Groupify: Language Specification

Sarah Fida and Jon Carl

Fall 2022

1 Introduction

The mathematical definition of a group is as follows: a set of elements, combined with a binary operation form a *group* if (1) the group is closed under the binary operation, (2) there exists an identity element, (3) every element has an inverse, and (4) the operation is associative.

This mathematical definition seems a bit daunting. Instead, let's think about an example. In this working example, we'll work with colors under composition. That is, imagine we have a palette of colors, say {red, green, blue} to start. Note that if we combined red and green, the output is yellow. But yellow is not contained within the original set, so we say that {red, green, blue} is not a group under composition since it is not **closed**.

Now, instead of {red, green, blue}, let's expand our set to be the set of all colors in the universe. Then our set is **closed** under composition since any combination of colors will always yield a color contained in the set. We also know that our set contains an identity color, namely clear (transparent). Given any color, when we mix clear with that color, we get the same color as the output.

You may have to think a little harder about inverses, but it turns out that every color has an inverse. That is, every color has some pairing such that when you mix the color with its partner, you get clear out. Lastly, we know that associativity holds under composition. It doesn't matter if we mix red in after we mix green and blue or if we mix blue in after we mix red and green. We still get the same color out.

Since the set of all colors under composition is (1) closed, (2) has an identity, (3) has inverses, and (4) associativity holds, we say that the set of colors combined with composition form a group. Let's call this group the *Color Group*. *Group Theory* is a fundamental area of research and has many applications beyond the provided example including mathematics, physics, chemistry, and computer science. In fact, *Group Theory* plays a pivotal role in the mathematics behind cryptography.

2 Design Principles

When defining the design principles that dictate the development of a programming language, we first note the importance of an intuitive user interface. Our programming language aims to simplify the classification of groups, and so executing this act via a single user input is what displays the effectiveness of making this task computational. Additionally, the input being representative of the problem at hand (i.e visually represented as a potential group) is what makes this language easily understandable, despite its technical complexity.

The technical principles which underpin the design are separated into parsing and evaluation. In order to evaluate the input, we need to store it in a useful form first that allows us to evaluate the input in a meaningful way. Structurally, this is handled in `Parser.fs`. Once we transform our data into an AST, we then handle our evaluation in `Evaluation.fs` in four steps. These steps correspond to the verifications required to determine if a set and operation form a group: (1) closure, (2) identity, (3) inverses, and (4) associativity. We print the output in `Program.fs`.

3 Examples

Note that each example shown below can be tested directly in the Groupify repository. To run each test, use

```
dotnet run ../GroupifyTest/examples/example-<example number>.groupify
```

at the command line.

1.

```
dotnet run "{0,1,2,3,4} +%5" --verbose
"
Numbers [Num 0.0; Num 1.0; Num 2.0; Num 3.0; Num 4.0] is a group under +%5 because

It is closed under +%5.

The identity element is 0.

Every element has an inverse: [(2, 3); (1, 4); (0, 0)].

+%5 is associative.

"
```

2.

```
dotnet run "Z +" --verbose
"
"Z" is a group under + because:

It is closed under +.

The identity element is 0.

Every element has an inverse.

+ is associative.

"
```

3.

```
dotnet run "{-1,1} *"
true
```

4.

```
dotnet run "{1,2,3} /" --verbose
"
Numbers [Num 1.0; Num 2.0; Num 3.0] is not a group under / because:

It is not closed. Notice that 1,2 are in Numbers [Num 1.0; Num 2.0; Num 3.0],
but 1 / 2 = 0.5 is not in Numbers [Num 1.0; Num 2.0; Num 3.0].

It contains no identity element.

1 is an element with no inverse.

It is not associative."
```

5.

```
dotnet run "{1,2,3,4} -" --verbose
"
Numbers [Num 1.0; Num 2.0; Num 3.0; Num 4.0] is not a group under - because:

It is not closed. Notice that 1,1 are in Numbers [Num 1.0; Num 2.0; Num 3.0;
Num 4.0], but 1 - 1 = 0 is not in Numbers [Num 1.0; Num 2.0; Num 3.0; Num 4.0].

It contains no identity element.

4 is an element with no inverse.

It is not associative."
```

6.

```
dotnet run "R⋆ +"
false
```

4 Language Concepts

The user needs to understand how to convert the input into a useful form. On the command line, the user provides (1) a set and (2) an operation. These are the big components of the programming language. At its core, though, a set is simply a collection of elements. Thus, the user needs to understand that one primitive is an element (i.e a number). Another primitive is the binary operation itself. These are the two primitives of the language. A set is just a list of one or more elements. After converting the input into an AST consisting of elements and an operation, we can evaluate the AST in an easier manner.

Evaluation consists of verifying the four tenets of a group: (1) closure, (2) identity, (3) inverses, and (4) associativity. If any of these verifications fail, then we conclude that the input does not form a valid group. When all four verifications pass, we say that the provided set and operation form a group.

5 Formal Syntax

```
<expr>      :==
            | <num>
            | <element>
            | <operation>
            | <group>

<num>       :== $\alpha \in \mathbb{Z}$

<element>   :== $\beta$

<operation> :==
            | +%$n$ for some $n \in \mathbb{N}$
            | +
            | -
            | ·
            | \

<group>     :== <operation><ExprSet>
```

```
<ExprSet>   :==
            | <Integers>
            | <Rationals>
            | <Reals>
            | <Complex>
            | <Numbers>
            | <Elements>

<Integers>  :== $\mathbb{Z}$
<Rationals> :== $\mathbb{Q}$
<Reals>     :== $\mathbb{R}$
<Complex>   :== $\mathbb{C}$
<Numbers>   :== $\{a, b, \cdots z\} \mid a, \ldots, z \in \mathbb{Z}$
<Elements>  :== $\alpha \in G$
```

6 Semantics

Syntax	Abstract Syntax	Type	Prec./Assoc.	Meaning
`num`	`num of int`	`int`	n/a	`num` is a primitive. We represent `num` using 32 bit F# ints (`Int32`).
`element`	`element of int*int* string`	`int*int* string`	n/a	`element` is a primitive. We represent `element` using a F# 3 tuple `int*int*string`.
`operation`	`operation of string * string * int`	`string * string * int`	n/a	`operation` is a primitive. We represent `operations` using F# tuples (`string * string * int`).
`group`	`group of expr * expr list`	`expr * ExprSet`	n/a	`group` is a primitive. We represent `groups` using F# tuples (`string * string * int`).
`Integers`	`Integers of string`	`string`	n/a	The `Integers` are a primitive type of `ExprSet`. We represent the `Integers` using a F# `string`.
`Rationals`	`Rationals of string`	`string`	n/a	The `Rationals` are a primitive type of `ExprSet`. We represent the `Rationals` using a F# `string`.
`Reals`	`Reals of string`	`string`	n/a	The `Reals` are a primitive type of `ExprSet`. We represent the `Reals` using a F# `string`.

`Complex`	`Complex of string`	`string`	n/a	The `Complex` numbers are a primitive type of `ExprSet`. We represent the `Complex` numbers using a F# `string`.
`Numbers`	`Numbers of Expr list`	`Expr list`	n/a	`Numbers` are a primitive type of `ExprSet`. We represent `Numbers` using a F# `Expr list`.
`Elements`	`Elements of Expr list`	`Expr list`	n/a	`Elements` are a primitive type of `ExprSet`. We represent `Elements` using a F# `Expr list`.

7 Remaining Work

Here is a brief checklist of the things we have to complete:

1. Add support for arbitrary elements (not just numbers). That is, we hope to provide support for the following example:

   ```
   dotnet run "{e, a, b}: e.e = e, e.a = a, e.b = b,
   a.e = a, b.e = b, a.b = e, b.a = e"
   true
   ```

   Note that since our machines don't explicitly know how to add e, a, b, we need to provide direct mappings for each pair of elements under the provided operation. We're thinking about better ways to ask for this user input than the example provided above. One idea: if user provides non-numerical input, provide a table for the user to complete that determines mappings. The table might look as follows:

	e	a	b
e	e	a	b
a	a	b	e
b	b	e	a

   Evaluating this table would be rather straightforward: in a sense, any set combined with a binary operation forms a group if this table is a *Latin square*. Evaluation would verify that the provided table is a Latin square.