do-Notation for Monads

While APIs based on monads are very powerful, the explicit use of `>>=` with anonymous functions is still somewhat noisy. Just as infix operators are used instead of explicit calls to `HAdd.hAdd`, Lean provides a syntax for monads called *`do`-notation* that can make programs that use monads easier to read and write. This is the very same `do`-notation that is used to write programs in `IO`, and `IO` is also a monad.

In Hello, World!, the `do` syntax is used to combine `IO` actions, but the meaning of these programs is explained directly. Understanding how to program with monads means that `do` can now be explained in terms of how it translates into uses of the underlying monad operators.

The first translation of `do` is used when the only statement in the `do` is a single expression `E`. In this case, the `do` is removed, so

```
do E
```

translates to

```
E
```

The second translation is used when the first statement of the `do` is a `let` with an arrow, binding a local variable. This translates to a use of `>>=` together with a function that binds that very same variable, so

```
do let x ← E1
   Stmt
   ...
   En
```

translates to

```
E1 >>= fun x =>
do Stmt
   ...
   En
```

When the first statement of the `do` block is an expression, then it is considered to be a monadic action that returns `Unit`, so the function matches the `Unit` constructor and

```
do E1
   Stmt
   ...
   En
```

translates to

```
E1 >>= fun () =>
do Stmt
   ...
   En
```

Finally, when the first statement of the `do` block is a `let` that uses `:=`, the translated form is an ordinary let expression, so

```
do let x := E1
   Stmt
   ...
   En
```

translates to

```
let x := E1
do Stmt
   ...
   En
```

The definition of `firstThirdFifthSeventh` that uses the `Monad` class looks like this:

```
def firstThirdFifthSeventh [Monad m] (lookup : List α → Nat → m α) (xs : List
α) : m (α × α × α × α) :=
  lookup xs 0 >>= fun first =>
  lookup xs 2 >>= fun third =>
  lookup xs 4 >>= fun fifth =>
  lookup xs 6 >>= fun seventh =>
  pure (first, third, fifth, seventh)
```

Using `do`-notation, it becomes significantly more readable:

```
def firstThirdFifthSeventh [Monad m] (lookup : List α → Nat → m α) (xs : List
α) : m (α × α × α × α) := do
  let first ← lookup xs 0
  let third ← lookup xs 2
  let fifth ← lookup xs 4
  let seventh ← lookup xs 6
  pure (first, third, fifth, seventh)
```

Without the `Monad` type class, the function `number` that numbers the nodes of a tree was written:

```
def number (t : BinTree α) : BinTree (Nat × α) :=
  let rec helper : BinTree α → State Nat (BinTree (Nat × α))
    | BinTree.leaf => ok BinTree.leaf
    | BinTree.branch left x right =>
      helper left ~~> fun numberedLeft =>
      get ~~> fun n =>
      set (n + 1) ~~> fun () =>
      helper right ~~> fun numberedRight =>
      ok (BinTree.branch numberedLeft (n, x) numberedRight)
  (helper t 0).snd
```

With `Monad` and `do`, its definition is much less noisy:

```
def number (t : BinTree α) : BinTree (Nat × α) :=
  let rec helper : BinTree α → State Nat (BinTree (Nat × α))
    | BinTree.leaf => pure BinTree.leaf
    | BinTree.branch left x right => do
      let numberedLeft ← helper left
      let n ← get
      set (n + 1)
      let numberedRight ← helper right
      ok (BinTree.branch numberedLeft (n, x) numberedRight)
  (helper t 0).snd
```

All of the conveniences from `do` with `IO` are also available when using it with other monads. For example, nested actions also work in any monad. The original definition of `mapM` was:

```
def mapM [Monad m] (f : α → m β) : List α → m (List β)
  | [] => pure []
  | x :: xs =>
    f x >>= fun hd =>
    mapM f xs >>= fun tl =>
    pure (hd :: tl)
```

With `do`-notation, it can be written:

```
def mapM [Monad m] (f : α → m β) : List α → m (List β)
  | [] => pure []
  | x :: xs => do
    let hd ← f x
    let tl ← mapM f xs
    pure (hd :: tl)
```

Using nested actions makes it almost as short as the original non-monadic `map`:

```
def mapM [Monad m] (f : α → m β) : List α → m (List β)
  | [] => pure []
  | x :: xs => do
    pure ((← f x) :: (← mapM f xs))
```

Using nested actions, `number` can be made much more concise:

```
def increment : State Nat Nat := do
  let n ← get
  set (n + 1)
  pure n

def number (t : BinTree α) : BinTree (Nat × α) :=
  let rec helper : BinTree α → State Nat (BinTree (Nat × α))
    | BinTree.leaf => pure BinTree.leaf
    | BinTree.branch left x right => do
      pure (BinTree.branch (← helper left) ((← increment), x) (← helper
right))
  (helper t 0).snd
```

Exercises

- Rewrite `evaluateM`, its helpers, and the different specific use cases using `do`-notation instead of explicit calls to `>>=`.
- Rewrite `firstThirdFifthSeventh` using nested actions.