Lesson 9

complete Applicatives and then Monads

The motivation for Applicatives is that of applying to containers functions of any arity

but also that of applying (pure) functions to arguments that have effects:
-failure
-non-determinism
-performing I/O

and also generic functions that use applicative operators in Prelude:

```
sequenceA :: Applicative f => [f a]-> f [a]
sequenceA []       =  pure []
sequenceA (x:xs) = pure (:) <*> x <*> sequenceA xs
```

```
getChars ::  Int -> IO String
getChars n = sequenceA (replicate n getChar)
```

Applicative laws

1) pure id <*> x = x
2) pure (g x) = pure g <*> pure x
3) x <*> pure y = pure (\g -> g y) <*> x
4) x <*> (y <*> z) = (pure (.) <*> x <*> y) <*> z

needs to work out types

1) id :: a -> a, x = f v:: f a => fmap id (f v) = f v = x
2) g :: a -> b e x :: a => pure (g x) :: f b

pure g :: f (a -> b), pure x :: f a, pure g <*> pure x :: f b

3) x :: f (a -> b), y :: a, pure (\g -> g y) :: f ((a -> b) ->b)

f ((a -> b) -> b) <*> x :: f b

```
4) x <*> (y <*> z) = (pure (.) <*> x <*> y) <*> z

y :: f (a -> b), z :: f a, (y <*> z) :: f b
x :: f (b -> c), (x <*> (y <*> z) ) :: f c

pure (.) :: f (b -> c) -> (a -> b) -> (a -> c)
pure (.) <*> x :: f (a -> b) -> (a -> c)
pure (.) <*> x <*> y :: f (a -> c)
(pure (.) <*> x <*> y) <*> z :: f c
```

is always true that
fmap g x = pure g <*> x

attention with [], (<*>) :: [a -> b] -> [a] -> [b]

pure g = [g] a list with 1 function only

fmap g x = g <$> x

g <$> x <*> y <*> z

Monads

we start with one example

```
data Expr = Val Int | Div Expr Expr

eval :: Expr -> Int
eval  (Val n)   =  n
eval (Div x y) =  eval x `div` eval y
```

possible fatal exception

```
eval (Div (Val 1) (Val 0))
***Exception: divide by zero
```

```
safediv :: Int -> Int -> Maybe Int
safediv _ 0  = Nothing
safediv n m = Just (n `div` m)
```

using it, we write an evaluator that is able to handle div by 0

```
eval :: Expr -> Maybe Int
eval (Val n)    = n
eval (Div x y) =case eval x of
                    Nothing -> Nothing
                    Just n     -> case eval y of
                                     Nothing -> Nothing
                                     Just m -> safediv n m
```

clearly with this eval,

```
eval (Div (Val 1) (Val 0))
Nothing
```

but eval is ugly, since maybe is Applicative, we could try to write eval in applicative style

```
eval :: Expr -> Maybe Int
eval (Val n)    = n
eval (Div x y) = pure safediv <*> eval x <*> eval y
```

But is not type correct !! safediv :: Int -> Int -> Maybe Int whereas we would need a Int -> Int -> Int

in the applicative style we can apply pure functions (as Int -> Int -> Int) to effectful arguments
but safediv may itself fail!

in eval there is a pattern that repeat

```
case eval x of
    Nothing -> Nothing
    Just m     -> case eval y of
                        Nothing ->  Nothing
                        Just n     -> safediv n m
```

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b

mx >>= f = case mx of
             Nothing -> Nothing
             Just x    -> f  x
```

bind operator

```
eval :: Expr -> Maybe Int
eval (Val n)     = Just n
eval (Div x y)  = eval x >>= \n ->
                  eval y >>= \m ->
                  safediv n m
```

generalizing, a typical expression with >>= is

```
m1 >>= \x1 ->
m2 >>= \x2 ->
.
.
mn >>= \xn ->
f x1 x2...xn

do x1 <- m1
   x2 <- m2
    ........
   f x1 ... xn
```

```
eval :: Expr -> Maybe Int
eval (Val n)     =  n
eval (Div x y) = do n <- eval x
                    m <- eval y
                    safediv n m

class Applicative m => Monad m where
return :: a -> m a
(>>=) :: m a -> (a -> m b) -> m b

return = pure
```

Examples

```
instance Monad Maybe where
-- (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
Nothing  >>= _    = Nothing
(Just  x)  >>= f     = f x

return = pure
```

```
instance Monad [] where
-- (>>=) :: [a] -> ( a -> [b]) -> [b]
xs >>= f = [y | x <-xs, y <- f x]


pairs :: [a] -> [b] -> [(a,b)]
pairs xs ys = do x <- xs
                 y <- ys
                 return (x,y)
```

Also IO type is a Monad

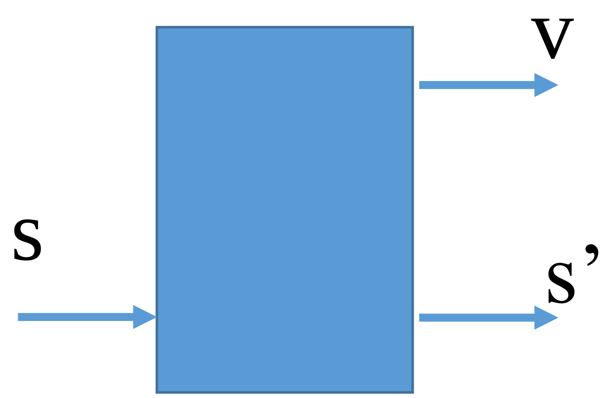
the definitions of return and (>>=) are built-in to the language.

State Monad

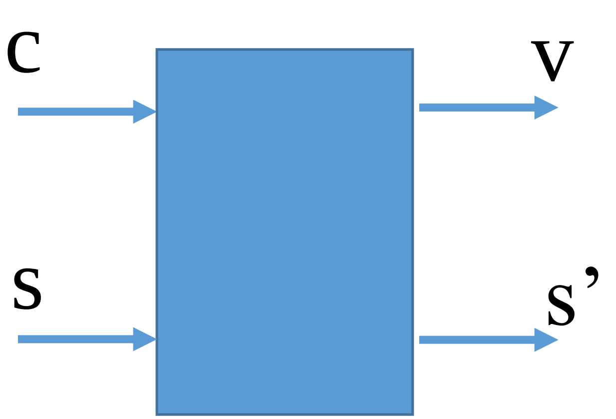
type State = Int

type ST = Int -> Int

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



Char -> ST a

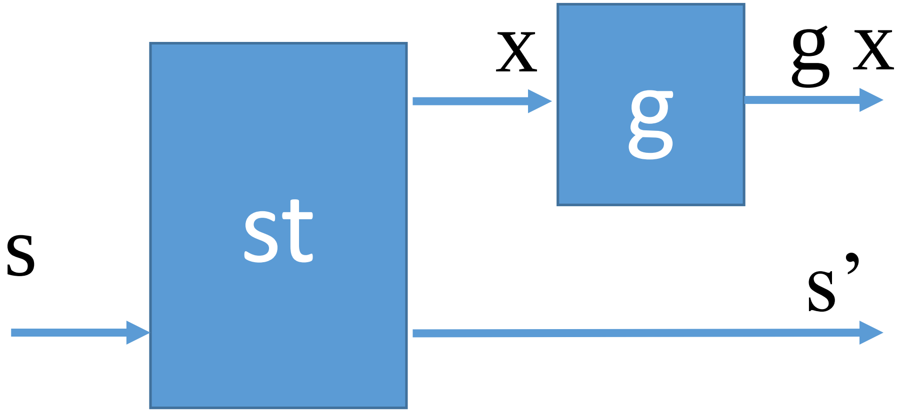


```
newtype ST a = S (State ->(a,
State))

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

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

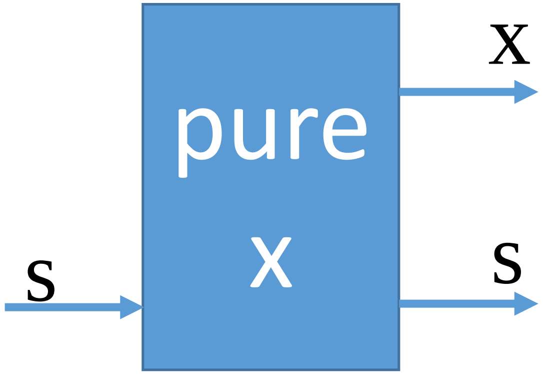
instance Functor ST where
--fmap :: (a -> b) -> ST a -> ST b
fmap g st = S(\ s -> let (x,s’) = app st s in (g x, s’))

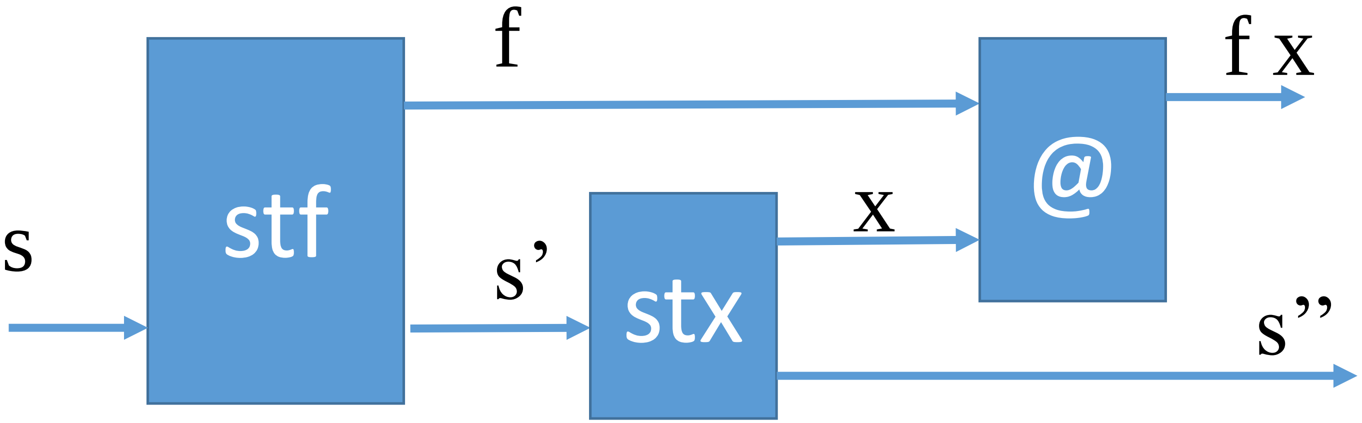


recall that in general fmap applies a function g to the values contained in a structure

in this case g is applied to the output of st

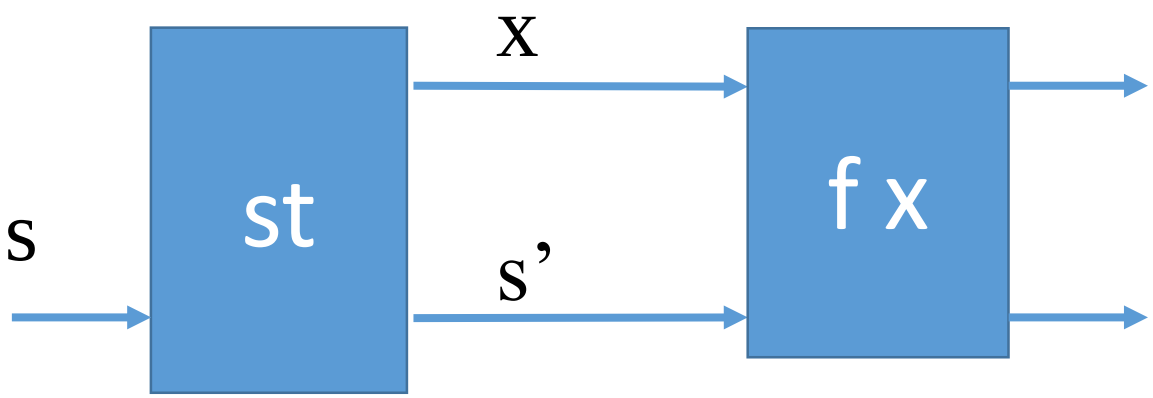
```haskell
instance Applicative ST where
--pure :: a -> ST a
pure x = S (\s -> (x,s))
--(<*>) :: ST (a->b) -> ST a -> ST b
stf <*> stx = S (\s ->
        let (f, s’) = appl stf s
            (x, s”) = app stx s’ in (f x, s”))
```





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) = Tree a -> ST (Tree 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)
```

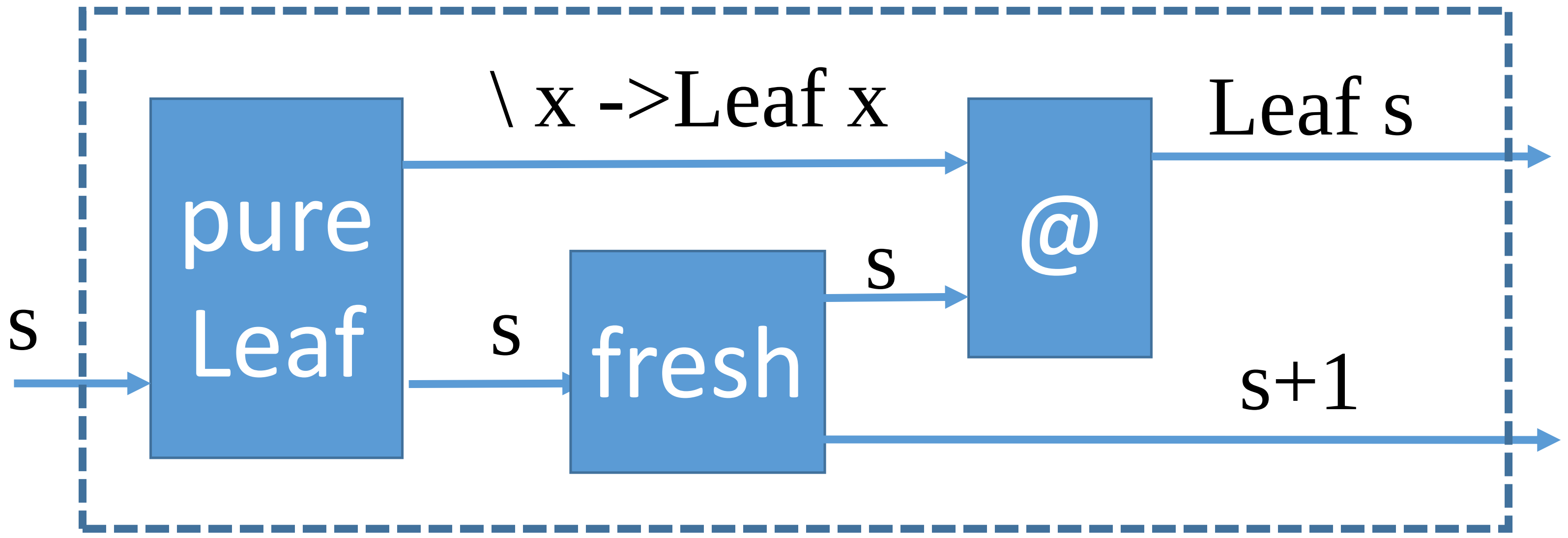
the integer that we passed around can be the State

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

we use that ST a is Associative

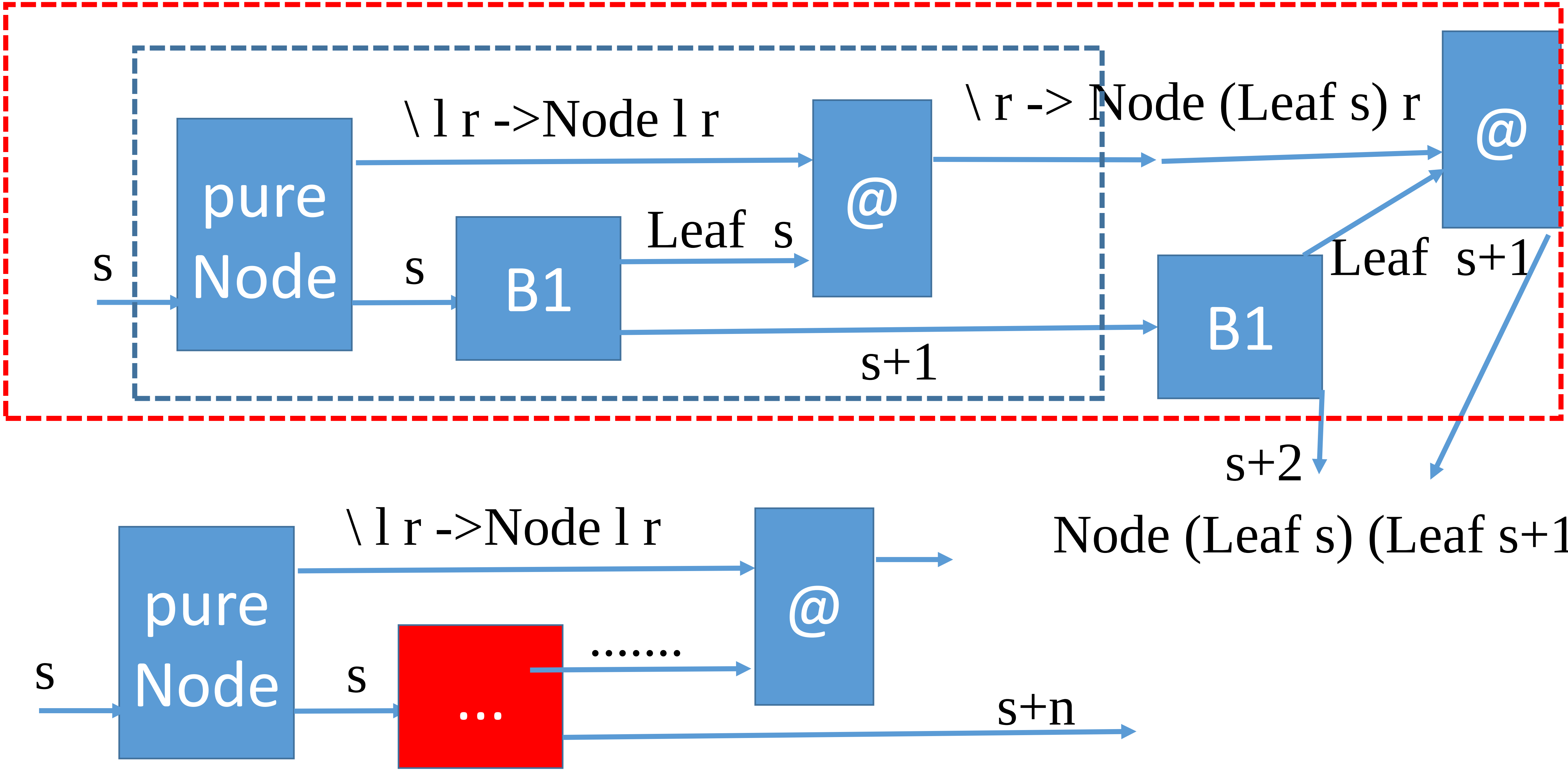
```
alabel :: Tree a -> ST (Tree Int)
alabel (Leaf _)     = pure Leaf <*> fresh
alabel (Node l r)   = pure Node <*> alabel l <*> alabel r
```

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 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’)
```