Groups (and Monoids)

The prototypical monoid is `ℕ`: it has an operation, a neutral element `0`, but no opposite.

In mathematics, we do not use monoids too much: they are somewhat *weak* algebraic structures. They are nonetheless crucial for our discussion about formalising algebraic structures for (at least) three reasons:

1. Groups are a generalisation of monoids: if we can prove some properties only relying on the theory of monoids, it makes less lines to code.
2. Rings are endowed with *two* operations but are a group only for one of them; for the other they (typically) are a monoid.
3. They are the "simplest" non-trivial example of an algebraic structure: so we can use it as a playground to understand how to reason about structures without getting lost in details.

Monoids

- In usual pen-and-paper mathematics, a monoid is a set `M` endowed with an **associative** operation `* : M × M → M` and a unit `1 ∈ M`, satisfying `∀ x, 1 * x = x * 1 = x`.

The way monoids are implemented in Mathlib is hierarchical: one defines sets with an operation, then generalises them to sets with an *associative* operations, then constructs sets with a particular element `1`, then generalises both notions together by requiring a compatibility between `1` and `*`, etc...

+++ Coming down to earth
A monoid comes with five main fields, gathered into a "structure"

```
structure (M : Type*) Monoid where
| mul : M → M → M                     -- denoted *
| one : M                             -- denoted 1
| mul_assoc (a b c : M) : a * b * c = a * (b * c)
| one_mul (a : M) : 1 * a = a
| mul_one (a : M) : 1 * 1 = a
```

We've already encountered structures: we saw that an equivalence is a pair of implications, and that the type `↑S` associated to a set `S : Set α` is a collection of pairs of the form `⟨x, hx⟩` where `x : α` and `hx : S x` is a proof.

Here,

- a *monoid structure* on `M` is a collection `⟨*, 1, mul_assoc, one_mul, mul_one⟩`
- a term of a monoid is just a term of it! The monoid is a type, so it comes with its terms even if it has more structure.
  `⌘`

In the last example we've seen that we were able to endow `Bool` with a monoid structure. But of course many types already come with a fixed, or canonical, monoid structure. To see whether this is true for a type `α` you can type

```
#synth Monoid α
```

to obtain the name of the declaration, if it exists, and an error otherwise.

`⌘`

+++

+++ Additive and commutative monoids
In principle, the symbol used to denote the operation `M → M → M` should play no role.

But on rings we certainly want to have two operations **with different symbols**.

Also, the names of properties of `* : M → M → M` ought to be `mul`-related, whereas the names of those of `+ : M → M → M` should probably have an `add` floating around. Likewise, the neutral element must be `1` or `0` according at what symbol we're using.

- An `AddMonoid` is like a monoid, but where `*` is written `+` and `1` is written `0`; and the `@toadditive` tag automatically creates the relevant translation.

  It has an extra-field `nsmul` of type `ℕ → M → M` that defines the multiplication by a natural number `n`, by default equal to `n • x = x + x ... + x` (`n` times). Type `•` as `\smul`.

- A `CommMonoid` is a special case of a monoid, but with an extra-field

  ```
  structure (M : Type*) CommMonoid M extends Monoid M where
  | mul_comm (a b : M) : a * b = b * a
  ```

And then one can go on to define `AddCommMonoid M` to be what you expect.

`⌘`

+++

+++ Monoid homomorphisms
A monoid homomorphism is a function `f : M → N` that respects the operation. There could be (at least) two ways to define this:

1. we could declare the property `MonHom : (M → N) → Prop` as

   def MonHom : (M → N) → Prop := f ↦ (∀ a b, f (a * b) = (f a) * (f b)) ∧ (f 1 = 1)

and let `MonoidHom` be the subset (or the subtype)

```
MonoidHom = {f : M → N | MonHom f} (or {f : M → N | MonHom f})
```

This would mean that a monoid homomorphism is a pair `⟨f, hf : MonHom f⟩`.

2. we could define a new type `MonoidHom M N`, as a structure

```
structure MonoidHom M N where
| toFun : M → N
| map_mul : ∀ a b, toFun (a * b) = (toFun a) * (toFun b)
| map_one : toFun 1 = 1
```

so that terms of `MonoidHom M N` would be *triples* ⟨f, map_mul f, map_one f⟩`.

These approaches are not *very* different, the problem with the first is that to access the proofs one has to destructure `hf` to `hf.1` and `hf.2`. Imagine if there were 20 properties...

- **Take-home message**: homomorphisms between algebraic structures are structures on their own, "bundling" together the underline function and all its properties.

It will be another story for continuous/differentiable/smooth functions...

+++

Groups

A group is a monoid `G` with inverses:

```
inv : G → G
inv_mul_cancel (g : G) : g⁻¹ * g = 1
```

Of course, there are also the notion of `AddGroup` with

```
neg : A → A
neg_add_cancel (a : A) : -a + a = 0
```

and notions of `CommGroup` and `AddCommGroup`, that add a commutativity constraint on `*` or on `+`, respectively, in order to define
commutative (or *abelian*) groups.

- There is a `group` tactic that proves identities that holds in any group (equivalently, it proves those identities that hold in free groups). The equivalent version for *commutative* groups is `abel`.

Concerning group homomorphisms, they are just **monoid** homomorphisms, so they are a structure with simply three structures: the function itself, and two proofs that it preserves multiplication and sends `1` to `1`.

`⌘`

Of course, there is also the notion a group *isomorphism*: this is a structure with four fields

```
structure GroupEquiv (G H : Type*) [Group G] [Group H] where
| toFun : G →* H
```

```
| invFun : H →* G
| left_inv : invFun ∘ toFun = id
| right_inv : toFun ∘ invFun = id
```

+++ How do you *state* that something is a group isomorphism?

- To *state* something means creating a type in `Prop`
- To prove a statement means creating an *inhabited* type in `Prop`
- A `GroupEquiv` is not a type in `Prop`, it has way too many terms...

  ```
  def IsoOfBijective (G H : Type*) [Group G] [Group H] (f : G →* H)
      (h_surj : Surjective f) (h_inj f) : G ≃* H := by
  ```

+++

⌘

Subgroups

A subgroup is *not defined* as a group that is also a subset. It is a subset closed under multiplication:

```
structure Subgroup (G : Type*) [Group G] where
| carrier : Set G
| mul_mem (x y : G) : x ∈ carrier → y ∈ carrier → x * y ∈ carrier
| inv_mem (x : G) : x ∈ carrier → x⁻¹ ∈ carrier
```

- This creates a new type `Subgroup G` whose terms are **the subgroups of G**. As for group isomorphisms above, this is not the proposition "`H` is a subgroup". You should expect to encounter expressions like

  H : Subgroup G

to declare that `H` is a subgroup of `G` (technically: a term of the type parametrising such subgroup structures).

+++ How can we prove that something *is* a subgroup?
Again, by *defining* a term!

Trivial ones

Among all subgroups of a group `G`, two fundamental examples are the trivial group `{1} ⊆ G` and the whole group `G ⊆ G`. To treat these things, we borrow the language of orders.

Indeed, subgroups are *ordered* (by inclusion of their carrier), so `{1} = ⊥` (the bottom element, typed `\bot` and `G = ⊤`, the top element (typed with `\top`).

+++

Rings

As for groups, the way to say that `R` is a ring is to type

```
(R : Type*) [Ring R]
```

The library is particularly rich insofar as *commutative* rings are concerned, and we're going to stick to those in our course. The tactic `ring` solves claim about basic relations in commutative rings.

Given what we know about groups and monoids, we can expect a commutative ring to have several "weaker" structures: typically these can be accessed through a `.toWeakStructure` projection.

`⌘`

+++ Morphisms and Ideals

- Morphisms work as for groups: they are simply functions respecting both structures on a ring, that of a multiplicative monoid and of an additive group: so, they're simply respecting both monoid structures, hence the notation `R →+* S` for a ring homomorphism. Of course, `≃+*` denotes ring isomorphism, so `R ≃+* S` is the **type** of all ring homomorphisms from `R` to `S`.

- Ideals

They're defined building upon the overarching structure of `Module`s, but it won't matter for us. Suffices it to say that in the following setting

```
example (R : Type*) [CommRing R] (I : Ideal R)
```

the type `Ideal R` consists of all ideals `I ⊆ R`; hence, a term `I : Ideal R` is such that

```
| I.carrier : Set R
| I.zero_mem : (0 : R) ∈ I
| I.add_mem (x y : R) : x ∈ I → y ∈ I → x + y ∈ I
| I.smul_mem (x y : R) : x ∈ I → x • y ∈ I
```

The `smul_mem` field is part of the definition, but it is sometimes handier to use either of

```
I.mul_mem_left (a b : R) : b ∈ I → a * b ∈ I
```

or

```
I.mul_mem_right (a b : R) : a ∈ I → a * b ∈ I
```

As for subgroups, the type `Ideal R` is ordered, and the ideal `{0} : Ideal R` is actually `⊥` whereas `R : Ideal R` is `⊤`.

⌘