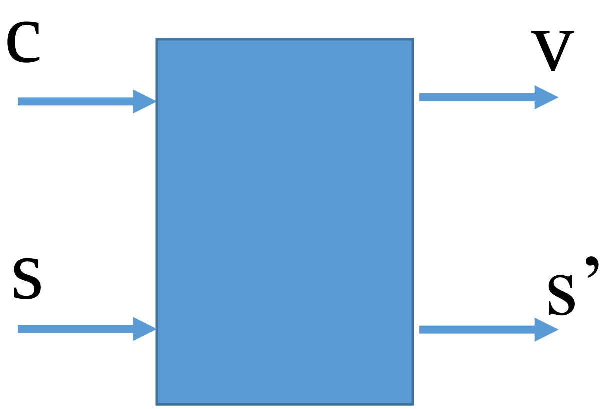
type State = Int

type ST = Int -> Int

type ST a = Int -> (a, Int)



Char -> ST a

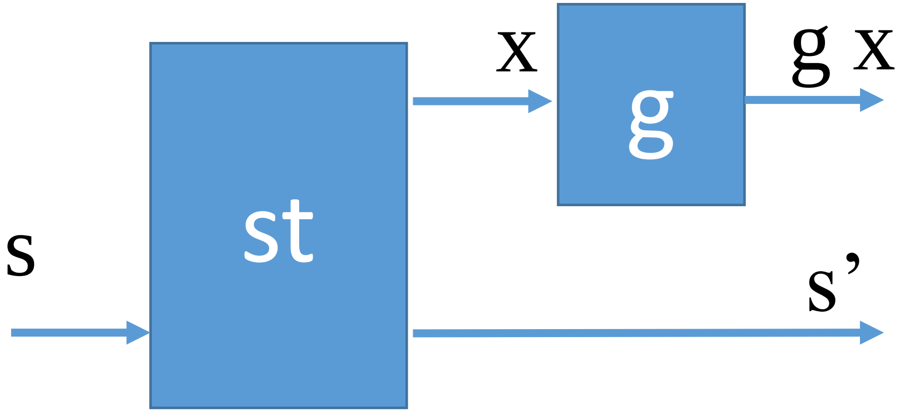


```
newtype ST a = S (State ->(a, State))

app :: ST a -> State ->(a, State)
app (S st) x = st x
```

ST a is a Functor, an Applicative and also a Monad

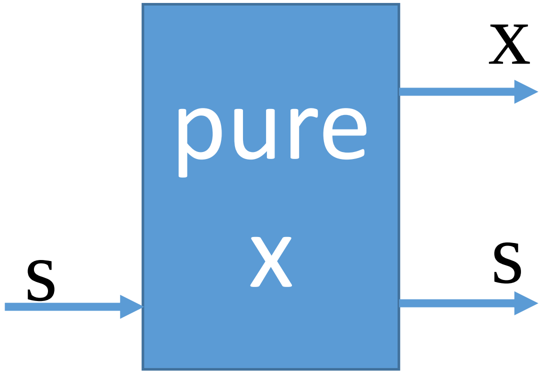
```
instance Functor ST where
--fmap :: (a -> b) -> ST a -> ST b
fmap g st = S(\ s -> let (x,s') = app st s in (g x, s'))
```

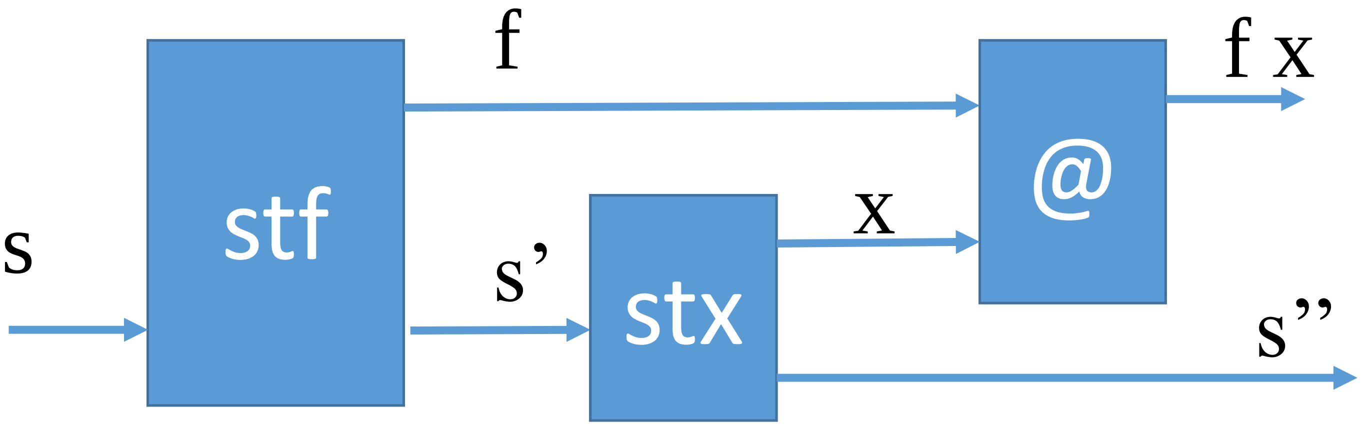


recall that in general fmap applies a function g to the values contained in a structure

in this case g is applied to the output of st

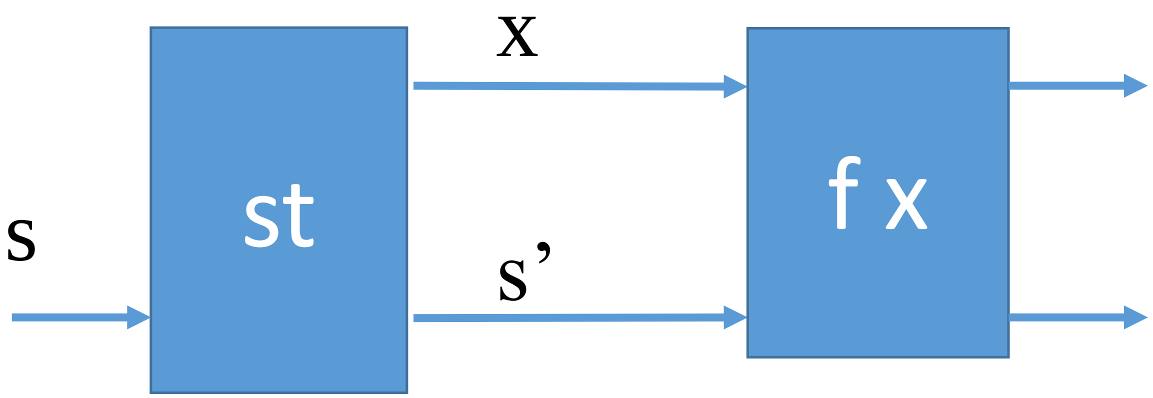
```
instance Applicative ST where
--pure :: a -> ST a
pure x = S (\s -> (x,s))
--(<*>) :: ST (a->b) -> ST a -> ST b
stf <*> stx = S (\s ->
        let (f, s’) = app stf s
            (x, s”) = app stx s’ in (f x, s”))
```





stf and stx are both ST and thus both produce a change of state

```
instance Monad  ST where
--(>>=) :: ST a -> (a -> ST b) -> ST b
st >>= f  =  S (\s -> let (x,s')=app st s in app (f x) s')
```



this ST depends on x

use ST a to define a function that takes a binay tree and relabels the leaves with fresh increasing integer according with the infix traversal 0, 1, 2, 3 .... or n, n+1, n+2,....

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
                    deriving Show
```

1st solution

```
rlabel :: Tree a -> Int -> (Tree Int, Int)
rlabel (Leaf _) n   = (Leaf n, n + 1)
rlabel (Node l r) n = (Node l’ r’, n’’)
                       where (l’,n’)   = rlabel l  n
                             (r’,n’’) = rlabel  r n’
```

this solution is complicated : we need to pass around a growing integer n. We can do better exploiting the fact that

```
rlabel :: Tree a -> Int -> (Tree Int, Int) ➔ Tree a -> ST (Tree Int)
```

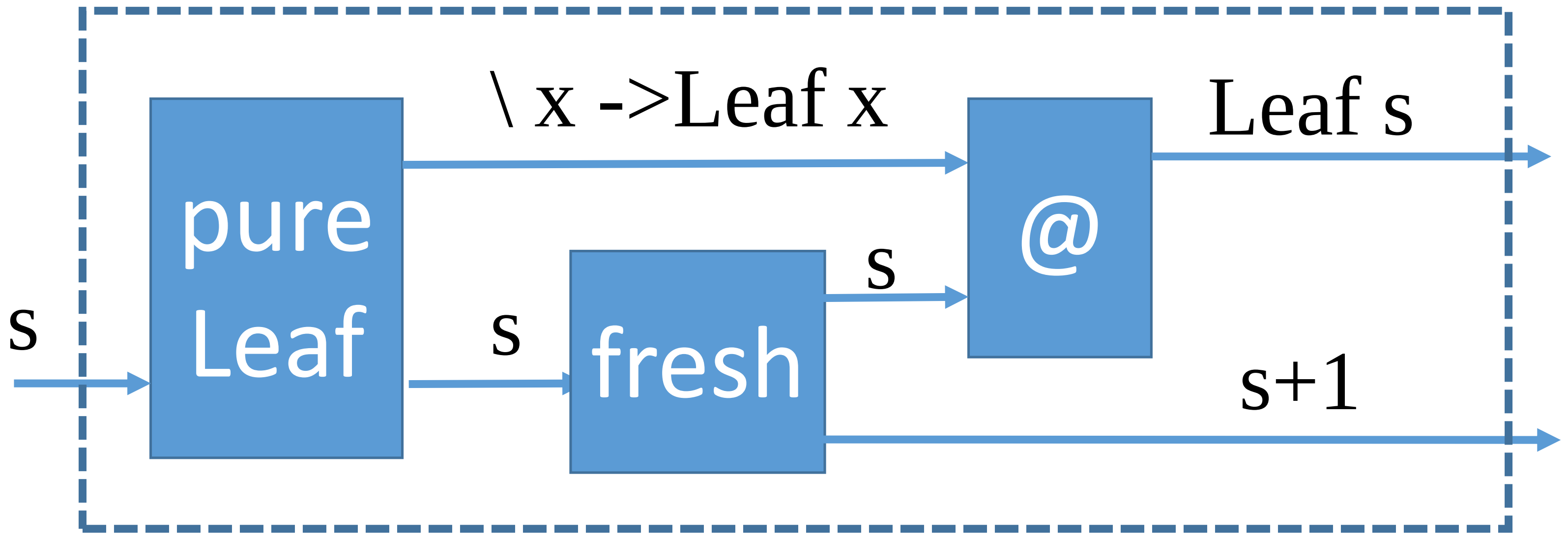
the integer that we passed around can be the State

```
fresh :: ST Int
fresh = S (\n -> (n, n+1))
```

we use the fact that ST a is Applicative

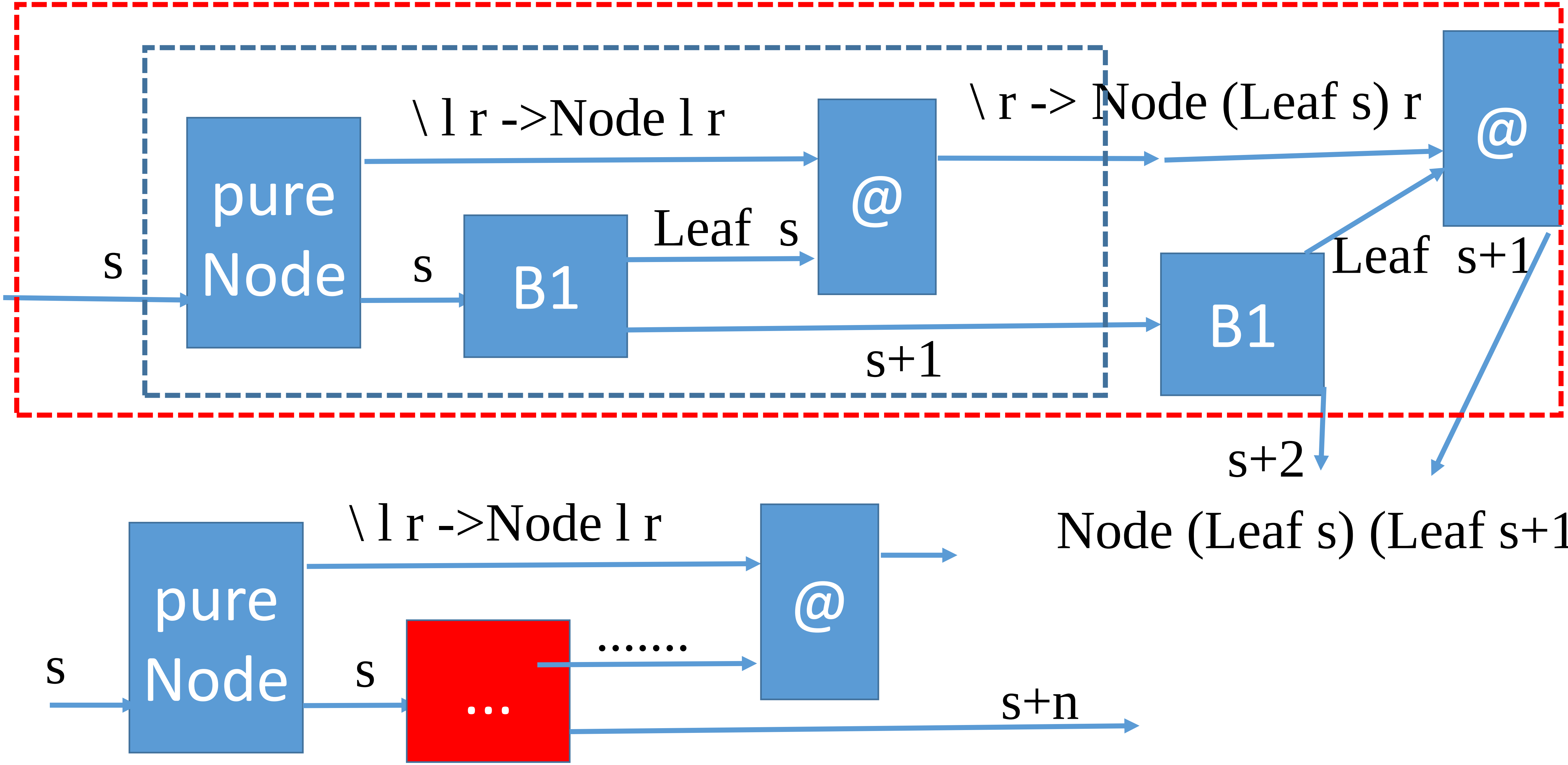
```
alabel :: Tree a -> ST (Tree Int)
alabel (Leaf _)     = pure Leaf <*> fresh
alabel (Node l r)   = pure Node <*> alabel l <*> alabel r
```

alabel (Leaf _) = pure Leaf <*> fresh :: ST (Tree Int)
= Int -> (Tree Int, Int)



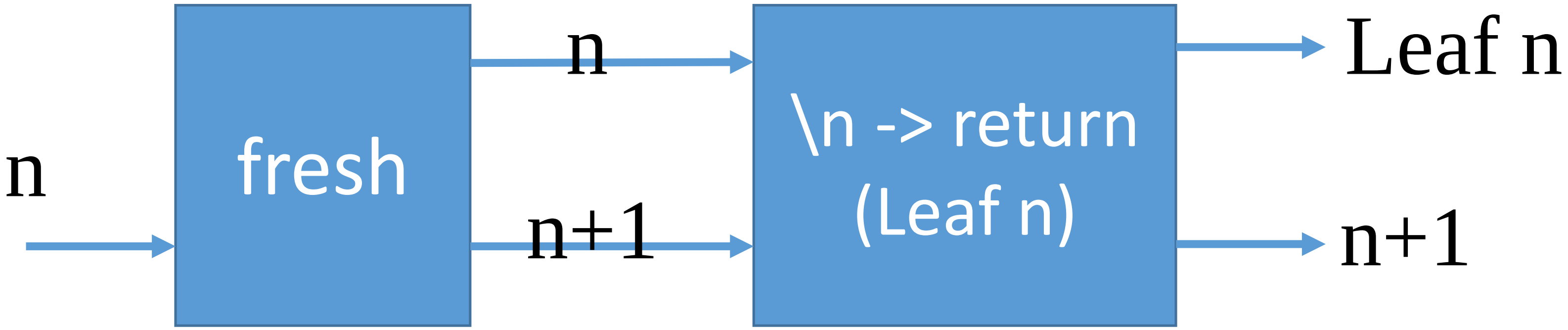
B1

alabel (Node l r) = pure Node <*> alabel l <*> alabel r



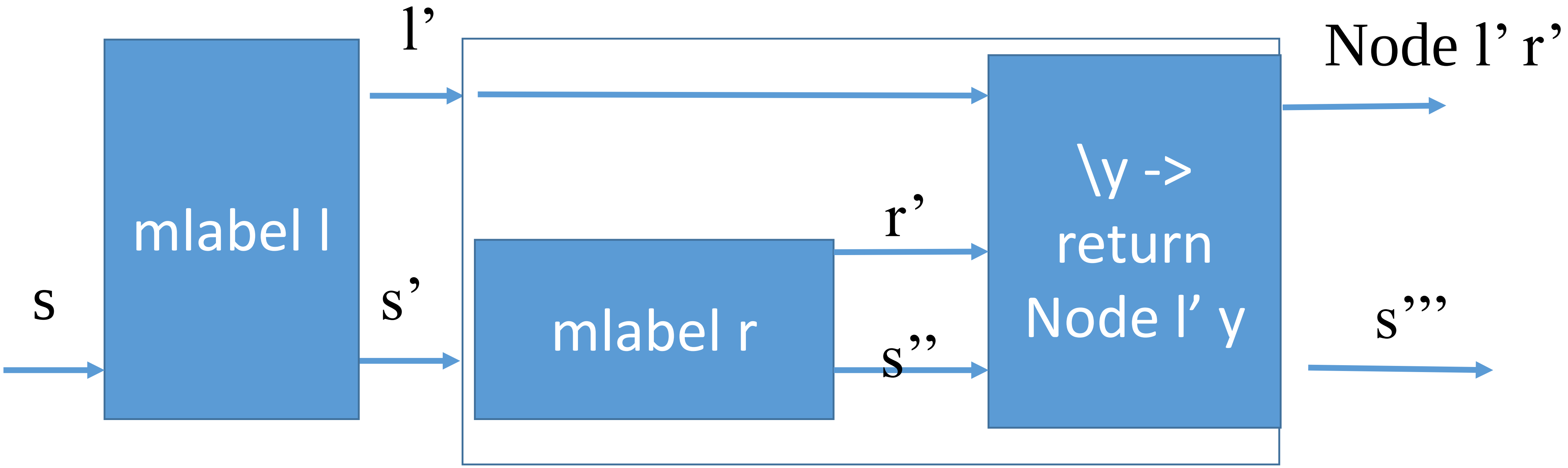
using the fact that ST is a Monad:

```
mlabel (Leaf _) = fresh >>= \n -> return Leaf  n
mlabel (Node l r) = mlabel l >>=
                        (\l’ -> mlabel r >>=
                        (\r’ -> return  Node l’  r’))
```



type
ST Tree Int

```
mlabel (Node l r) = mlabel l >>=
                    (\x -> mlabel r >>=
                    (\y -> return  Node x  y))
```



using the do notation of Monads

```
mlabel :: Tree a -> ST (Tree Int)
mlabel (Leaf _)    = do n <- fresh
                        return (Leaf n)
mlabel (Node l r)  = do l' <- mlabel l
                        r' <- mlabel r
                        return (Node l' r')
```

exercise

Tree a = Leaf a | Node (Tree a) a (Tree a)

```
mlabel (Leaf _)     = fresh >>= \n -> return (Leaf n)
mlabel (Node l _ r) = (mlabel l) >>=
                      (\l' -> fresh >>=
                      (\n -> (mlabel r) >>=
                      (\r' -> return (Node l' n r'))))
```

with do notation:

```
mlabel (Leaf _)     = do n <- fresh
                         return (Leaf n)

mlabel (Node l a r) = do l' <-  mlabel l
                         n <- fresh
                         r' <- mlabel r
                         return (Node l' n r')
```

generic functions

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f  []        = return  []
mapM f (x:xs)   =  do y <- f x
                      ys <- mapM f xs
                      return (y : ys)
conv :: Char -> Maybe Int
conv c | isDigit c   = Just (digitToInt c)
       | otherwise = Nothing
mapM conv «1234»
Just [1,2,3,4]
mapM conv «123b»   ➔ Nothing
```

```
filterM :: Monad m => (a -> m Bool) -> [a] ->m [a]

filterM p []        = return []
filterM p (x:xs) = do b <- px
                      ys <- filterM p xs
                       return (if b then x:ys else ys)

filterM (\x -> [True,False]) [1,2,3]
[[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[]]
```

```
join :: Monad m => m  (m a) -> m a
join mmx = do mx <- mmx
              x <- mx
              return x
```

```
join [[1],[2,3],[],[4]]
[1,2,3,4]

join (Just (Just 1))
Just 1
join (Just Nothing)
Nothing
```

Monad laws

1. return x >>= f   = f x
2. mx >>= return   = m x
3. (mx >>= f) >>= g = mx >>= (\x -> (f x >>= g))

check the types !!