Algebraic Type Design - A Step towards Formal Design Patterns in Functional Programming

Praseed Pai K.T.

About the Presenter

- A Seasoned Software Engineering Professional with more than twenty five years of Exposure
- Author of Two books on Computer Programming
- Explorer in “Philosophical Tools for Software Engineering” (Has Presented on it, Written one university accredited paper, Designed a Pattern based on Advaita Vedanta to transition from OOP to FRP)
- An Expert level professional in Cross Cultural Encounters (How to deal with a Russian/Eastern European? , Working with Racial stereotypes like Jews / Chinese / Latin Americans)
- A Critique of Digital Technology Fads (Programmers will be better off , if they stick to Programming. Do not run after so called AI/ML, BlockChain etc) - “Plumbing is preferred over Painting!”
- I also help Programmers eliminate their “Math-Phobia”

The Goal of this Presentation

- Modern Programs and Libraries can be written by Composing pure Functions (Functions with well designed properties) together by leveraging computational structures like Monads, Functors , Monoid etc
- Modern FP patterns help us to validate programs for correctness (at a formal level)
- Declarative Programming is almost here!

GOF Pattern Lacks Formal Verification

- GOF Pattern can help us to Compose Classes Together (Composite) , but since the State is mutable , can be problematic in muti core, parallel/concurrent and distributed environment
- A Well behaved Pattern should be mathematically verified

Two case Studies of Formal Verification

- The Verification of “COM Component”
- The Verification of a Custom Type which mimics the semantics of int, double , float

Enough of Mathematical Objects, What about the Real World?

- By carefully designing types and choosing operations correctly, we can verify any type
- From Category theory (Mathematics of Mathematics) people have borrowed concept of Monads to structure computations and sequence them in a linear manner
- At the bottom, Monad is a type with map and flatMap method implemented
- Let us see two Monads in Action (We will see the rationale of Monads later)

Hmm...there seems some substance beneath the hype of Monads

- Scala has constructs (for comprehension) to work with Monadic design
- I am still scared …. Why it works and How it Works?
- Let us start with Arithematic and I assure you that I require only high school mathematics to teach you Monads (Well almost!)

Let us Start to Count/Measure

- We will take defenition of number for granted at this point in Time
- What is Arithematic?
  - Arithematic is the process of composing (combining) numbers to generate new numbers
  - Eg:- 2 + 3 , 2 + 3*4
  - We use Symbols (+ | * | / | -) for denoting operators and the numbers we manipulate using operators are called opeands
  - We can chain operators to create Arithematic Expressions

Arithematic Expressions

- What is an Expression ?
- Expression is a chain of operations which are glued together
- An Expression consits of Terms , Factors and (Sub) Expressions!
- Terms are what you add and Factor is what you multiply
- Egs :- (2+3*4) = > 2 and 3*4 are terms
- 2 is a factor as it is 2*1
- 3 and 4 are factors as they are multiplied
- The Following Backus Naur Form can express an Expression
- <Expr> := <Term> (+ | *) <Expr>
- <Term> := <Factor> (* | /) <Term>
- <Factor> := + <Factor> | (<Expr>) | <Number> | -<Factor>

Infix/Prefix and Postfix notation

- Different Notations for Expressions
- Infix Notation (Mathematics and most Programming languages)
- Prefix Notation (LISP/Scheme uses it)
- PostFix Function (Stack based Evaluation , FORTH and Display PostScript)

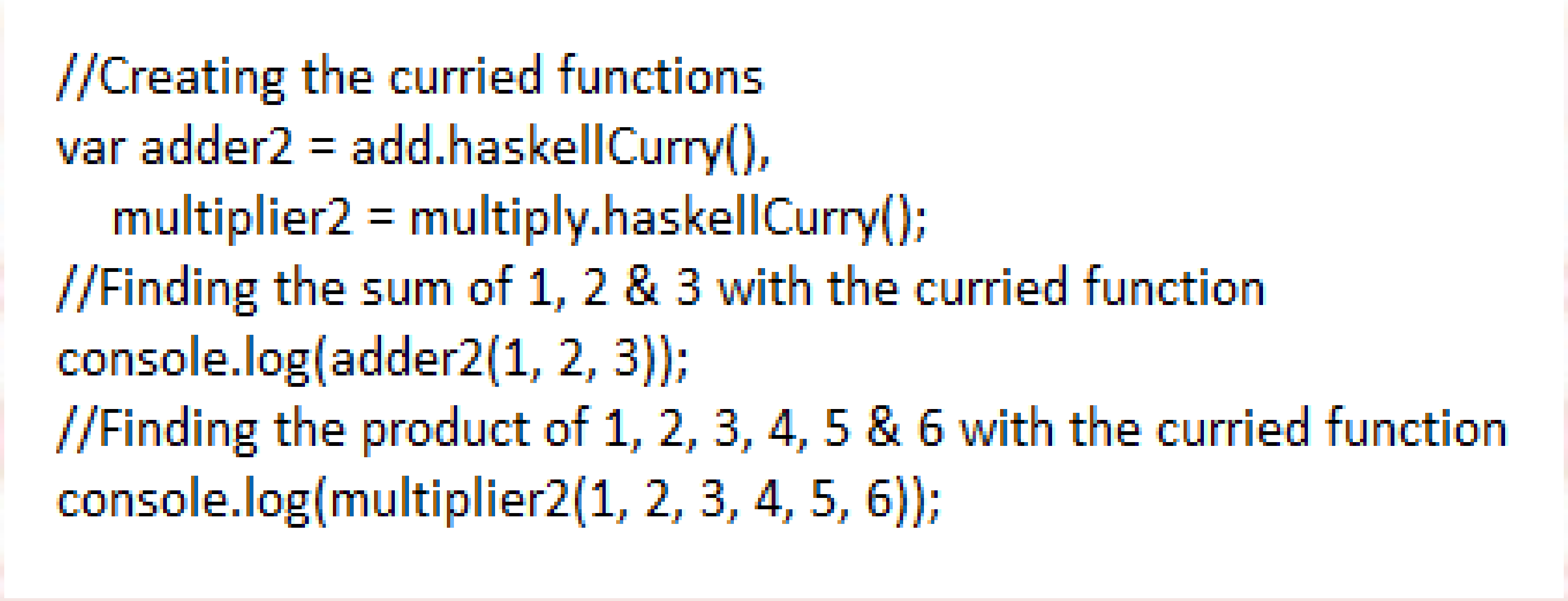
How Lisp/Scheme Works ?

- The Code is stored as a List
- The First Element of List is denoted by Car(Lst)
- The Rest of the List is denoted by Cdr(Lst)
- Eval(Lst) is the Evaluation Function
- Apply(fn)

```
◆   Eval(Lst) :=> if (C := Car(Lst))  { return Value(C) }
                  else if (V := Car(Lst))  { return Env.Lookup(V) }
                  else  { return Apply(Car(Lst),Eval(Cdr(Lst)))|
Apply(fn,Lst) :=> { return fn(Lst) }
```

Eval/Apply magic for Currying

- How to use Eval/Apply to support a “natural looking” Expression for Currying



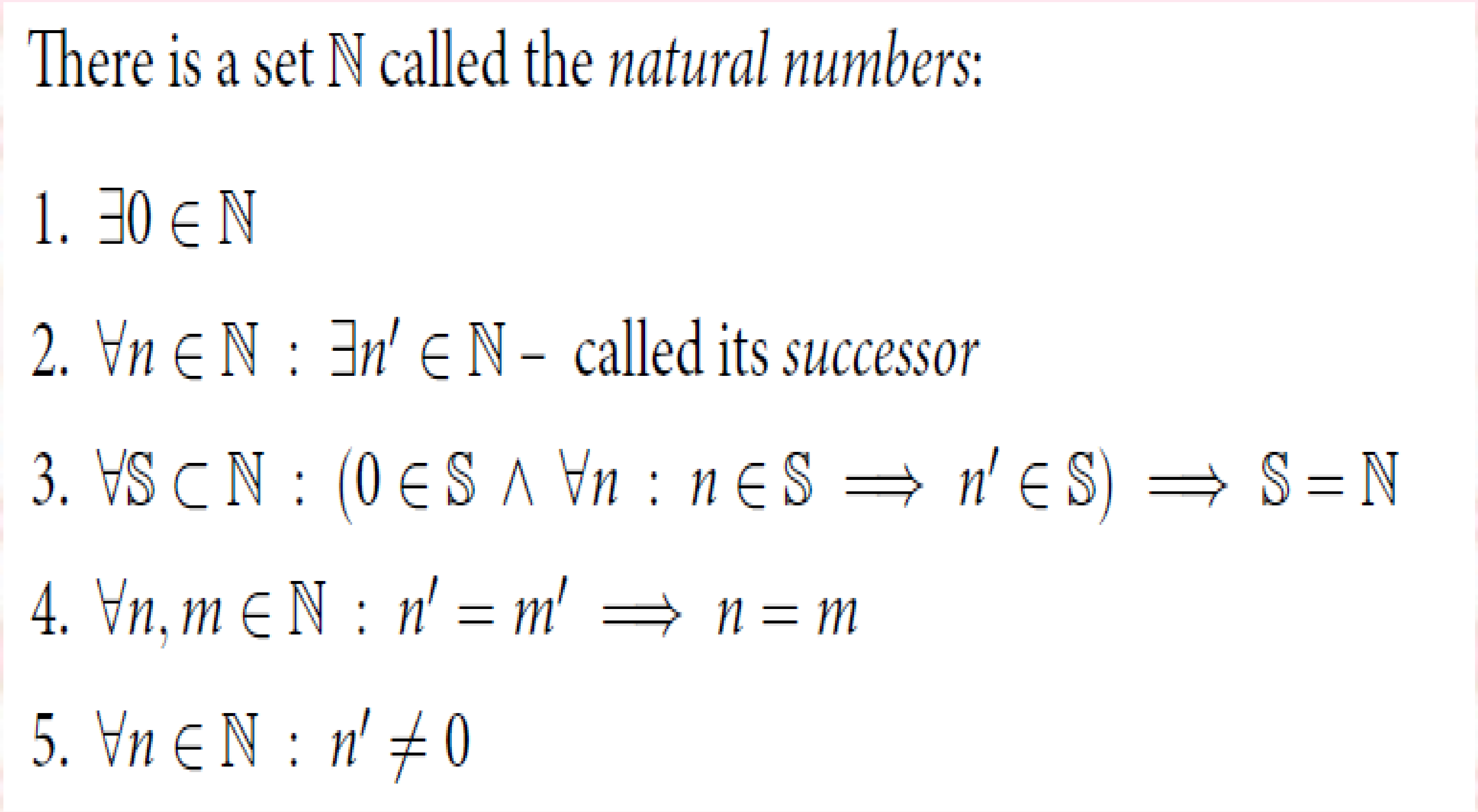
```
//Creating the curried functions
var adder2 = add.haskellCurry(),
    multiplier2 = multiply.haskellCurry();
//Finding the sum of 1, 2 & 3 with the curried function
console.log(adder2(1, 2, 3));
//Finding the product of 1, 2, 3, 4, 5 & 6 with the curried function
console.log(multiplier2(1, 2, 3, 4, 5, 6));
```

Operands can be any of the below in Arithematic

Adding More Power to Arithematic

- Natural Numbers (N)
- Whole Numbers (W)
- Integers (Z)
- Rationals (Q)
- Reals (R)
- Complex Numbers (C)
- Quaternions
- Octonions

Peano's Axiom for Natural Numbers



There is a set $\mathbb{N}$ called the *natural numbers*:

1. $\exists 0 \in \mathbb{N}$
2. $\forall n \in \mathbb{N} : \exists n' \in \mathbb{N}$ – called its *successor*
3. $\forall \mathbb{S} \subset \mathbb{N} : (0 \in \mathbb{S} \wedge \forall n : n \in \mathbb{S} \implies n' \in \mathbb{S}) \implies \mathbb{S} = \mathbb{N}$
4. $\forall n, m \in \mathbb{N} : n' = m' \implies n = m$
5. $\forall n \in \mathbb{N} : n' \neq 0$

Mixed Mode Expressions in Arithematic

- We can mix number types in an expressions
- The necessity of Casting (Promotion or Coercion)
- E:= C + R => C + (C)R (0i + R)

  R + N => R + (R)N

Properties of Arithematic Operators

A Good Operator(s) should have

- Associativity
- Commutativity
- Closure (Type of Result does matter)
- Distributivity (Two ops in an Expression)
- Inverse Element (Additive/Multiplicative)
- Identity Element (0 | 1 | “”)

Let us Move To Algebra

- Algebra is the Generalization of Arithematic
- Instead of directly manipulating number types, we use Symbols like x,y,z,t,a,b etc to represent Operands
- 2 + 3 will be come x + y (or a + b)
- X, y, z , t, a, b are called variables, which denotes values
- Every variable has got a TYPE and a Potential value set (PVS) for that TYPE

TYPE(s) in Algebra

- When we say We add x + y,
  x:int + y:int => z:int (overflow!)
- X can take value from a finite set (or a small subset of the possible values) like PVS = { 0,1,2,3 } and Y can take value from another finite set { 1,2 }
- When a value is attached to X or Y, we call it “Binding” (There is no concept of Assignment in Mathematics!)

Universal Quantifier and Bounded Variable

For each x, if x is not zero, then its square is positive.

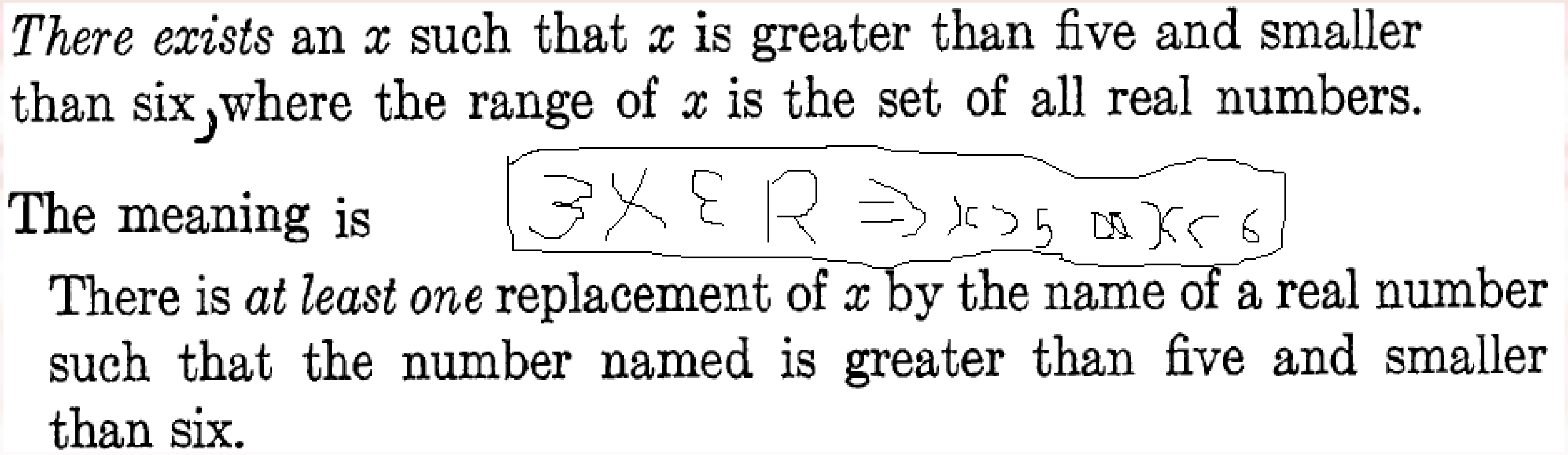
∀ x E N , X <> 0 => X2 > 0

The meaning is

For each replacement of x by the name of a real number, if the number named is not zero, then its square is positive.

```
def add ( a:Int , b:Int ) = a + b
```

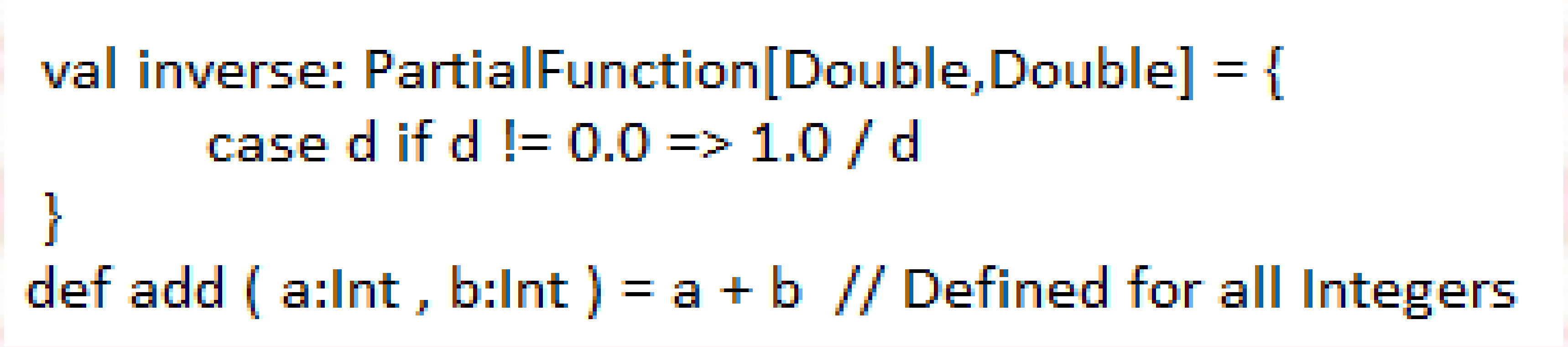
Existential Quantifier and Bound Variable



There exists an x such that x is greater than five and smaller than six, where the range of x is the set of all real numbers.

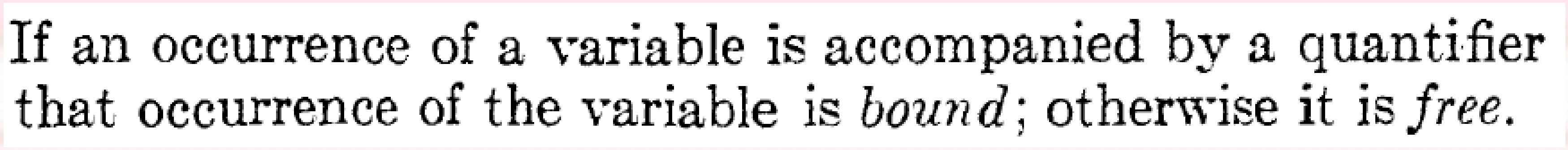
The meaning is $\boxed{\exists X \in R \Rightarrow x > 5 \text{ DA } x < 6}$

There is *at least one* replacement of x by the name of a real number such that the number named is greater than five and smaller than six.

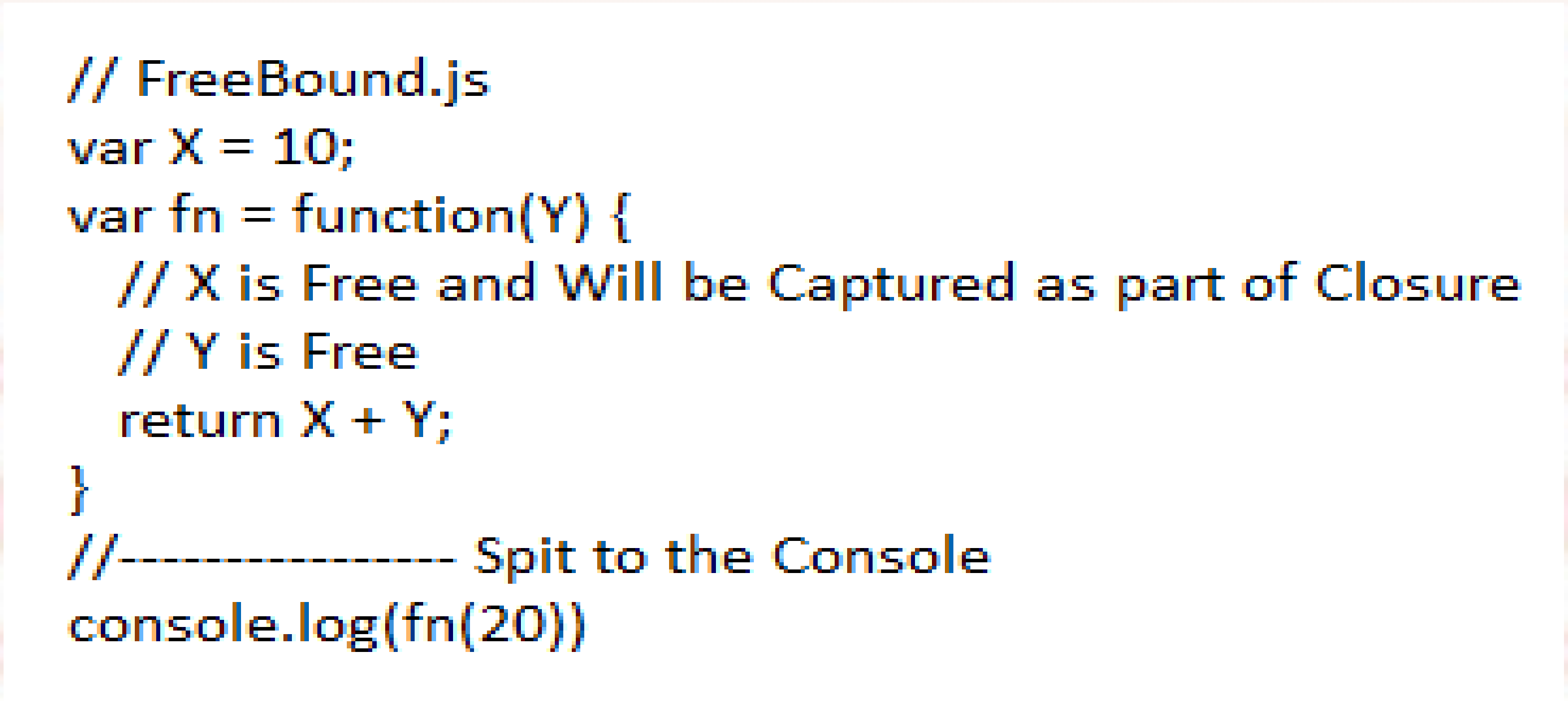


```
val inverse: PartialFunction[Double,Double] = {
        case d if d != 0.0 => 1.0 / d
}
def add ( a:Int , b:Int ) = a + b  // Defined for all Integers
```

Free Variables



If an occurrence of a variable is accompanied by a quantifier that occurrence of the variable is *bound*; otherwise it is *free*.



```
// FreeBound.js
var X = 10;
var fn = function(Y) {
    // X is Free and Will be Captured as part of Closure
    // Y is Free
    return X + Y;
}
//------------------ Spit to the Console
console.log(fn(20))
```

Associative Property

- ((A op B) op C) == (A op (B op C))
- 2 + 3 + 4 can be written as
  ((2 + 3) + 4) or (2 + (3 + 4))
- If a operator is associative, we can do parallel reduction (a long sequence of numbers can be chunked into small sequence to be reduced by different people, processors or mechanical devices)

Commutative Property

- (A op B)   == (B op A)
- 2 * 3 can be written as 3 * 2
- If a operator is commutative, order in which one performs operation does not matter
- We can do Out of Order Execution (Relational DB Engine exploits this property to evaluate relational cross products)
- On top of Parallel reduction, we can perform Parallel shuffle as well

Closure

- When two Homogeneous Types of numbers are Operated Upon, if we get the same Type as result, it is called “Closure”
- Addition of Two natural numbers are closed
- So do Multiplication of Two Natural Numbers
- Closure helps us to Chain Operations without much “trouble”

Closure in Regular Expressions

Re(NULL) => NULL
Re(“”) => “”
Re([a-z]) => [a-z]
Re.Re => Re
(Re | Re) => Re
Re* => Re
The above stuff defines Re (Recursive definition)
What about R+?
Re+ = Re.Re*

Closure in SQL

Data is stored in a data structure called Relation
Relations can be combined using Rel Ops

CartesianProduct(Rel1,Rel2..Reln) => Rel
Restrict(Rel,Predicate) => Rel
Project(Rel,fieldlist) => Rel
Rename(Rel)=> Rel
SetOperators(Rel1..Reln) => Rel
Group(Rel,Pred) => Rel
And so on…

Closure in FP

- Correct Functional Composition works because of the Closure of Operations
- A = F(G(H(X))) (F map G map H on X)
- The Closure of Operations are violated when we have non determinism , exceptions, nulls , I/O , Stateful operations
- We will encapsulate the above “mess” into to implementation and try to give hygenic behavior at the interface level (S*** will be in the Pit!)

Algebra To Modern Algebra

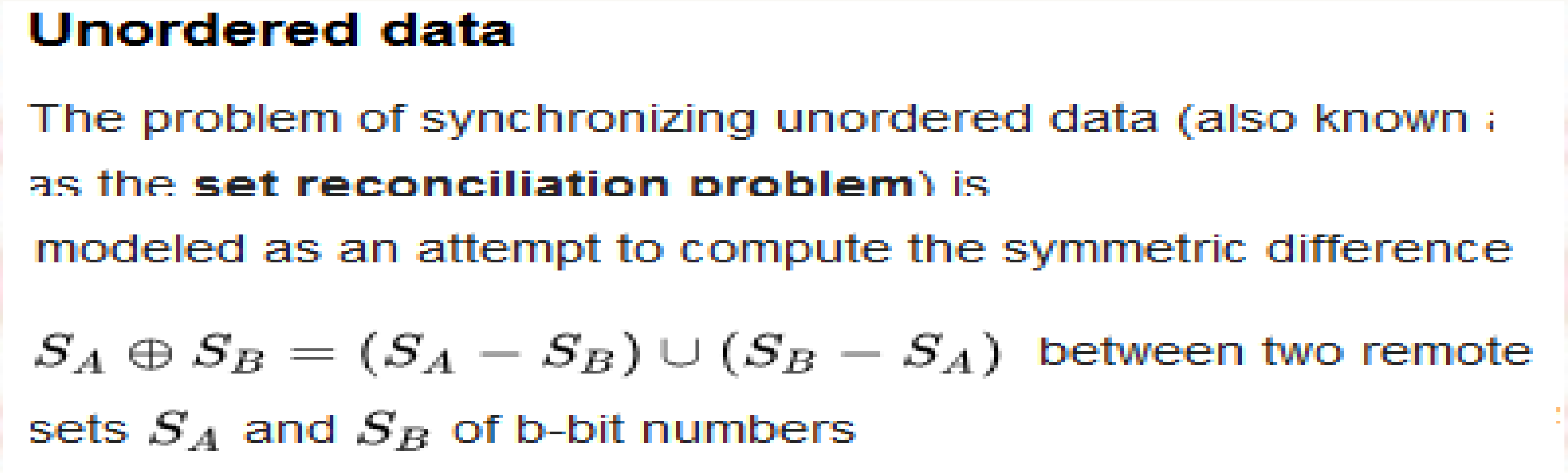
- Why Limit Algebra to Number Types?
- Algebra of Sets (Operations on Sets)
- Algebra of Relations
- Algebra of Matrices
- Algebra of Vectors
- Algebra on Groups, Rings, Fields, Lattices
- We Create Hierarchy of Mathematical Structures with Varying Property
- Mathematical Properties like Associativity,Commutativity, Closure nicely extrapolates to Modern Algebra

Sets and Computer Programming

- Sets, Operations on Sets, Collections etc
- MultiSet or Bag (Set with duplicates)
- Symmetric Set Differences and Data Synchronization

Sets and Computer Programming

- Sets, Operations on Sets, Collections etc
- MultiSet or Bag (Set with duplicates)
- Symmetric Set Differences and Data Synchronization



Unordered data

The problem of synchronizing unordered data (also known

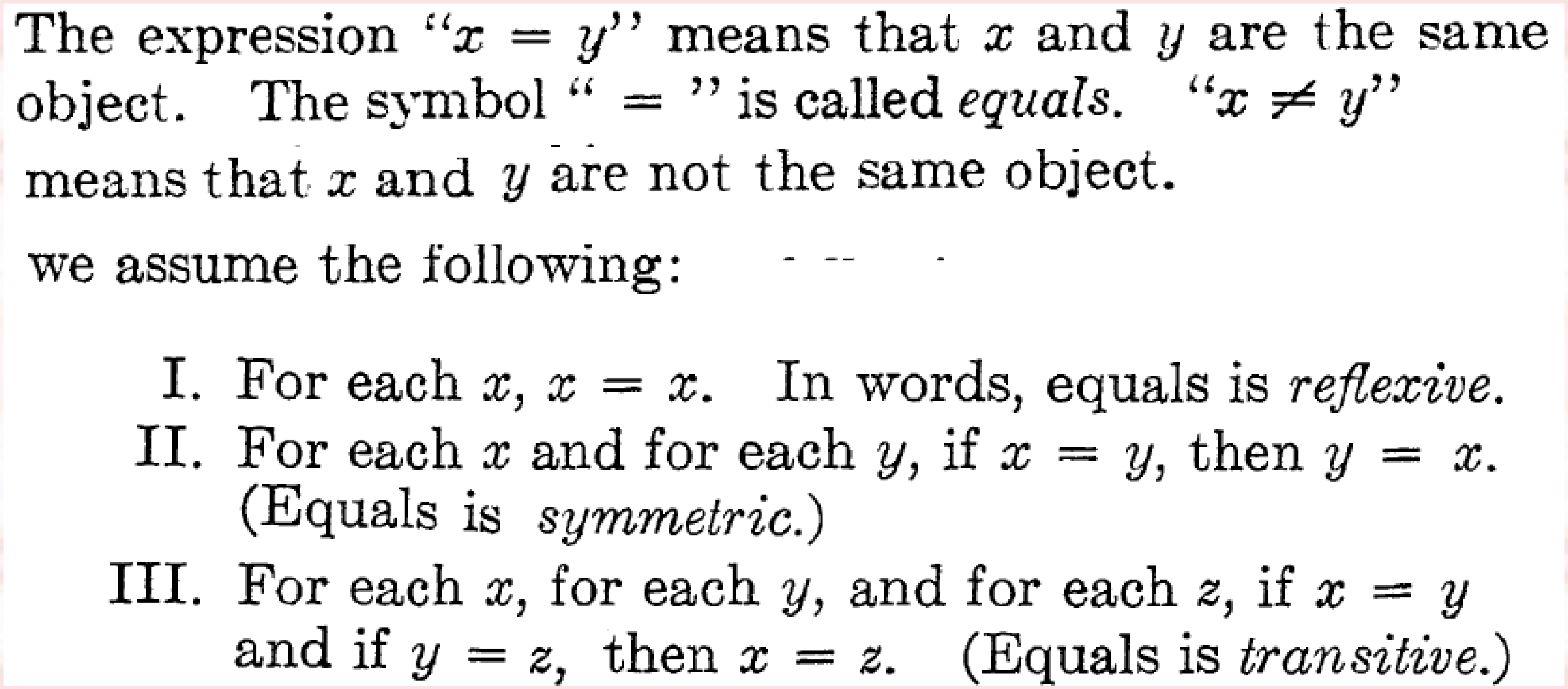
as the **set reconciliation problem**) is

modeled as an attempt to compute the symmetric difference

$S_A \oplus S_B = (S_A - S_B) \cup (S_B - S_A)$ between two remote

sets S_A and S_B of b-bit numbers

Some notions about Equality



The expression "$x = y$" means that x and y are the same object. The symbol " $=$ " is called *equals*. "$x \neq y$" means that x and y are not the same object.

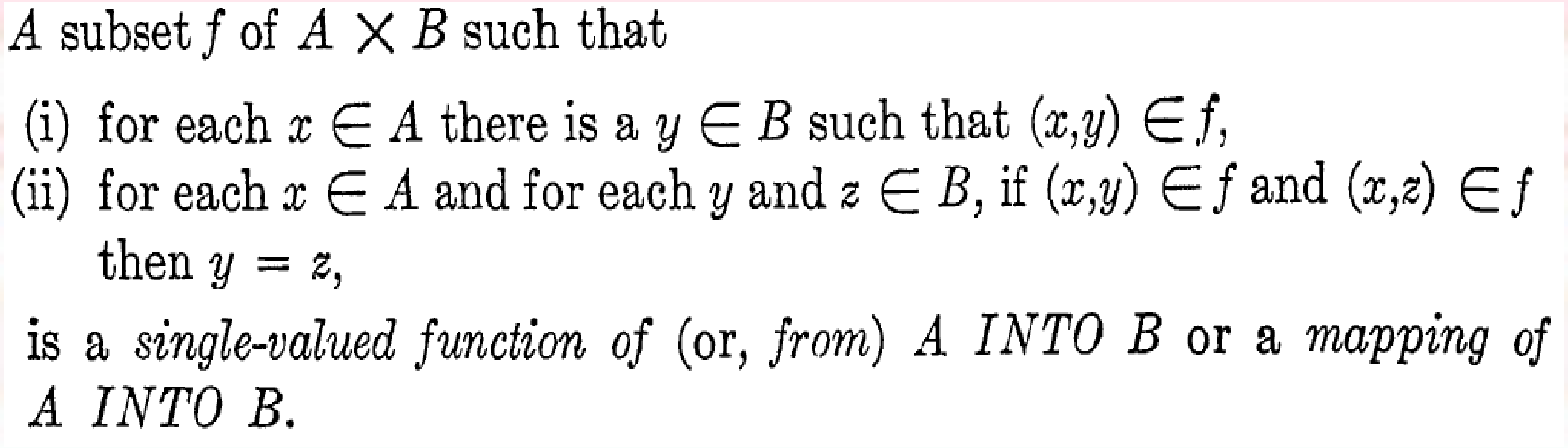
we assume the following:

I. For each x, $x = x$. In words, equals is *reflexive*.
II. For each x and for each y, if $x = y$, then $y = x$. (Equals is *symmetric*.)
III. For each x, for each y, and for each z, if $x = y$ and if $y = z$, then $x = z$. (Equals is *transitive*.)

Relations

- Relations are subsets of Cartesian Products between Sets/Bags
- We Apply a Predicate to find the Subset of Cartesian Product between two sets
- Mathematical Properties of Relations like Reflexivity/Symmetry/Transitivity
- Notion of Equality
- Algebra of Relations (which makes SQL engines tick)
- Using Properties of Relation to Veriify a COM Component.

A Formal Definition of Function

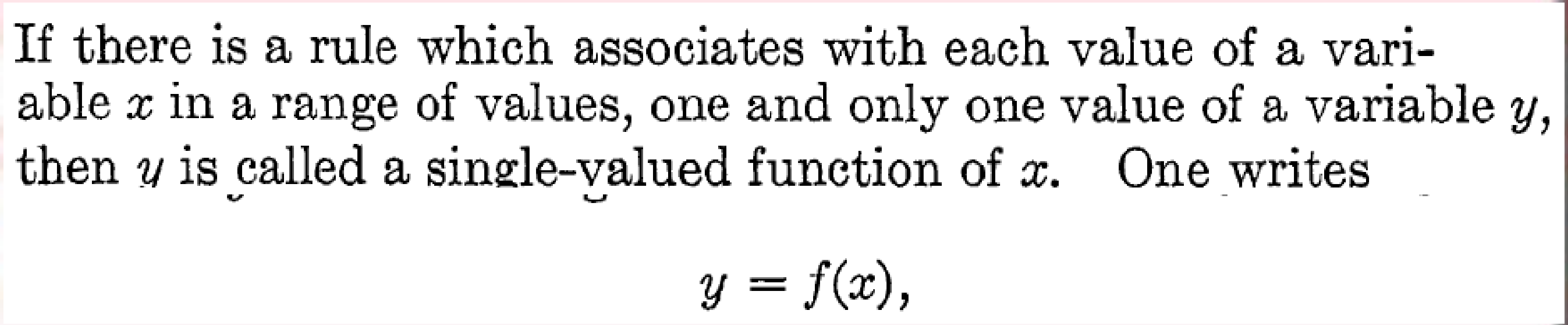


A subset f of $A \times B$ such that

(i) for each $x \in A$ there is a $y \in B$ such that $(x,y) \in f$,
(ii) for each $x \in A$ and for each y and $z \in B$, if $(x,y) \in f$ and $(x,z) \in f$ then $y = z$,

is a *single-valued function of* (or, *from*) A $INTO$ B or a *mapping of* A $INTO$ B.

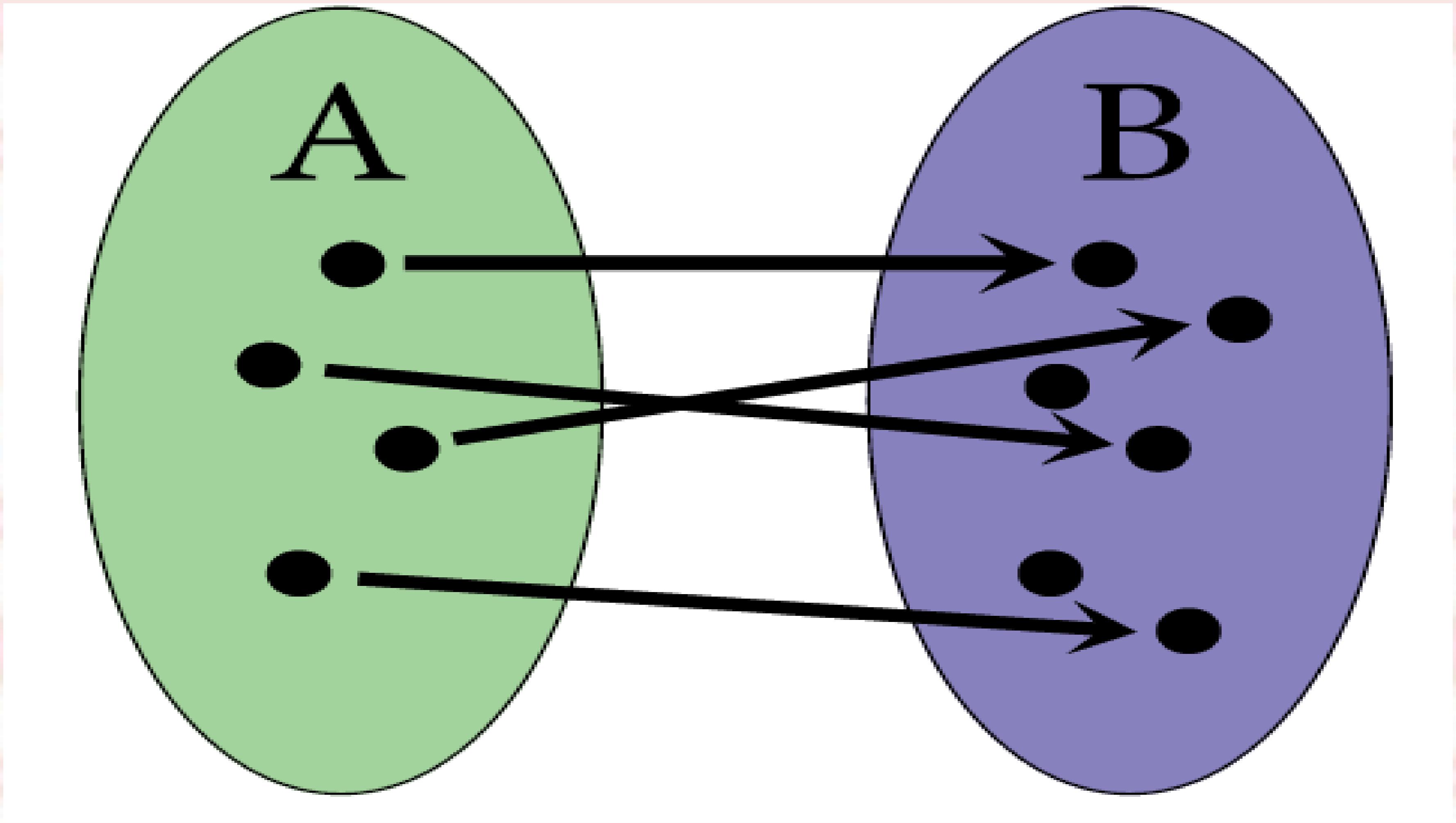
A Simple definition of Function



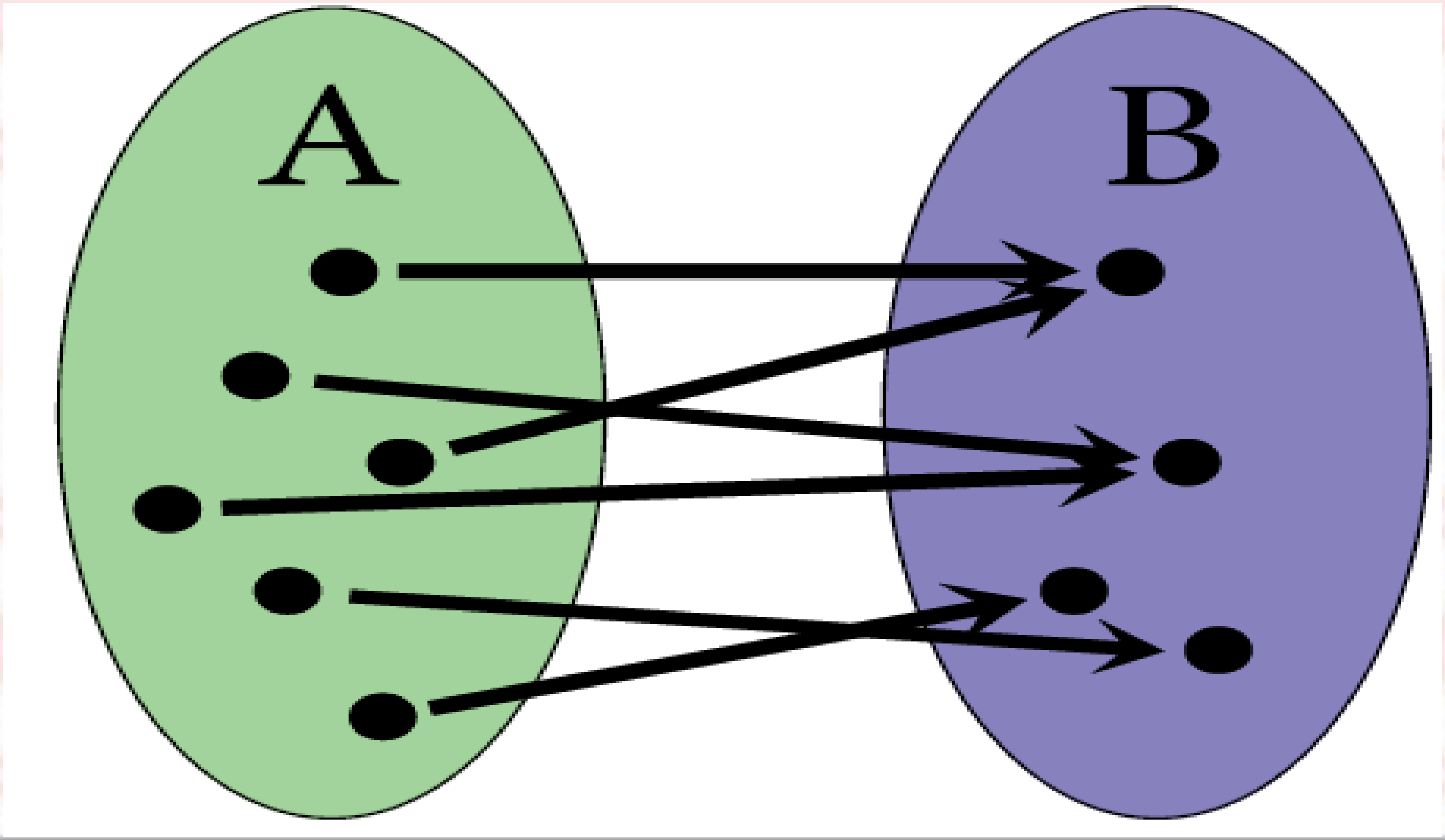
If there is a rule which associates with each value of a variable x in a range of values, one and only one value of a variable y, then y is called a single-valued function of x. One writes

$$y = f(x),$$

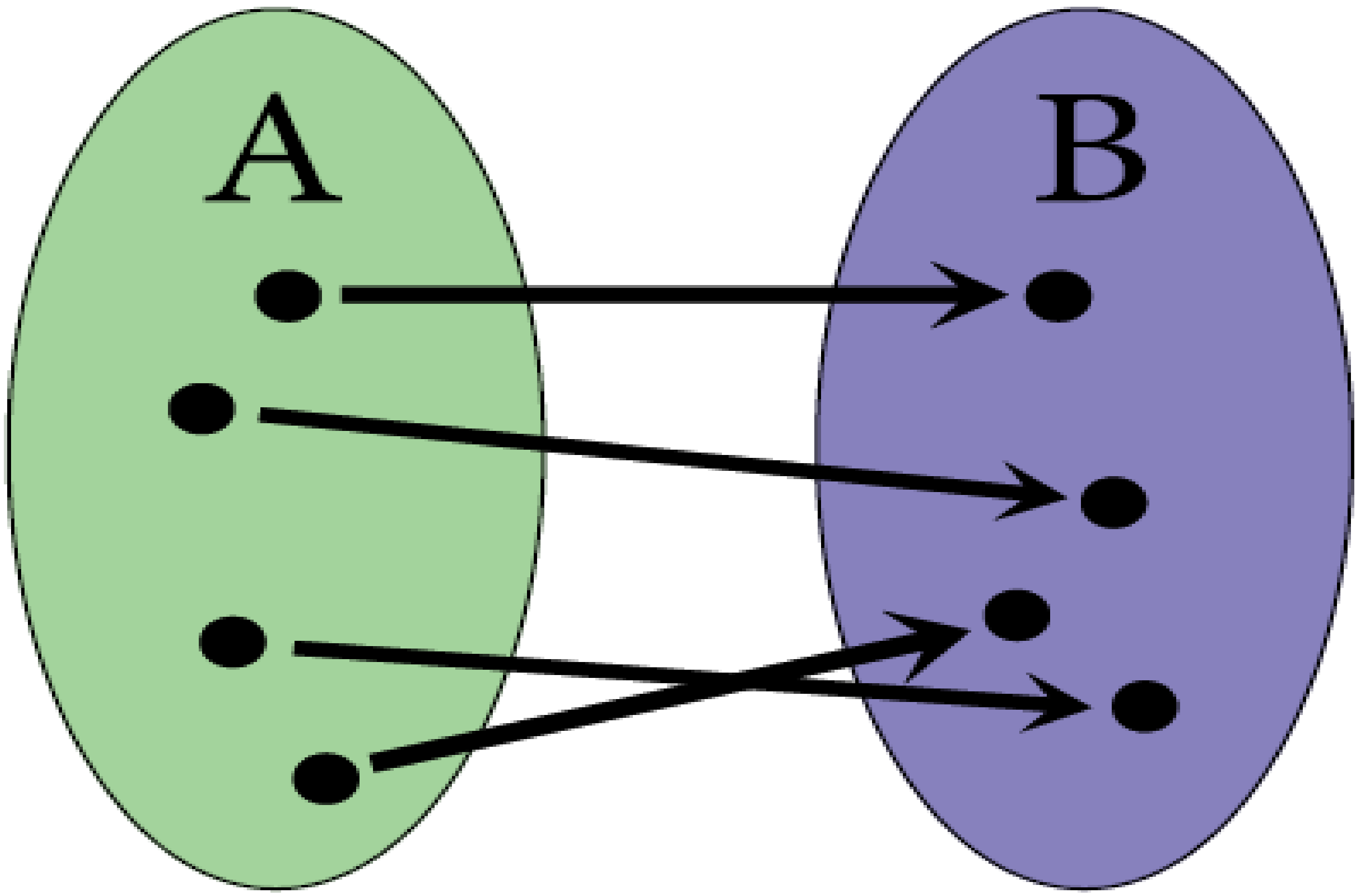
Injective Function



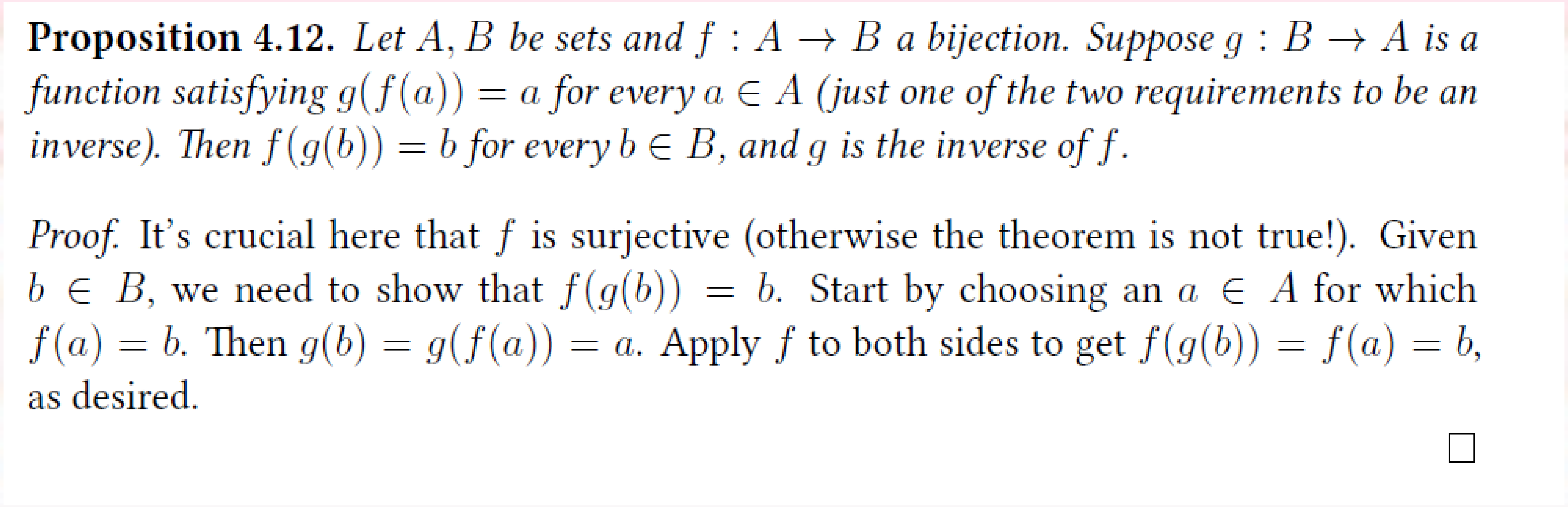
Surjective Function



Bijective (both Injective and Surjective)



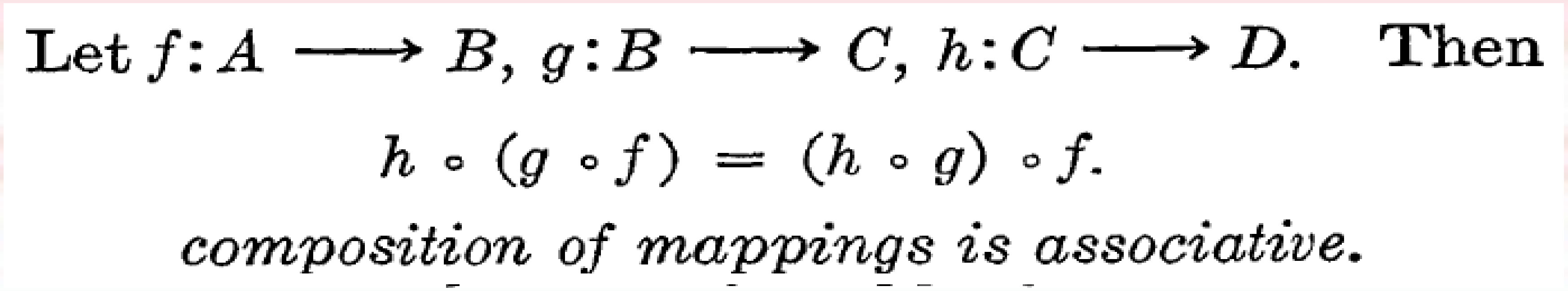
Inverse Function



Proposition 4.12. *Let A, B be sets and $f : A \to B$ a bijection. Suppose $g : B \to A$ is a function satisfying $g(f(a)) = a$ for every $a \in A$ (just one of the two requirements to be an inverse). Then $f(g(b)) = b$ for every $b \in B$, and g is the inverse of f.*

Proof. It's crucial here that f is surjective (otherwise the theorem is not true!). Given $b \in B$, we need to show that $f(g(b)) = b$. Start by choosing an $a \in A$ for which $f(a) = b$. Then $g(b) = g(f(a)) = a$. Apply f to both sides to get $f(g(b)) = f(a) = b$, as desired. $\square$

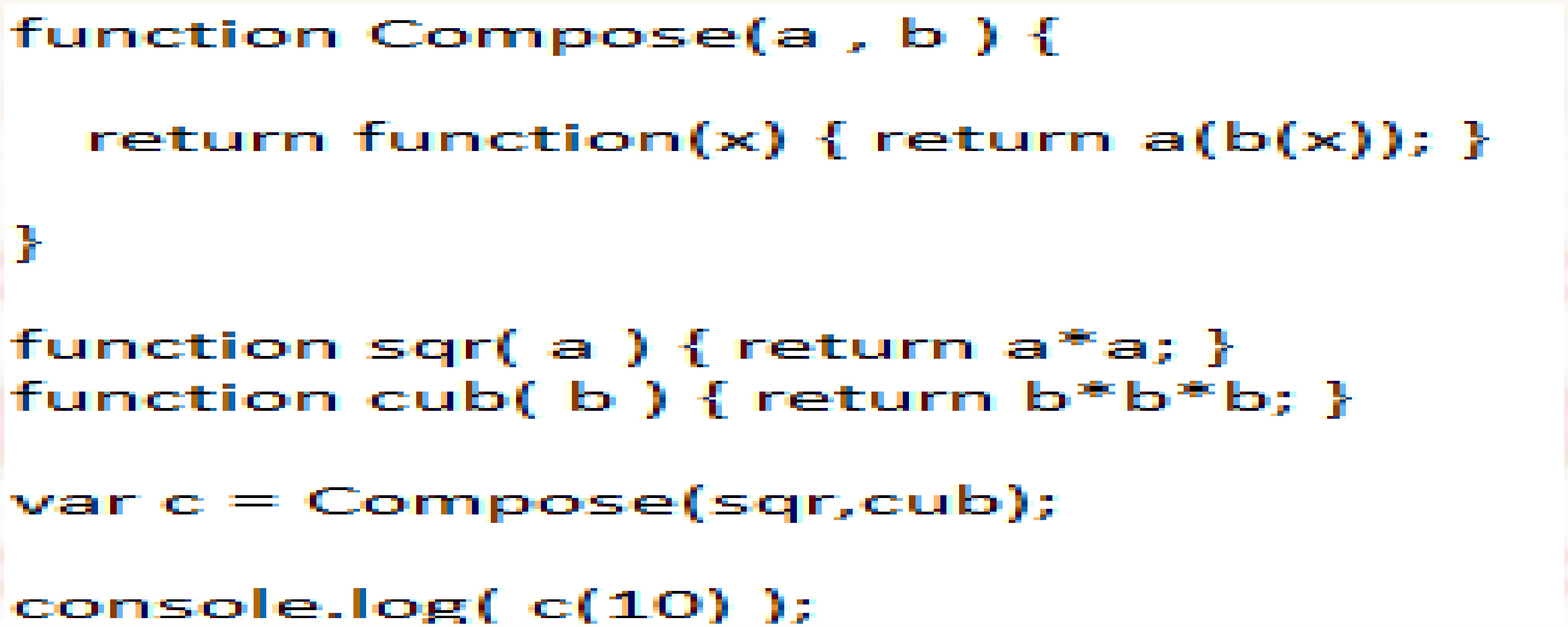
Functions are Compossible



Let $f\colon A \longrightarrow B$, $g\colon B \longrightarrow C$, $h\colon C \longrightarrow D$. Then

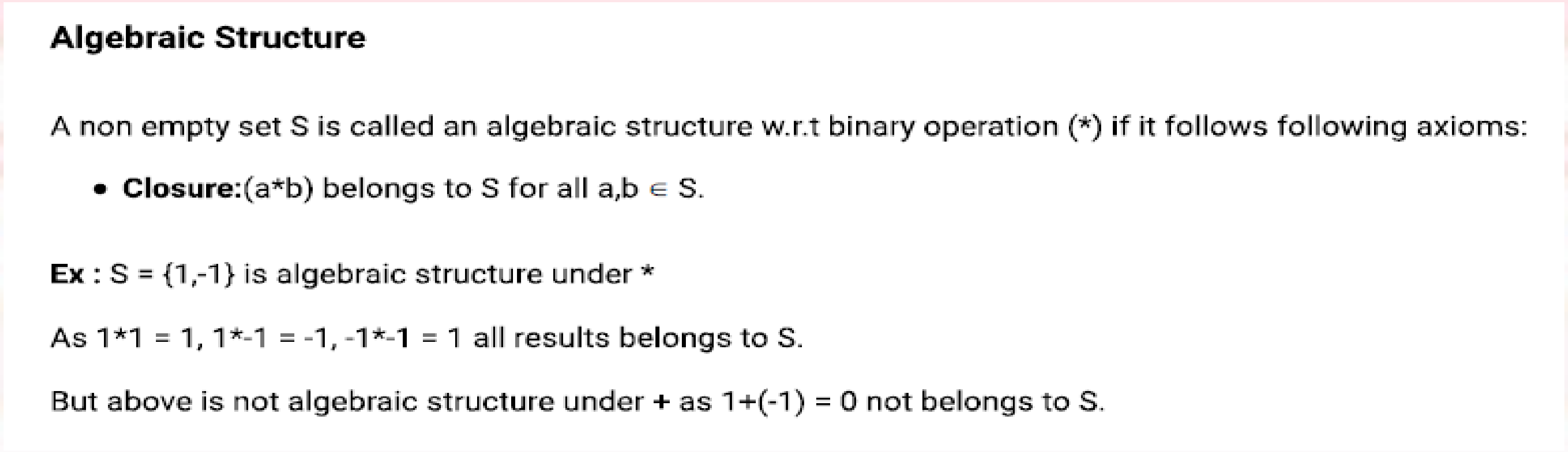
$$h \circ (g \circ f) = (h \circ g) \circ f.$$

composition of mappings is associative.



```
function Compose(a , b ) {

   return function(x) { return a(b(x)); }

}

function sqr( a ) { return a*a; }
function cub( b ) { return b*b*b; }

var c = Compose(sqr,cub);

console.log( c(10) );
```

Algebraic Structure



Algebraic Structure

A non empty set S is called an algebraic structure w.r.t binary operation (*) if it follows following axioms:

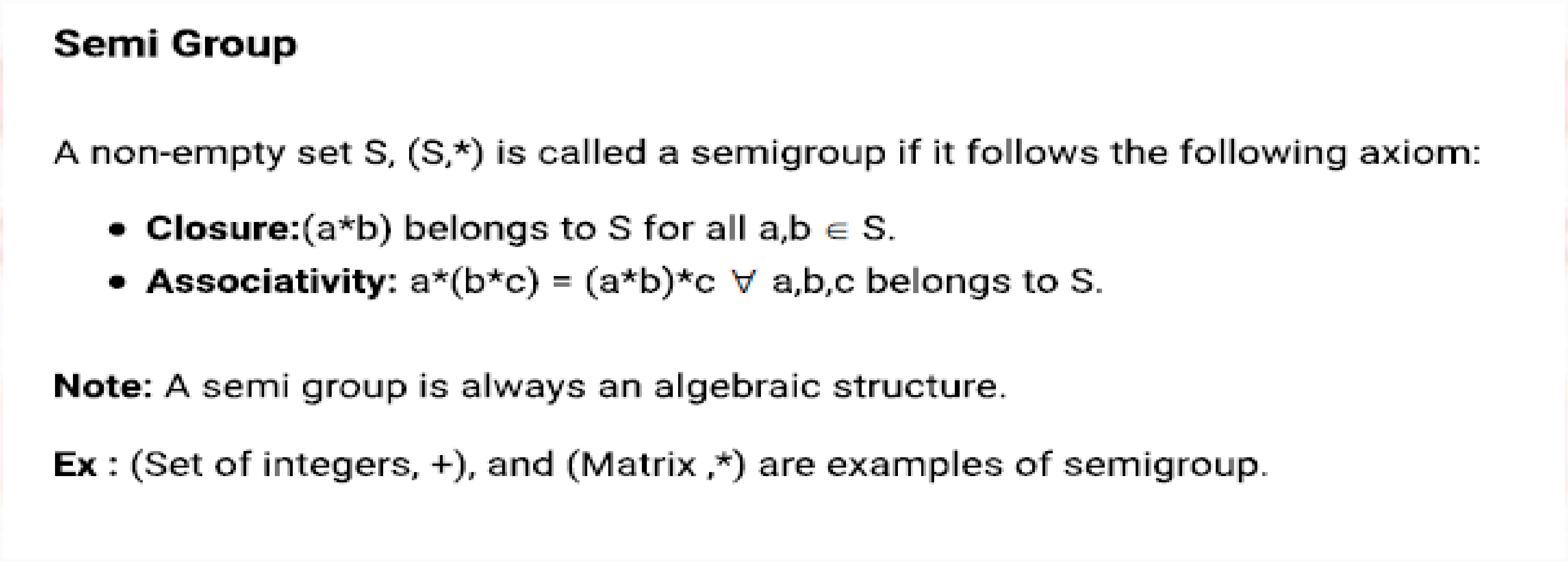
- **Closure:**(a*b) belongs to S for all a,b ∈ S.

Ex : S = {1,-1} is algebraic structure under *

As 1*1 = 1, 1*-1 = -1, -1*-**1** = 1 all results belongs to S.

But above is not algebraic structure under **+** as 1+(-1) = 0 not belongs to S.

SemiGroup



Semi Group

A non-empty set S, (S,*) is called a semigroup if it follows the following axiom:

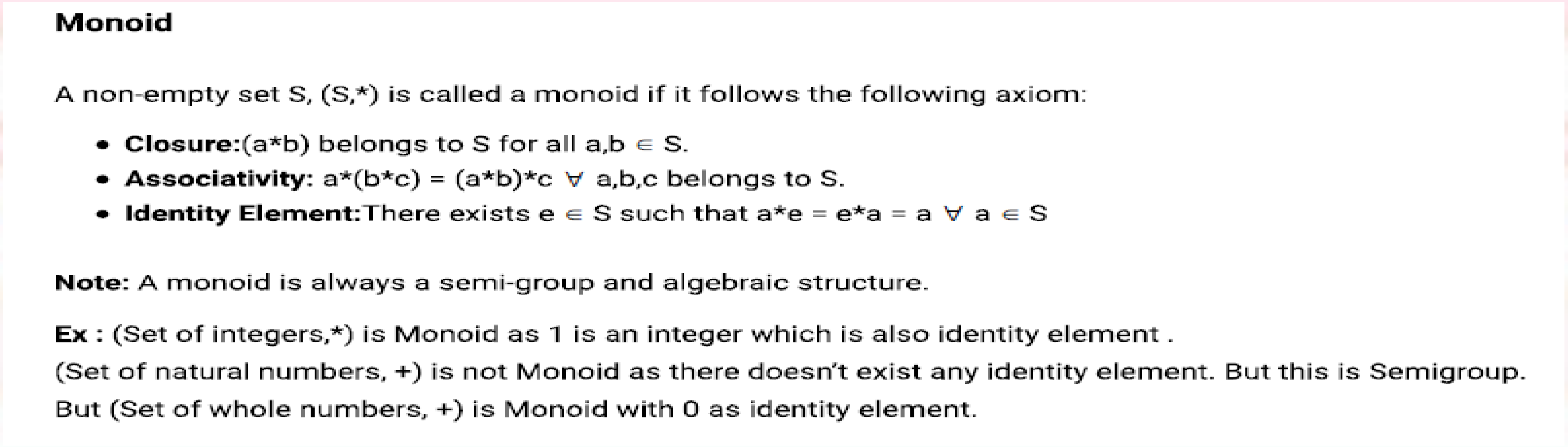
- **Closure:**(a*b) belongs to S for all a,b ∊ S.
- **Associativity:** a*(b*c) = (a*b)*c ∀ a,b,c belongs to S.

Note: A semi group is always an algebraic structure.

Ex : (Set of integers, +), and (Matrix ,*) are examples of semigroup.

```
trait SemiGroup[A] {

  def combine(x: A, y: A): A
}
```

Monoid



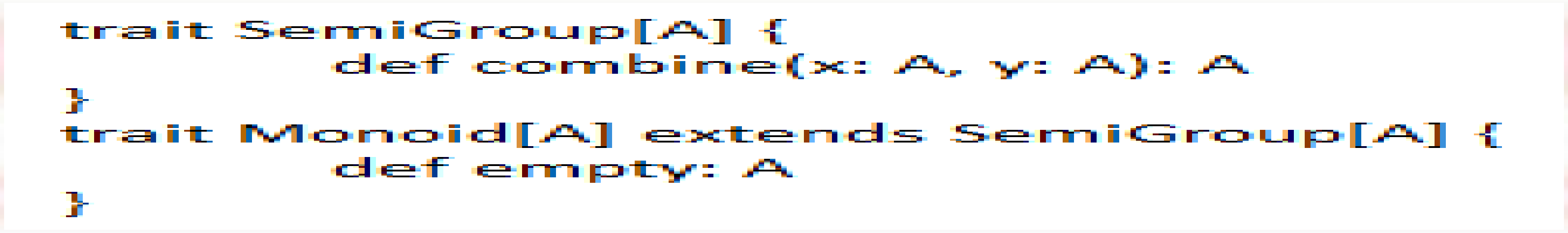
Monoid

A non-empty set S, (S,*) is called a monoid if it follows the following axiom:

- **Closure:**(a*b) belongs to S for all a,b ∈ S.
- **Associativity:** a*(b*c) = (a*b)*c ∀ a,b,c belongs to S.
- **Identity Element:**There exists e ∈ S such that a*e = e*a = a ∀ a ∈ S

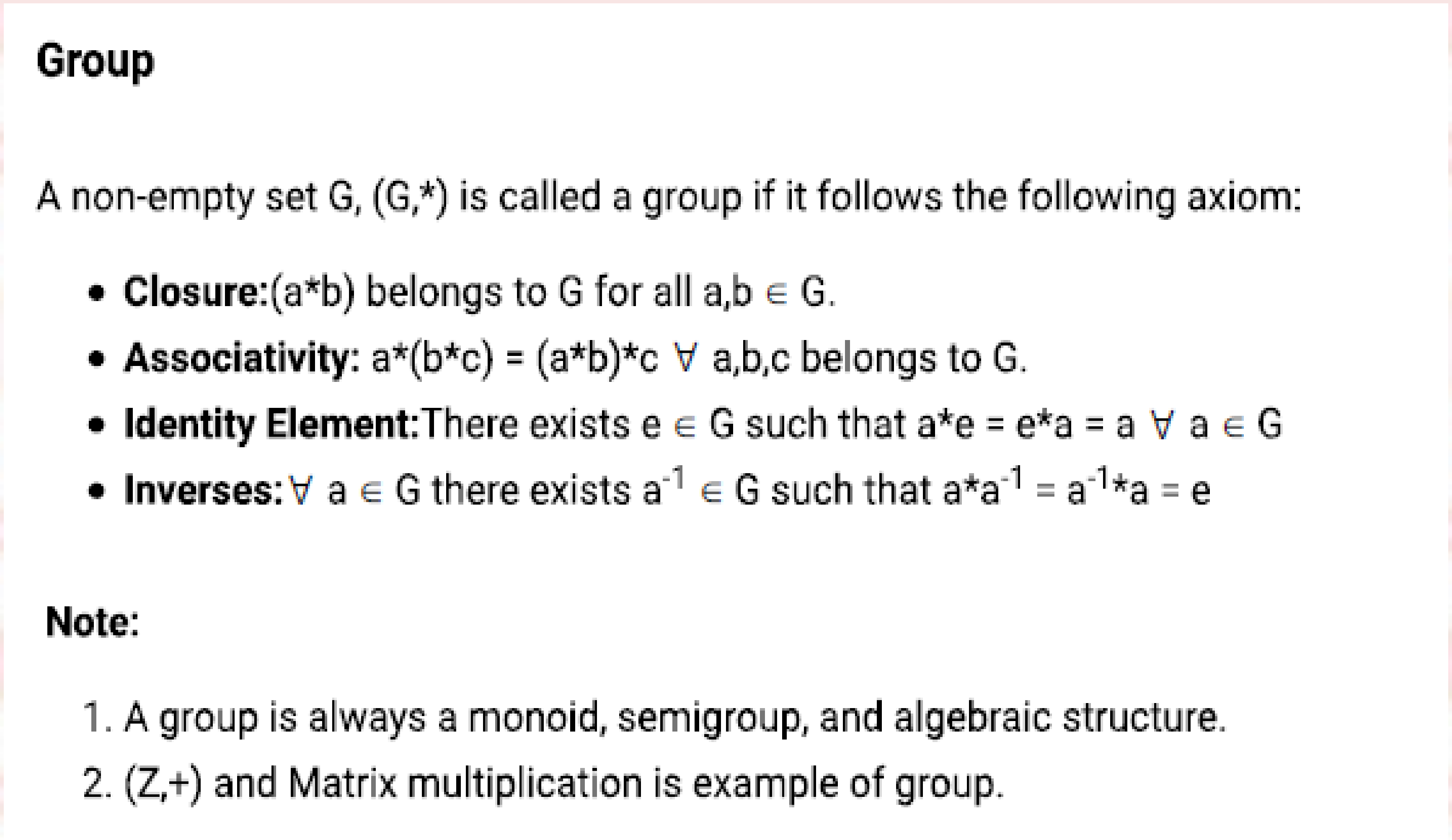
Note: A monoid is always a semi-group and algebraic structure.

Ex : (Set of integers,*) is Monoid as 1 is an integer which is also identity element .
(Set of natural numbers, +) is not Monoid as there doesn't exist any identity element. But this is Semigroup.
But (Set of whole numbers, +) is Monoid with 0 as identity element.



```
trait SemiGroup[A] {
          def combine(x: A, y: A): A
}
trait Monoid[A] extends SemiGroup[A] {
          def empty: A
}
```

Group



Group

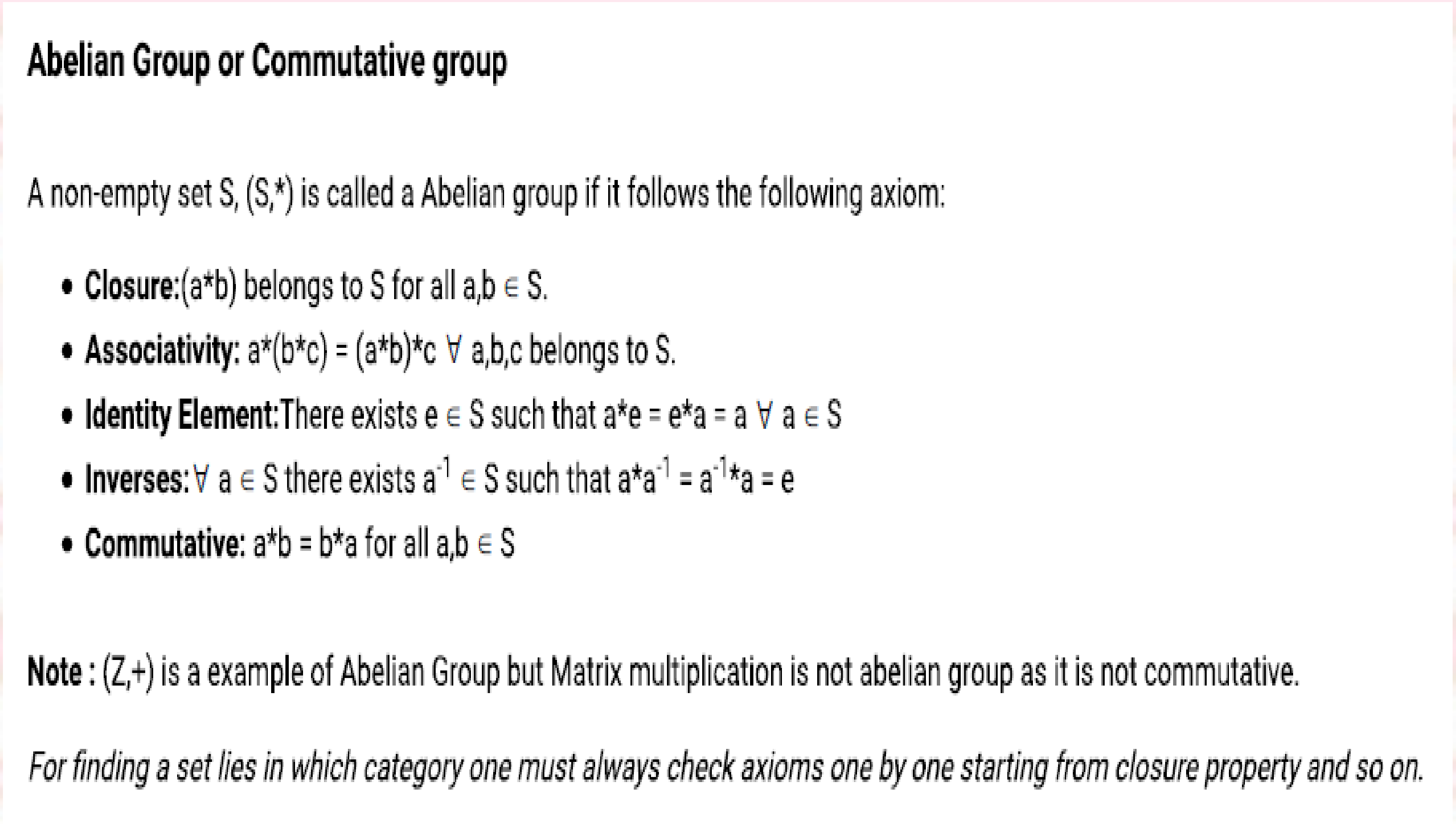
A non-empty set G, (G,*) is called a group if it follows the following axiom:

- **Closure:**(a*b) belongs to G for all a,b ∈ G.
- **Associativity:** a*(b*c) = (a*b)*c ∀ a,b,c belongs to G.
- **Identity Element:**There exists e ∈ G such that a*e = e*a = a ∀ a ∈ G
- **Inverses:**∀ a ∈ G there exists a^{-1} ∈ G such that $a*a^{-1} = a^{-1}*a = e$

Note:

1. A group is always a monoid, semigroup, and algebraic structure.
2. (Z,+) and Matrix multiplication is example of group.

AbelianGroup



Abelian Group or Commutative group

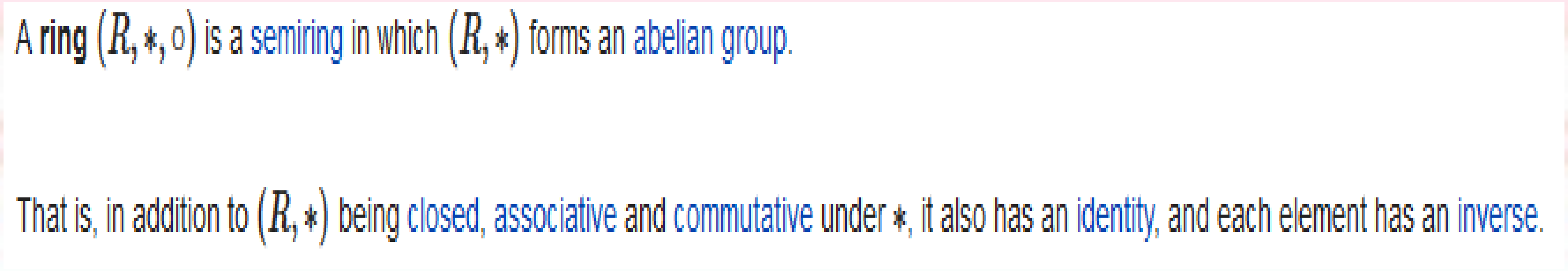
A non-empty set S, (S,*) is called a Abelian group if it follows the following axiom:

- **Closure:**(a*b) belongs to S for all a,b ∊ S.
- **Associativity:** a*(b*c) = (a*b)*c ∀ a,b,c belongs to S.
- **Identity Element:**There exists e ∊ S such that a*e = e*a = a ∀ a ∊ S
- **Inverses:**∀ a ∊ S there exists a^{-1} ∊ S such that $a*a^{-1} = a^{-1}*a = e$
- **Commutative:** a*b = b*a for all a,b ∊ S

Note : (Z,+) is a example of Abelian Group but Matrix multiplication is not abelian group as it is not commutative.

For finding a set lies in which category one must always check axioms one by one starting from closure property and so on.

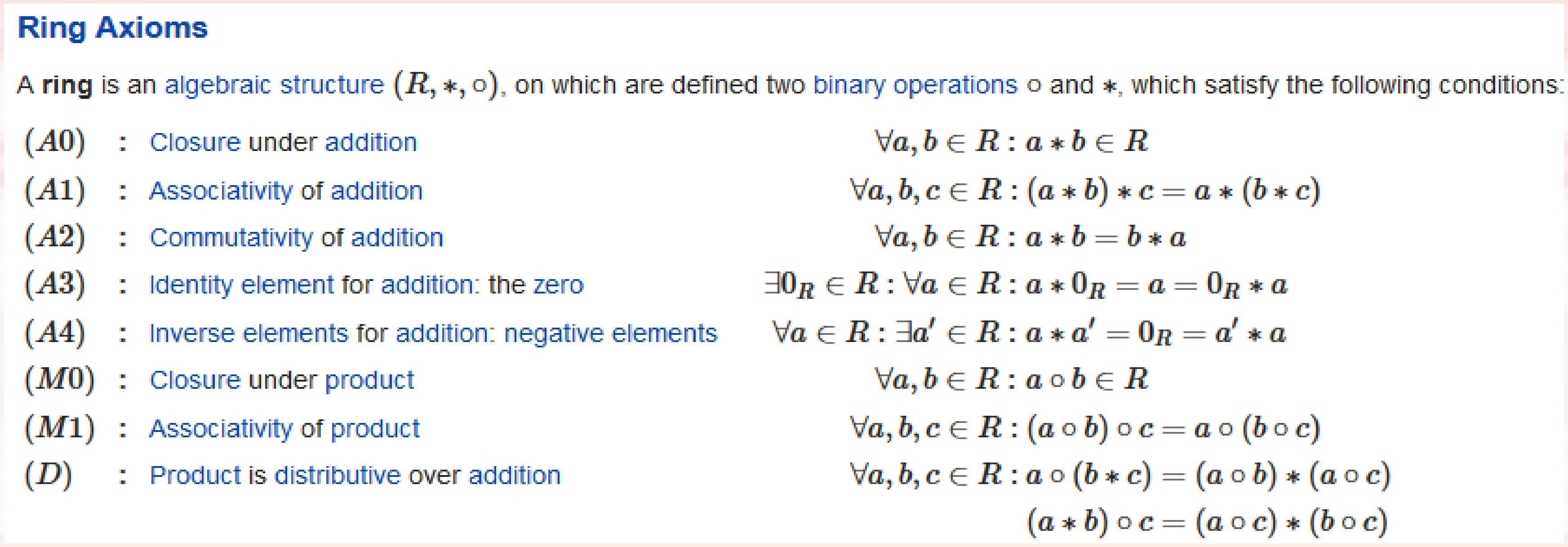
Ring Structure



A **ring** $(R, *, \circ)$ is a semiring in which $(R, *)$ forms an abelian group.

That is, in addition to $(R, *)$ being closed, associative and commutative under $*$, it also has an identity, and each element has an inverse.

Ring Properties

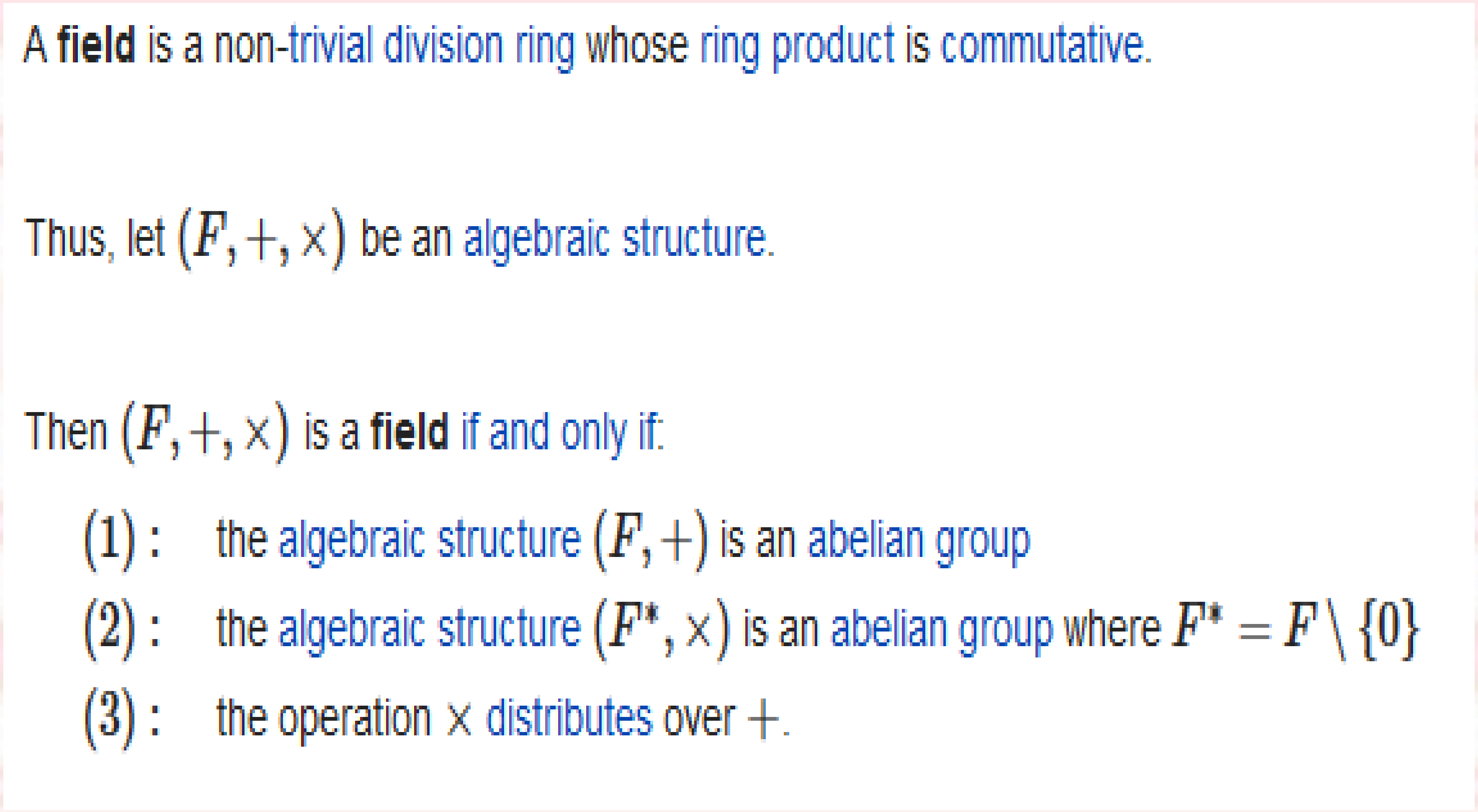


Ring Axioms

A **ring** is an algebraic structure $(R, *, \circ)$, on which are defined two binary operations $\circ$ and $*$, which satisfy the following conditions:

$(A0)$	:	Closure under addition	$\forall a, b \in R : a * b \in R$
$(A1)$	:	Associativity of addition	$\forall a, b, c \in R : (a * b) * c = a * (b * c)$
$(A2)$	:	Commutativity of addition	$\forall a, b \in R : a * b = b * a$
$(A3)$	:	Identity element for addition: the zero	$\exists 0_R \in R : \forall a \in R : a * 0_R = a = 0_R * a$
$(A4)$	:	Inverse elements for addition: negative elements	$\forall a \in R : \exists a' \in R : a * a' = 0_R = a' * a$
$(M0)$	:	Closure under product	$\forall a, b \in R : a \circ b \in R$
$(M1)$	:	Associativity of product	$\forall a, b, c \in R : (a \circ b) \circ c = a \circ (b \circ c)$
(D)	:	Product is distributive over addition	$\forall a, b, c \in R : a \circ (b * c) = (a \circ b) * (a \circ c)$
			$(a * b) \circ c = (a \circ c) * (b \circ c)$

Field Structure



A **field** is a non-trivial division ring whose ring product is commutative.

Thus, let $(F, +, \times)$ be an algebraic structure.

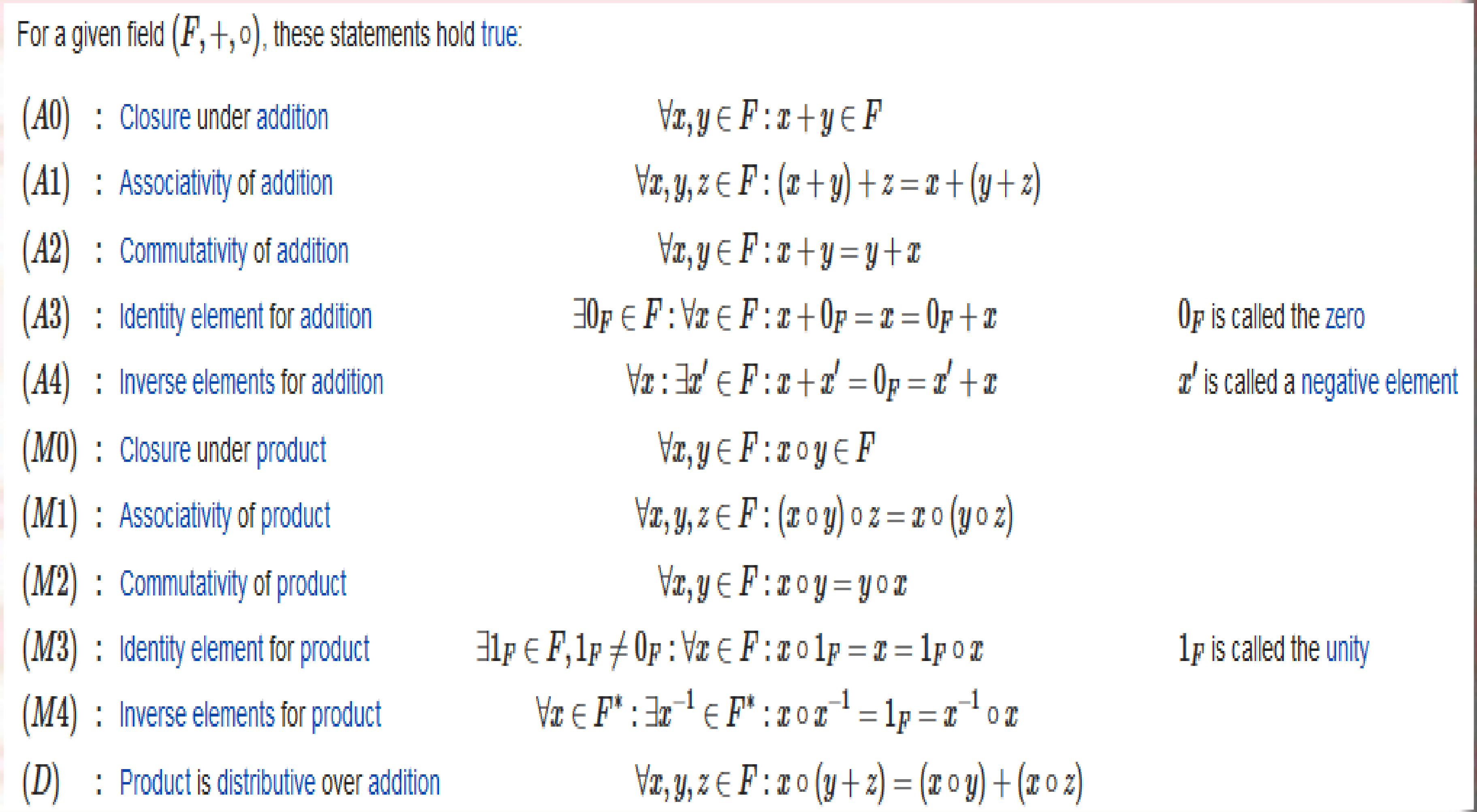
Then $(F, +, \times)$ is a **field** if and only if:

$(1):$ the algebraic structure $(F, +)$ is an abelian group

$(2):$ the algebraic structure $(F^*, \times)$ is an abelian group where $F^* = F \setminus \{0\}$

$(3):$ the operation $\times$ distributes over $+$.

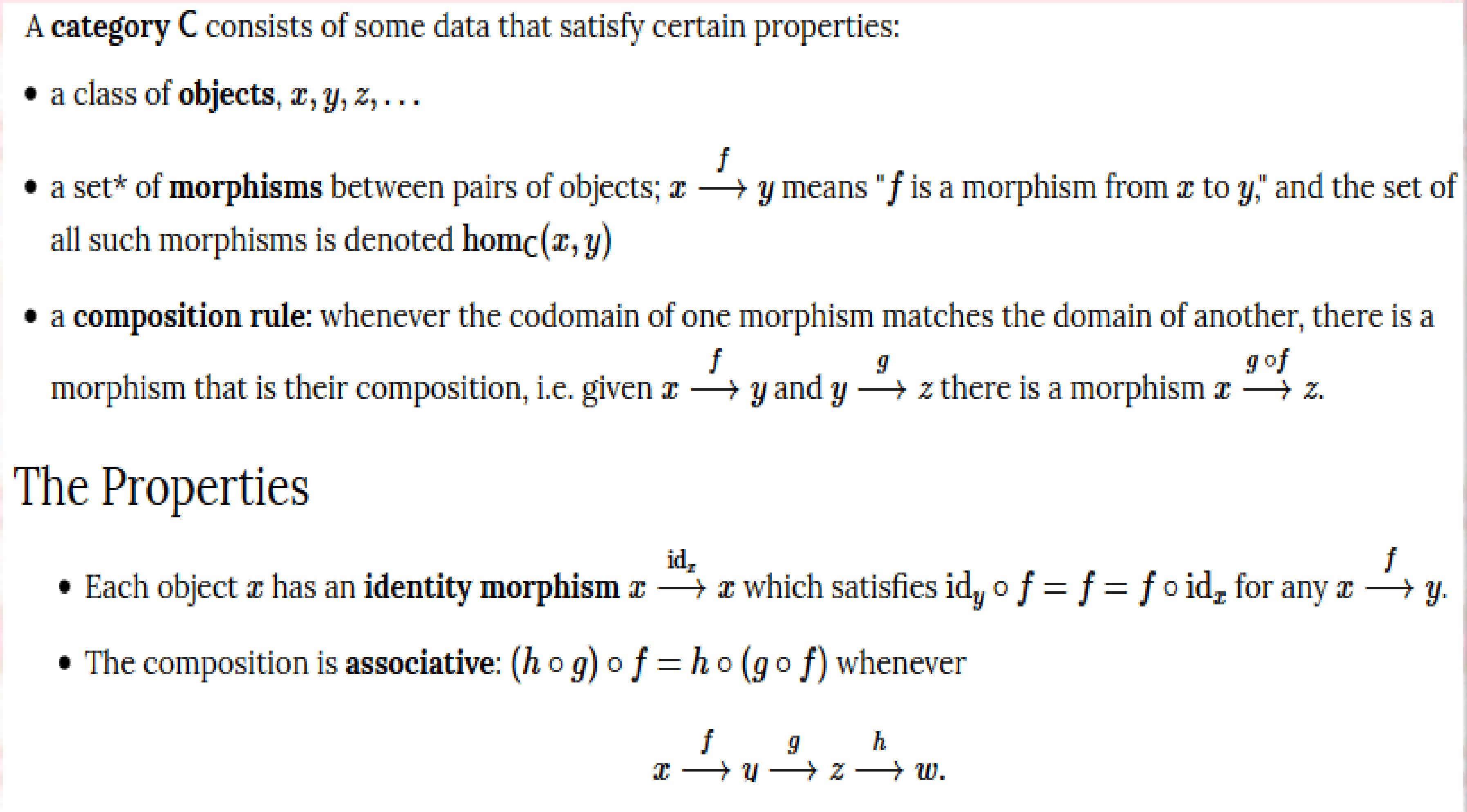
Field Properties



For a given field $(F,+,\circ)$, these statements hold true:

$(A0)$	:	Closure under addition	$\forall x,y \in F : x+y \in F$	
$(A1)$	:	Associativity of addition	$\forall x,y,z \in F : (x+y)+z = x+(y+z)$	
$(A2)$	:	Commutativity of addition	$\forall x,y \in F : x+y = y+x$	
$(A3)$	:	Identity element for addition	$\exists 0_F \in F : \forall x \in F : x + 0_F = x = 0_F + x$	0_F is called the zero
$(A4)$	:	Inverse elements for addition	$\forall x : \exists x' \in F : x + x' = 0_F = x' + x$	x' is called a negative element
$(M0)$	:	Closure under product	$\forall x,y \in F : x \circ y \in F$	
$(M1)$	:	Associativity of product	$\forall x,y,z \in F : (x \circ y) \circ z = x \circ (y \circ z)$	
$(M2)$	:	Commutativity of product	$\forall x,y \in F : x \circ y = y \circ x$	
$(M3)$	:	Identity element for product	$\exists 1_F \in F, 1_F \neq 0_F : \forall x \in F : x \circ 1_F = x = 1_F \circ x$	1_F is called the unity
$(M4)$	:	Inverse elements for product	$\forall x \in F^* : \exists x^{-1} \in F^* : x \circ x^{-1} = 1_F = x^{-1} \circ x$	
(D)	:	Product is distributive over addition	$\forall x,y,z \in F : x \circ (y+z) = (x \circ y) + (x \circ z)$	

What is a Category?



A **category** C consists of some data that satisfy certain properties:

- a class of **objects**, $x, y, z, \ldots$
- a set* of **morphisms** between pairs of objects; $x \xrightarrow{f} y$ means "f is a morphism from x to y," and the set of all such morphisms is denoted $\hom_{\mathsf{C}}(x, y)$
- a **composition rule:** whenever the codomain of one morphism matches the domain of another, there is a morphism that is their composition, i.e. given $x \xrightarrow{f} y$ and $y \xrightarrow{g} z$ there is a morphism $x \xrightarrow{g \circ f} z$.

The Properties

- Each object x has an **identity morphism** $x \xrightarrow{\mathrm{id}_x} x$ which satisfies $\mathrm{id}_y \circ f = f = f \circ \mathrm{id}_x$ for any $x \xrightarrow{f} y$.
- The composition is **associative**: $(h \circ g) \circ f = h \circ (g \circ f)$ whenever

$$x \xrightarrow{f} y \xrightarrow{g} z \xrightarrow{h} w.$$

Functor

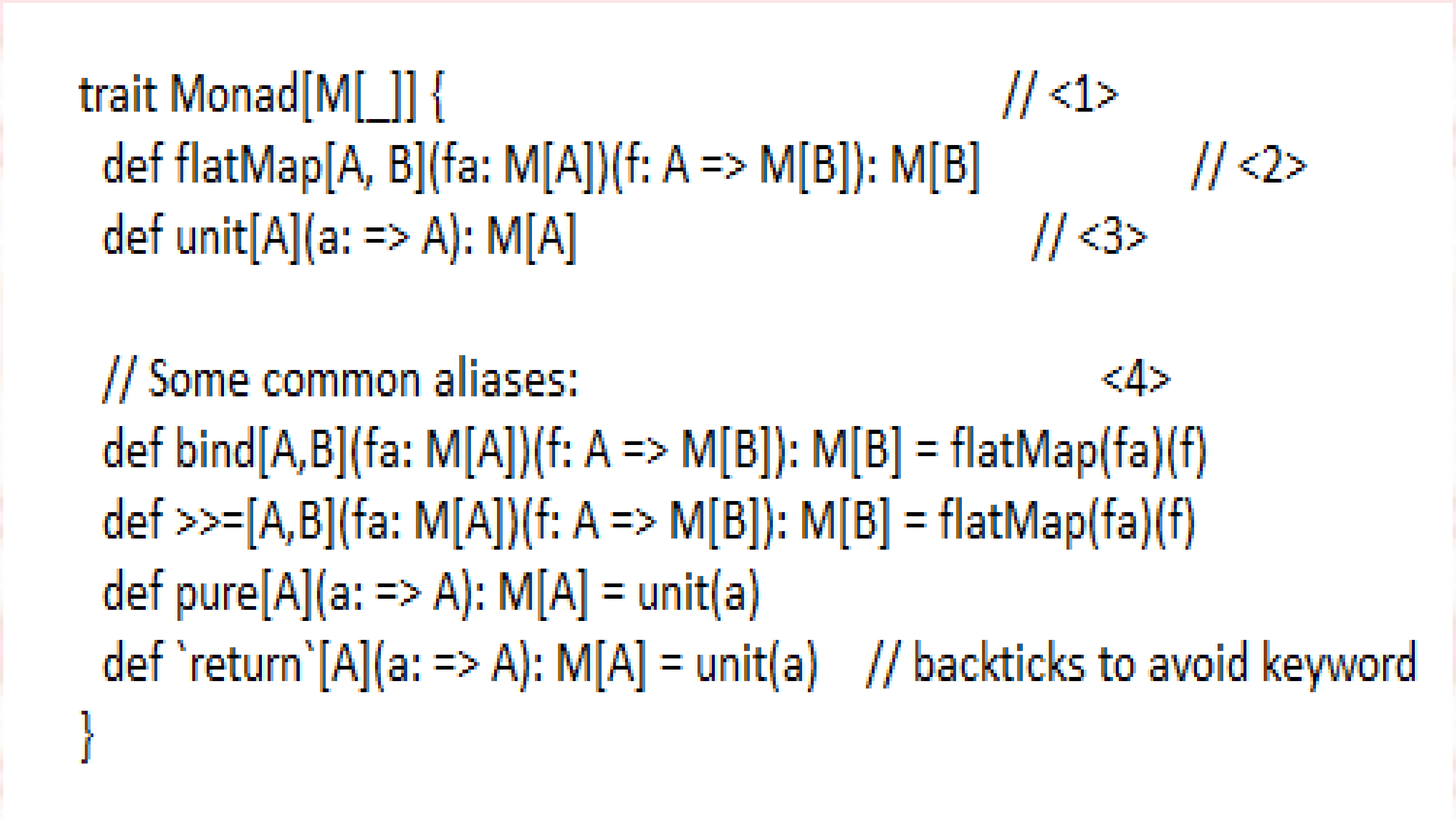
- Functor is a Type which has got an associated Map Method

Monad

- A Monad is a Category which Obeys
  - Associativity
  - Left Identity
  - Right Identity
- A Monadic Type will be having
  - Map
  - FlatMap
  - A Lift Function which “lifts” value

  (To a Monadic Type)

Monad (Interface)



```
trait Monad[M[_]] {                                       // <1>
  def flatMap[A, B](fa: M[A])(f: A => M[B]): M[B]              // <2>
  def unit[A](a: => A): M[A]                               // <3>

  // Some common aliases:                                      <4>
  def bind[A,B](fa: M[A])(f: A => M[B]): M[B] = flatMap(fa)(f)
  def >>=[A,B](fa: M[A])(f: A => M[B]): M[B] = flatMap(fa)(f)
  def pure[A](a: => A): M[A] = unit(a)
  def `return`[A](a: => A): M[A] = unit(a)    // backticks to avoid keyword
}
```

Algebraic Types (Some theoretical notions)

- Sum Type (discriminated union , C/C++ union, microsoft's variants etc)
- Product Type (records, tuples etc in most languages)
- Exponential Object (Models Function Application and Currying)
- The Sum Type/Product Type/Exponential Objects form a CCS (Cartesian Closed Category)
- Modern Statically typed language follows the semantics of CCS

Questions

- If any ?