函数式语言程序设计

Monoid

半群 Semigroup，幺半群 Monoid

`Semigroup` 是关于满足结合律的二元操作的抽象，而 `Monoid` 是进一步关于单位元和满足结合律的二元操作的抽象，在实际编程中这类结构十分常见：

- 数字 `0` 和 `(+)`， `1` 和 `(*)`
- `[]` 和 `(++)`
- 逻辑值 `True` 和 `(&&)`，`False` 和 `(||)`

`...`

半群 Semigroup，幺半群 Monoid

```
class Semigroup a where
    (<>) :: a -> a -> a

class Semigroup a => Monoid a where
    mempty :: a
    mconcat :: [a] -> a
    mconcat = foldr mappend mempty

-- x <> (y <> z) = (x <> y) <> z
-- x <> mempty = x

instance Semigroup [a] where (<>) = (++)
instance Monoid [a] where mempty = []
```

半群 Semigroup，幺半群 Monoid

```
newtype Sum a = Sum { getSum :: a }
instance Num a => Semigroup (Sum a) where
    Sum x <> Sum y = Sum (x + y)
instance Num a => Monoid (Sum a) where
    mempty = Sum 0

newtype Product a = Product { getProduct :: a }
instance Num a => Semigroup (Product a) where
    Product x <> Product y = Product (x * y)
instance Num a => Monoid (Product a) where
    mempty = Product 1
```

半群 Semigroup，幺半群 Monoid

```
newtype All = All { getAll :: Bool }
instance Semigroup All where
    All x <> All y = All (x && y)
instance Monoid All where
    mempty = All True

newtype Any = Any { getAny :: Bool }
instance Semigroup Any where
    Any x <> Any y = Any (x || y)
instance Monoid Any where
    mempty = Any False
```

endomorphism

```
newtype Endo a = Endo { appEndo :: a -> a }

instance Semigroup (Endo a) where
    f <> g = f . g
instance Monoid (Endo a) where
    mempty = id

-- id . f = f
-- (f . g) . h = f . (g . h)
```

Monoid 与 Applicative

```
instance Monoid a => Applicative (Const a) where
    -- pure :: c -> Const a c
    pure _ = Const mempty
    -- (<*>) :: Const a (b -> c)
              -> Const a b -> Const a c
    Const x <*> Const y = Const (x <> y)

-- Const (Sum 1) <*> Const (Sum 2) == Const (Sum 3)
```

Monoid 与 Applicative

```
instance Monoid w => Applicative (w,) where
    -- pure :: x -> (w, x)
    pure _ = (w, x)
    -- (<*>) :: (w, b -> c) -> (w, b) -> (w, c)
    (w1, f) <*> (w2, x) =
        (w1 `mappend` w2, f x)

-- (Sum 1, ...) <*> (Sum 2, ...) == (Sum 3, ...)
```

Bicyclic semigroup

```
data Balance = Balance { close :: Int, open :: Int }
                                   deriving (Show, Eq)

-- "))(((",  Balance 2 3
-- "(", Balance 0 1
-- "))(((" ++ "(", Balance 2 4
-- "(" ++ "))(((", Balance 1 3

instance Semigroup Balance where
    Balance c1 o1 <> Balance c2 o2
        | o1 > c2 = Balance c1 (o1 - c2 + o2)
        | otherwise = Balance (c1 + c2 - o1) o2

instance Monoid Balance where mempty = Balance 0 0
```

Bicyclic semigroup

```
parseBalance :: Char -> Balance
parseBalance '(' = Balance 0 1
parseBalance ')' = Balance 1 0
parseBalance _   = Balance 0 0

balance :: String -> Bool
balance str =
    mconcat (map parseBalance str) == Balance 0 0
```