08-group.lean

```
import Mathlib
```

Groups and their morphisms

In this file, we illustrate how Mathlib develops the theory of everyday algebraic structures, starting from semigroups, monoids, groups, and their morphisms.

By "illustrate", we do not mean to reconstruct these structures from scratch, as we haven't yet covered how to define structures and type classes in Lean.

Instead, we accept the Mathlib definitions and axioms, but reprove some of their consequences manually, with mention of the relevant theorems in Mathlib.

Hopefully, this will help you focus on building up the theory mathematically while getting familiar with the Mathlib API. Be advised that you can always ctrl+click on any name to see its actual definition in Mathlib.

For a more complete treatment (especially on the philosophy behind API design), read MiL chapter 7 and 9.

1 Semigroups

1.1 Objects

A `Semigroup` is a type with an associative binary operation `*`.

```
section

#check Semigroup
variable (G : Type*) [Semigroup G] (a b c : G)
example : a * (b * c) = (a * b) * c := by rw [mul_assoc]
```

An `AddSemigroup` is exactly the same as `Semigroup`, only with additive `+` notation.

```
variable (A : Type*) [AddSemigroup A] (a b c : A)
example : a + (b + c) = (a + b) + c := by rw [add_assoc]
```

Note that using the notation of `+` does not necessarily mean that the operation is commutative. To this end, we have `CommSemigroup` and `AddCommSemigroup`.

```
#check CommSemigroup
#check mul_comm

#check AddCommSemigroup
#check add_comm

end
```

1.2 Morphisms

A `MulHom` is a morphism between two semigroups that preserves the multiplication. The notation for this is `G₁ →ₙ* G₂`.

It's a bundle of:

- a function `f : G₁ → G₂`
- a proof that `f` preserves multiplication.

Strictly speaking, this definition does not require the `*` operation to be associative on `G₁` or `G₂`.

```
section

#check MulHom
variable {G₁ G₂ G₃ : Type*} [Semigroup G₁] [Semigroup G₂] [Semigroup G₃]
         (f : G₁ →ₙ* G₂) (g : G₂ →ₙ* G₃) (a b : G₁)
example : f (a * b) = f a * f b := by rw [map_mul]
```

Additive version of `MulHom` is `AddHom` (`→ₙ+`).

```
#check AddHom
```

You might already notice that a `MulHom` can be used just like a function. This is because Mathlib has instantiated the `FunLike` type class to `MulHom`, which provides the function coercion.

```
#synth FunLike (MulHom G₁ G₂) G₁ G₂
```

Creating a new `MulHom` requires providing all the data needed.

```
#check MulHom.mk
example : ℕ →ₙ+ ℕ := ⟨(· * 2), by intros; ring⟩
```

Composition of `MulHom`s as functions preserves multiplication.

```
example : G₁ →ₙ* G₃ := ⟨g ∘ f, by intros; dsimp; rw [map_mul, map_mul]⟩
```

To avoid manually constructing `MulHom` every time when composing, We may use `MulHom.comp g f`. The dot convention `g.comp f` is used here for convenience.

```
example : (g.comp f) (a * b) = (g.comp f) a * (g.comp f) b := by simp
```

A `MulHom` is determined by the underlying function.

```
#check MulHom.ext
example (f₁ : G₁ →ₙ* G₂) (f₂ : G₁ →ₙ* G₂) (h : f₁.toFun = f₂.toFun) : f₁ = f₂ := by
  ext x
  change f₁.toFun x = f₂.toFun x
  rw [h]
```

Above shows the bundled definition of `MulHom`, how to create it, and how to compose them. The same philosophy is adopted for other morphism-like structures in Mathlib, such as `MonoidHom`.

```
end
```

2 Monoids

2.1 Objects

A `Monoid` is a `Semigroup` with an identity element `1` s.t. `a * 1 = a` and `1 * a = a`.

```
section

#check Monoid
variable (G : Type*) [Monoid G] (a b c : G)
example : a * 1 = a := by rw [mul_one]
example : 1 * a = a := by rw [one_mul]
```

[EXR] characterization of the identity element

```
example (h : ∀ x : G, x * a = x) : a = 1 := by
  specialize h 1
  rw [one_mul] at h
  exact h
```

`Monoid` additionaly enables power notation `a ^ n` for natural number `n`.

```
#check Monoid.npow
example : a ^ 0 = 1 := by rw [pow_zero]
example (n : ℕ) : a ^ (n + 1) = a ^ n * a := by rw [pow_succ]
```

We are not prepared to prove this until we talk about induction.

```
#check one_pow
```

`AddMonoid` is the additive version of `Monoid`.

```
variable (A : Type*) [AddMonoid A] (a b c : A)
example : a + 0 = a := by rw [add_zero]
example : 0 + a = a := by rw [zero_add]
example : 0 • a = 0 := by rw [zero_smul]
example (n : ℕ) : (n + 1) • a = n • a + a := by rw [succ_nsmul]
```

For commutative monoids, we have `CommMonoid` and `AddCommMonoid`.

```
#check CommMonoid
#check AddCommMonoid

end
```

2.2 Morphisms

A `MonoidHom` is a morphism between two monoids that preserves the multiplication and the identity. The notation for this is `G →* H`.

```
section

#check MonoidHom
variable {G₁ G₂ G₃ : Type*} [Monoid G₁] [Monoid G₂] [Monoid G₃]
         (f : G₁ →* G₂) (g : G₂ →* G₃) (a b : G₁)
example : f (a * b) = f a * f b := by rw [map_mul]
example : f 1 = 1 := by rw [map_one]
```

Additive version of `MonoidHom` is `AddMonoidHom` (`→+`).

```
#check AddMonoidHom
```

`MonoidHom` need additional data to `MulHom`: preservation of `1`.

```
#check MonoidHom.mk
example : ℕ →+ ℕ := ⟨⟨( · * 2), by simp⟩, by intros; ring⟩

end
```

3 Groups

3.1 Objects

In Mathlib, a `Group` is defined to be a `Monoid` where every element `a` has an left inverse `a⁻¹` s.t. `a⁻¹ * a = 1`.

```
section

#check Group
variable (G : Type*) [Group G] (a b c : G)
#check a⁻¹
example : a⁻¹ * a = 1 := by rw [inv_mul_cancel]
```

The following exercises lead to a proof of: In a group, a left inverse is also a right inverse. This recovers the usual definition of a group.

[EXR] left multiplication is injective

```
example (h : a * b = a * c) : b = c := by
  apply_fun (a⁻¹ * · ) at h
  rw [← mul_assoc, ← mul_assoc, inv_mul_cancel, one_mul, one_mul] at h
  exact h
#check mul_left_cancel -- this has a name
```

[EXR] a left inverse actually also a right inverse

```
example : a * a⁻¹ = 1 := by
  apply_fun (a⁻¹ * · )
  · dsimp
    rw [← mul_assoc, inv_mul_cancel, one_mul, mul_one]
  · apply mul_left_cancel
#check mul_inv_cancel -- this has a name
```

The following proves that `G` is a `DivisionMonoid`. You don't need to know what this means for now.

```
#synth DivisionMonoid G
```

[EXR] characterization of a right inverse

```
example (h : a * b = 1) : b = a⁻¹ := by
  -- if you does not want to use `apply_fun`
  rw [← one_mul b, ← inv_mul_cancel a, mul_assoc, h, mul_one]
#check eq_inv_of_mul_eq_one_right -- this has a name
```

[EXR] characterization of a left inverse

```
example (h : a * b = 1) : a = b⁻¹ := by
  apply_fun ( · * b⁻¹) at h
  rw [mul_assoc, mul_inv_cancel, mul_one, one_mul] at h
  exact h
#check eq_inv_of_mul_eq_one_left -- this has a name
```

[EXR] involutivity of the inverse

```
example : (a⁻¹)⁻¹ = a := by
  symm; apply eq_inv_of_mul_eq_one_right
  exact inv_mul_cancel a
#check inv_inv -- this has a name
```

[EXR] inverse of a product

```
example : (a * b)⁻¹ = b⁻¹ * a⁻¹ := by
  apply inv_eq_of_mul_eq_one_left
  rw [← mul_assoc, mul_assoc b⁻¹, inv_mul_cancel, mul_one, inv_mul_cancel]
#check mul_inv_rev -- this has a name
```

some other injectivity
[EXR] inverse is injective

```
example (h : a⁻¹ = b⁻¹) : a = b := by
  apply_fun ( · ⁻¹) at h
  rw [inv_inv a, inv_inv b] at h
  exact h
#check inv_injective -- this has a name
```

[EXR] right multiplication is injective

```
example (h : b * a = c * a) : b = c := by
  apply_fun ( · * a⁻¹) at h
  rw [mul_assoc, mul_assoc, mul_inv_cancel, mul_one, mul_one] at h
  exact h
#check mul_right_cancel -- this has a name
```

wheelchair tactic for groups

```
#help tactic group
example : (a ^ 3 * b⁻¹)⁻¹ = b * a⁻¹ * (a ^ 2)⁻¹  := by
  group
```

[TODO] `DivInvMonoid` enables `zpow` notation for integer powers `a ^ n` where `n : ℤ`.

[IGNORE] It extends `npow` for monoids. See the library note [forgetful inheritance] for the philosophy of this definition.

Additive and commutative versions of groups are as usual.

```
#check AddGroup
#check CommGroup
#check AddCommGroup

end
```

3.2 Morphisms

`MonoidHom` It is also used for group homomorphisms.

```
section

variable {G₁ G₂ G₃ : Type*} [Group G₁] [Group G₂] [Group G₃]
         (f : G₁ →* G₂) (g : G₂ →* G₃) (a b : G₁)
```

[EXR] Monoid homomorphisms preserve inverses

```
example : f (a⁻¹) = (f a)⁻¹ := by
  apply eq_inv_of_mul_eq_one_right
  rw [← map_mul, mul_inv_cancel, map_one]
#check map_inv -- this has a name
```

[EXR] `MonoidHom` requires one to show preservation of `1`. But this is redundant for group homomorphisms.

```
example (φ : G₁ →ₙ* G₂) : φ 1 = 1 := by
  haveI : φ 1 * φ 1 = φ 1 * 1 := by rw [← map_mul, mul_one, mul_one]
  exact mul_left_cancel this
```

Hence in the case of groups, Mathlib provides a constructor `MonoidHom.mk'` that only requires the preservation of multiplication to build a `MonoidHom`.

```
#check MonoidHom.mk'

end
```