Scala

Statically typed → compiler knows the type of every value before the program runs

This markdown file documents everything learned in Week 1 to 6 of 50.054 Compiler Design and Program Analysis related to Scala

Lambda Calculus to Scala

back-to-top

Lambda Calculus	Scala
Variable	x
Constant	1, 2, true, false
Lambda abstraction λx.t	(x: T) ⇒ e
Function application t_1 t_2	e1(e2)
Conditional	if (e1) { e2 } else { e3 }
Let Binding	val x = e1; e2
Recursion	def f(x:Int): Int = f(x); f(1)

Every Scala expression can be represented in lambda calculus form.

Scala runs imperatively → step-by-step like C or Java

Other Terminologies

back-to-top

REPL

Read-Eval-Print Loop → an interactive programming environment where you can type code line by line and immediately see the result.

Scala is a REPL

Immutable

Cannot be changed after creation

Try

- A container type that represents a computation that may either result in a value `(Success)` or an exception `(Failure)`. It is used for error handling without throwing exceptions

Why for what?

Try (with `Success` and `Failure`) lets you represent computations that can fail, capturing either the result `(Success)` or the error `(Failure)` without throwing exceptions.

Success and Failure

- A subclass of `Try`

```
Success(42)
// means the computation succeeded and produced the value 42
```

Some and None

- a subclass of Option that wraps a value to indicate presence.

```
Some(2)
```

↑ This means the option contains value 2, if there is no value, use `None`

Why for what?

Lets you represent the presence and/or absence of a value safely → a function that might not return a result can return `Option[Int]` instead of just `Int`

Running Scala

back-to-top
Make sure your file extensions is `.scala`

Make sure all your code is nested under an object that extends app

```scala
object name_of_file extends App {

  // Code here

}
```

1. Using Scalac and Scala commands

Make sure to cd into the right directory first

```
# Compile
scalac name_of_file.scala

# Run
scala name_of_file.scala
```

2. Using build.sbt (Alternative)

Make sure the scala files you are running is in exactly:

- src
  - main
    - scala
      - name_of_file.scala

```
# Activate sbt at the root directory
sbt

# Running
clean
compile
run

# If you want it to compile and run everytime u save
~run

# For specific files
~runMain name_of_file
```

3. VSC inbuilt run (Another alternative way to run Scala)

Need to download Scala CLI first

Running test cases in cohort

back-to-top

```
# Running all Test Cases
test

# Running one Specific Test Case
testOnly *TestEx1
```

Functions and Methods

back-to-top

```scala
def name_of_function (name_of_parameters : type_of_parameter) : return_type = {
   body_of_function
}

// Example:
def add(x:Int, y:Int): Int = {
  x + y
}
```

- `def`
  - Keyword to define a function with type parameters
- `add`
  - Function name
- `(x:Int, y:Int)`
  - Function will take in parameter x, with type Integer and y, with type Integer as well
- `:Int`
  - Return type will be an Int

Another Example

```scala
def flatten[A](l: List[List[A]]) = { }
```

- `[A]` → Type parameter
  - it makes the function generic, meaning it can work on lists of any type
- `(l :List[List[A]])` → Parameter list
  - function takes one parameter called l
- `= {}` → `{}` is the function body

OOP in Scala

back-to-top

```scala
trait FlyBehavior{
  def fly()
}

abstract class Bird(species:String, fb:FlyBehavior){
  def getSpecies():String = this.species
  def fly():Unit=this.fb.fly()
}

class Duck extends Bird("Duck", new FlyBehavior(){
  override def fly() = println("I can't fly!")
})

class BlueJay extends Bird("BlueJay", new FlyBehavior(){
  override def fly() = println("Swoosh")
})
```

- trait defines an interface
- class constructors are in-line
- Unit is a type, similar to void in Java
- Functions and methods' return types can be inferred by compiler

Variable Types

back-to-top

Val vs Var

```scala
// immutable
val x = 5

// mutable
var y = 10
```

Type Inference

Scala often figures out type automatically

```
val n = 42 //inferred as Integer
val s = "hi" //inferred as String
val n: Int = 42 //or you can declare yourself
```

Expressions vs Statements

back-to-top

- Expression has value *e.g. 1+2 evaluates to 3*
- Statement does something but has no meaningful value *e.g. println("hi")*

If Else

back-to-top

```
val max = if (a>b) a else b
```

- if a is bigger than b, return a, else return b

Recursion

back-to-top

Regular Recursion

```
def sum(l: List[Int]): Int = l match {
  case Nil      => 0
  case x :: xs  => x + sum(xs)
}

sum(List(1,2,3))

// sum(1::2::3::Nil)
// → 1 + sum(2::3::Nil)
// → 1 + (2 + sum(3::Nil))
// → 1 + (2 + (3 + 0))
// Unwind call stack
// → 1 + (2 + 3 )
// → 1 + 5
// → 6
```

Each recursive call has to wait for the next one to finish

Tail Recursion Optimisation

- recursive call is the last operation
- "throws" away the previous recursion

```scala
import scala.annotation.tailrec

def sumTail(l: List[Int]): Int = {
  @tailrec
  def go(lst: List[Int], acc: Int): Int = lst match {
    case Nil      => acc
    case x :: xs  => go(xs, acc + x)  // recursive call is LAST
  }
  go(l, 0)
}

sumTail(List(1,2,3))

// go([1,2,3], 0)
// → go([2,3], 1)
// → go([3], 3)
// → go([], 6)
// → 6
// Done ~
```

- `acc`
  - the 'accumulator', a variable that accumulates or keeps track of the running total of the computations as we go deeper into recursion
  - instead of waiting to add everything after coming back up, just add along the way

Map

back-to-top

- Idea: apply the function to every element in the list

```scala
val nums = List(1,2,3,4)
val doubled = nums.map(x => x * 2)
println(doubled)

// Output: List(2,4,6,8)
```

[Bad Example] Trying to add 10 to each elements in the list

```scala
def addToEach(x:Int, l:List[Int]):List[Int] = l match {
  case Nil => Nil
  case (y::ys) => {
    val yx = y+x
    yx::addToEach(x,ys)
  }
}

// addToEach(10, List(1,2,3))

// Input x=10, l = [1,2,3]
// y = 1, ys = [2,3]
// Compute yx = 1 + 10 = 11
// Call yx::addToEach(10, [2,3])

// Input x=10, l = [2,3]
// y = 2, ys = [3]
// yx = 2 + 10 = 12
// Call yx::addToEach(10, [3])

// Input x=10, l = [3]
// y = 3, ys = []
// yx = 3 + 10 = 13
// Call yx::addToEach(10, [])

// Final call his Nil

// 13 :: Nil
// 12 :: (13::Nil)
// 11 :: (12 :: (13::Nil))
// Result = List(11, 12, 13)
```

From ↑ to ↓

```scala
def addToEach(x:Int, l:List[Int]):List[Int] = l.map(y=>y+x)
```

FlatMap

- Map + Flatten (joining multiple lists into 1)

```
// Regular Map
val nums = List(1, 2, 3)
val result = nums.map(x => List(x, x + 1))

// List(List(1, 2), List(2, 3), List(3, 4))


// FlatMap
val nums = List(1, 2, 3)
val result = nums.flatMap(x => List(x, x + 1))

// List(1, 2, 2, 3, 3, 4)
```

Fold

back-to-top

- Reduce a collection down to a single value by repeatedly applying a function
- has:
  - foldLeft → process left to right
  - foldRight → process right to left
  - fold → pick a direction depending on the collection, usually left
- Requires:
  - A starting value (the accumulator).
  - A binary operation (a function that combines the accumulator with each element).

General Form

```
list.fold(initialValue)((acc,element) => newAcc)

// Start with initialValue,
// then for each element,
// update the acc (accumulator).
```

Example

```
//  Goal : 1 + 2 + 3 + 4
val nums = List(1, 2, 3, 4)
val result = nums.fold(0)((acc, x) => acc + x)
println(result)


// Step 0: acc = 0, x = null, new acc = null
// Step 1: acc = 0, x = 1, new acc = acc + x = 1
// Step 2: acc = 1, x = 2, new acc = acc + x = 3
// Step 3: acc = 3, x = 3, new acc = acc + x = 6
// Step 4: acc = 6, x = 4, new acc = acc + x = 10

// Final output: (((0 + 1) +2) +3) +4
```

foldRight vs foldLeft

- foldLeft → Tail-recursive

```
List(1,2,3).foldLeft(0)(_ - _)

// ((0-1)-2)-3 = -6

List(1,2,3).foldRight(0)(_ - _)

// 1-(2-(3-0)) = 2
```

Filter

back-to-top

- Takes a predicate function (a function that returns true/false)
- Keeps only elenments for which the function is true
- Returns a new list, original stays unchanged

```
val nums = List(1,2,3,4,5,6)
val evens = nums.filter(x => x % 2 == 0)
println(evens)

Output:
List(2,4,6)
```

For expression in Scala

back-to-top

- combination of flatmap and map

```
for x in [1,2,3]
  x+1
```

is the same as:

```
map(lambda x:x+1, [1.2.3])
```

in Scala

```
for {x <- List(1,2,3)} yield (x+1)
```

using map

```
List(1,2,3).map(x=>x+1)
```

Enum (Algebraic Data Type example)

back-to-top

```scala
enum MathExp {
  case Plus(e1:MathExp, e2:MathExp)
  case Minus(e1:MathExp, e2:MathExp)
  case Multiply(e1:MathExp, e2:MathExp)
  case Div(e1:MathExp, e2:MathExp)
  case Const(v:Int)
}

import MathExp.*
val

def eval(e:MathExp):Int = e match {
  case Plus(e1, e2) => eval(e1) + eval(e2)
  case Minus(e1, e2) => eval(e1) - eval(e2)
  case Mult(e1, e2) => eval(e1) * eval(e2)
  case Div(e1, e2) => eval(e1) / eval(e2)
  case Const(i) => i
}
```

- `enum` keyword lets you define a closed set of possible values for a type
- `MathExp` is a type
- Each `case` (like `Plus`, `Minus`, etc) is a **constructor** → a way to create one "variant" of that type
- As all case belong to the same enum, the compiler knows all possibilities

Using enum (ADT example)

```scala
import MathExp.*
val expression = Mult(Plus(Const(1), Const(2), Const(3)))

// (1+2)*3
```

List

back-to-top

Creating an immutable list

```scala
val l = List(1, 2, 3)
```

- recall 'val' defines an immutable variable

Accessing Elements

```scala
l(1)
```

- Similar to python's indexing

Attempting to Modify a list

```scala
l(1) = 3
```

- *ERROR* → Scala's list are immutable

Iterating over a list

```scala
for (i <- l) {println(i)}

Output:
  1
  2
  3
```

Flattening a list

```scala
def flatten[A](l: List[List[A]]): List[A] =
  l.flatMap(inner => inner)

flatten(List(List(1,2), List(3,4), List(5)))
// Result: List(1, 2, 3, 4, 5)
```

Pattern Matching with Lists

- use Match

```
l match {
  case Nil => "empty"
  case (x :: xs) => "not empty"
}
```

Pattern Constructors

```
//Nil and '::'

Nil -> represents the empty list
:: -> constructs a list by perpending an element to another list
```

E.g.

```
1 :: 2 :: Nil = List(1,2)
```

- A list in scala is either:
  - Empty → Nil
  - Non-empty → a head element plus a tail list, written as head :: tail

Match

back-to-top

```
l match {
  case Nil      => "empty"
  case (x :: xs) => "not empty"
}
```

- match is like a switch-case for structured data
- in the above code, we are matching l with 2 potential patterns, an empty list or a non-empty list

- l is being tested against 2 patterns
- if l is an empty list, it will return "empty"
- if l is a non-empty list, it means x will match the first element (the head) and xs will match the tail (a list of all the remaining elements)

```
val list = List(10, 20, 30)

list match {
  case Nil       => println("empty")
  case (x :: xs) => println(s"head = $x, tail = $xs")
}
```

Output:

```
head = 10, tail = List(20,30)
```

Summing a List (match)

back-to-top

```
def sum(l: List[Int]): Int = {
  l match {
  case Nil     => 0
  case (x::xs) => x + sum(xs)
  }
}
```

- If the list is empty (Nil), return 0.
- If the list has a head (x) and tail (xs), return x + sum(xs).
  → This recursively sums all elements.

Example:

```
sum(List(1,2,3))
// = 1 + sum(List(2,3))
// = 1 + (2 + sum(List(3)))
// = 1 + (2 + (3 + sum(Nil)))
// = 1 + (2 + (3 + 0))
// = 6
```

Indexing an element (match)

back-to-top

```
def first(l: List[Int]): Int = l match {
  case Nil => None
  case (x::xs) => x
}
```

- This is like accessing an element but with extra steps
- Why is this better? It handles an OutOfBounds error by returning None instead
- def first(....) → defining a function named first
- Parameter: l:List[Int] → l is a list of integers
- : Int → the function returns an Int

findMax() function (match)

back-to-top

```
def findMax(l:List[Int]):Int = {
  l match{

    case Nil => Int.MinValue
    case (x::xs) =>
      val tailMax = findMax(xs)
      if (x>tailMax) x else tailMax
  }
}
```

- def findMax → defines a function named findMax

- (l: List[Int]) → takes one parameter l, which must be a list of integers
- : Int → return type is an integer
- = → everything after the = is the body

Case #1

```
case Nil => Int.MinValue
```

- if l is empty, return Int.MinValue
- MinValue is the built-in smallest integer of scala (-2,147,483,648)
- We need to return this because we are guarenteeing that any real integer in the list will be greater than this
- In the event the list has only 1 value that is -50 for sure, it is STILL the biggest one despite being the only one, so it will be compared to the Int.MinValue and "win"

Case #2

```
case x :: xs =>
  val tailMax = findMax(xs)
  if (x > tailMax) x else tailMax
```

- if l is non-empty, Scala splits it into
  - x → the head (first element)
  - xs → the tail (the rest of the list)
- Then, it recursively calls findMax on the tail(xs) which will find the maximum of the rest of the listL
- it compares the head of the list(x) with the maximum of the tail (tailMax)
- if head is larger, return x, else return taiMax

Example

1. **Start**
   - `l = List(3, 7, 2)` → not empty
   - `x = 3` , `xs = List(7, 2)`
   - `tailMax = findMax(List(7, 2))`

2. **Recursive call: `findMax(List(7, 2))`**
   - `x = 7` , `xs = List(2)`
   - `tailMax = findMax(List(2))`

3. **Recursive call: `findMax(List(2))`**
    - `x = 2` , `xs = Nil`
    - `tailMax = findMax(Nil)`

4. **Base case: `findMax(Nil)`**
    - Returns `Int.MinValue`

5. **Unwinding recursion**
    - Compare `2` vs `Int.MinValue` → **2**
    - Compare `7` vs `2` → **7**
    - Compare `3` vs `7` → **7**

✅ Final result: `7`

Span Function

back-to-top

INCOMPLETE

```
// span function takes a list and returns a list of pairs
// each pair consists of the first element and each of the other elements in the list
// span(List(1, 2, 3, 4))
// Output: List((1,2), (1,3), (1,4))
def span[A](l: List[A], check: A => Boolean): (List[A], List [A]) = l match {
    case Nil => Nil
    case (x::xs) if check(x) => {
        val (ys, zs) = span(xs, check)
        (x::ys, zs)
    }
}
```

Currying

back-to-top

Normal Form

```
def tax(price:Double, rate:Double):Double = price * rate

// When calling tax, need to always provide price and rate as arguments
```

Currying Ver/Form

```
def curriedTax(rate: Double)(price: Double): Double = price * rate
  val gst_with_rate = curriedTax(0.08)(_: Double)
  val gst_with_price = curriedTax(_: Double)(100)

// Good because i can FIX rate and have price as argument or FIX price and have rate as argument
```

Why?

- Code reusability
- Partial application, plusone becomes a function, dont diedie need 2 parameters

Function Composition

back-to-top

- Combining 2 or more functions to produce a new function

```scala
def f(x: Int): Int = x + 1
def g(x: Int): Int = x * 2

val h = f andThen g // h(x) = g(f(x))
val k = f compose g // k(x) = f(g(x))

h(3) // f(3) = 4, g(4) = 8 → result: 8
k(3) // g(3) = 6, f(6) = 7 → result: 7

andThen: f andThen g = g(f(x))
compose: f compose g = f(g(x))
```

Generic/ Polymorphic ADT

back-to-top

recall ADT

```scala
enum List[+A] {
  case Nil
  case Cons(x: A, xs: MyList[A])
}
```

- `List[+A]` is generic: it can be `List[Int]` , `List[String]` , etc.
- BUT it cannot have both Int and String in the same

Subtyping inside Enum

back-to-top

```scala
enum Shape {
  case Circle(radius: Double)
  case Rectangle(width: Double, height: Double)
}

// Double is a variable type in Scala, similar to float

val c: Shape = Shape.Circle(2.0)
val r: Shape = Shape.Rectangle(3.0, 4.0)

println(area(c)) // Output: 12.566...
println(area(r)) // Output: 12.0
```

- `Shape.Circle` and `Shape.Rectangle` are both subtypes of `Shape`
- You can write functions that accept `Shape` and use pattern matching to handle each subtype
- It allows you to treat all cases as the same base type (Shape), but also access specific data for each subtype

Covariant and Invariant

back-to-top

(not tested)

```scala
class InvariantBox[A]
class CovariantBox[+A]

val cov: CovariantBox[Any] = new CovariantBox[String] // OK
val inv: InvariantBox[Any] = new InvariantBox[String] // Error
```

- the + determines whether its Co or In

Function Overloading

back-to-top

- Using the same function name for multiple implementations in different type context

```scala
def greet(name: String): String = s"Hello, $name!"
def greet(name: String, age: Int): String = s"Hello, $name! You are $age years old."
```

- Issue 1: Type mismatch error
- Issue 2: Duplicated Code

case Class

back-to-top

- An enum with only 1 constructor can be rewritten as a case class

```scala
enum Person {
  case Person (name: String, contacts:List[Contact])
}

// can be written as:

case class Person(nume: String, contacts: List[Contact])
```

- For what?
  - `name` and `contacts` are immutable → once you create a Person, you cannot change their name or contacts
  - Less boilerplate
  - Access to automatic methods like equals, hashCode, toString, and copy
  -

Type Class (Traits)

back-to-top

- recall Trait in Java is a class where its methods are compulsory when instantiated
- a "contract" for behavior that you can give to any type, even those you didn't write youtself

Imagine you have different kinds of vehicles: cars, bikes, boats. You want to "drive" them, but not all vehicles have a steering wheel or pedals. Instead of forcing every vehicle to inherit from a "Drivable" class, you create a "Drivable" contract. If a vehicle can be driven, you provide instructions for how to drive it.

Type class: The "Drivable" contract.
Type class instance: The specific instructions for each vehicle.
Generic function: A function that can "drive" any vehicle, as long as it has instructions.

```
// Define a type class called Drivable for any type A
trait Drivable[A] = {
  // Specifies that any type A with a Drivable instance MUST HAVE the drive method
  def drive(a: A): String
}

// Type Class instances
// These are case classes representing different vehicle types
case class Car(model: String)
case class Bike(brand: String)
case class Boat(name: String)

///////////////////////////////////////////////////////
// You only need a case class when you want to create a new type.
// For built-in types, you can directly create type class instances.
///////////////////////////////////////////////////////

// Drivable instance for Car (Preferred for Scala 3)
given Drivable[Car] with {
  def drive (c:Car): string = s"Driving car: ${c.model}"
}

// Drivable instance for Car (Older but still works)
given carDrivable: Drivable[Car] = new Drivable[Car] {
  def drive(c: Car): String = s"Driving car: ${c.model}"
}
// c is the name
// Car is the type class instance
```

- `given` → the type class instance keyword

Class Example

```
trait JS[A] {
  def toJS(v:A):String
}

// Type class instance for Int
given JS[Int] with {
  def toJS(v: Int): String = v.toString
}

// Type class instance for String
given JS[String] with {
  def toJS(v: String): String = s"'${v}'"
}


given toJSList[A](using jsa:JS[A]):JS[List[A]] = new JS[List[A]] {
  def toJS(as:List[A]):String = {
    val j = as.map(a=>jsa.toJS(a)).mkString(",")
    s"[${j}]"
  }
}
```

- `toJSList` is used to define a type class instance for `List[A]`
- `val j...mkString(", ")` → For each element a in the list, convert it to JS using
- `enum` variable type is private, whereas `type class` can still change (recall enum)

Higher Kinded Type & Functor Type Class

back-to-top

```
// General 'Formula'
trait Functor[F[_]] {
  def map[A,B](ta:F[A])(f:A => B):F[B]
}
```

- `[F[_]]` means the trait `Functor` is taking in a type F that has type generic
- `[A,B]` → `A` is the type of the elements inside the container F.

- `B` is the type you get after applying the function f to each element.
- `(ta:F[A])` → Argument: Container is a F of A which can be a list of Int for e.g. - `(f:A => B)` → The Function that transfer A to B - `F[B]` → the output is a container of B

Why for what?

- Lets you transform data inside containers in a safe, consistent way
- Keeps the structure, if you map over a list, you still get a list

Analogy

Functor → universal remote control that works on many different TVs

The "on" button → Functor's Map function

You give it a rule (e.g., “+1” / “uppercase” / “add subtitles”)

When you press the "on" button, its like saying "Map add 10"

The remote will apply "add 10" to any TV you point at

The TV doesnt change, only its content change

It’s a standard contract: ***“If you give me any structure F[*]** **that’s a functor, I can transform the inside values without knowing what F really is.”_**

Without Functor

```
def do10xList(list: List[Int]): List[Int] = list.map(x => x * 10)
def do10xListOption(opt: Option[Int]): Option[Int] = opt.map(x => x * 10)
def do10xListTry(t: Try[Int]): Try[Int] = t.map(x => x * 10)
```

With Functor

```scala
// General
trait Functor[F[_]] {
  def map[A,B](ta:F[A])(f:A => B):F[B]
}

// Instantiating it --> example of using a List
def do10xListGeneric[F[_]](
    container: F[Int]
)(using functor: Functor[F]): F[Int] = {
  functor.map(container)(x => x * 10)
}

// Using the functor
println(do10xListGeneric(List(1, 2, 3)))
println (do10xListGeneric(Option(5)))
println(do10xListGeneric(Try(7)))


// Now we can use the do10xListGeneric method with Int, Try, Option
```

Kinds vs Types

back-to-top

- **Values** like `42`, `"hello"` have **Types** `Int`, `String` respectively
- **Types** themselves have a "type of types" → and those are called **KINDS**
- **Kinds** describes what shape of type arguments a type constructor accepts
- Kinds are to types what types are to values
- A type constructor like `List` has kind _ → _
- List[_] by itself is not a type → takes 1 type argument (of kind _) and produces a new type (also of kind _)

```scala
val list = List(1,2,3)
```

- `list.head` has value: 1, Type: Int, Kind: *
- `list` has value List(1,2,3), Type: List(Int), Kind : *
- List(the type constructor) has Kind: *→*

Functor Laws

back-to-top

1. Identity Law

Mapping with the identity function (id(x) = x) should not change the functor’s contents.

```
List(1,2,3).map(x => x) == List(1,2,3)
```

2. Composition Law

Recall function composition

```
fa.map(f).map(g) == fa.map(g compose f)

val f: Int => Double = _ * 2.0
val g: Double => String = _.toString

List(1,2,3).map(f).map(g)
// = List("2.0", "4.0", "6.0")

List(1,2,3).map(g compose f)
// = List("2.0", "4.0", "6.0")
```

Why the need?

A 'bad' functor could cheat:

- ignore the function you give to `map`
- apply the functon multiple times/ in the wrong order
- randomly shuffle or duplicate elements

Foldable Type Class

back-to-top

recall fold

Just a trait that defines `foldLeft` and/or `foldRight` methods → lets you write code that can fold any foldable type, not just lists

Why for what?

Lets you write functions that work for any foldable type, not just `List` or `Option` that Scala has inbuilt `foldLeft` and `foldRight` for.

For example you may want to fold other things like `trees` , with foldable type class you can define how folding works for your tree type where inside it you can use generic code like `foldLeft`

```scala
// Define a binary tree
// Goal: Reduce all values in binary tree into a single result
sealed trait Tree[+A]
case class Leaf[A](value: A) extends Tree[A]
case class Branch[A](left: Tree[A], right: Tree[A]) extends Tree[A]

// Implement fold for Tree
def foldTree[A, B](tree: Tree[A], acc: B)(f: (B, A) => B): B = tree match {
  case Leaf(value)      => f(acc, value)
  case Branch(l, r)     =>
    val leftFolded = foldTree(l, acc)(f)
    foldTree(r, leftFolded)(f)
}

// Example usage: sum all values in a tree
val tree: Tree[Int] = Branch(Leaf(1), Branch(Leaf(2), Leaf(3), Leaf(4)))
val sum = foldTree(tree, 0)(_ + _)
println(sum)
```

- `foldTree` recursively visits every node and combines values using the function `f`
- You can use it to sum, multiply, or process any tree of values

Option

back-to-top

- The Option type in Scala is used to represent a value that may or may not exists → safer alternative to `null`

- `Option[A]` is a container that represents "maybe a value" → either Some(a:A) when a value exists, or None when it doesn't.

It is Scala's safe alternative to null and expresses optionality in the type system.

```
val s: Option[String] = Some("hello")
val n: Option[String] = None
val maybe = Option(null)    // => None
val also = Option("world")  // => Some("world")
```

Uses of Option

Replacing null

```
// before:
val name: String = System.getenv("NAME")
val display = if (name!=null) name else "Guest"

// Option
val nameOpt: Option[String] = Option(System.getenv("NAME"))
val display = nameOpt.getOrElse("Guest")
```

Error Handling with Option Type

back-to-top
A better alternative to try-catch exception

```
enum Option[+A] {
    case None
    case Some(v:A)
}
```

`Some` is a case class that represents the presence of a value in an Option.

`Some(value)` is the alternative to `None` in Scala's `Option` type

- `Some(value)` means there is a value

- `None` means there is no value

Both together are the 2 possible cases for an `Option`

```
def eval(e:MathExp):Option[Int] = e match {
case MathExp.Plus(e1, e2) => eval(e1) match {
  case None => None
    case Some(v1) => eval(e2) match {
      case None => None
      case Some(v2) => Some(v1 + v2)
  }
}
// case for catching division by 0
// if u divide by 0 it just returns 'None' by checking the denominator
case MathExp.Div(e1, e2) => eval(e1) match {
  case None => None
  case Some(v1) => eval(e2) match {
    case None => None
    case Some(0) => None
    case Some(v2) => Some(v1 / v2)
  }
}
case MathExp.Const(i) => Some(i)
}

eval(Div(Const(1), Minus(Const(2), Const(2))))
// Output: None
```

Error Handling with Either Type

back-to-top

- Handle Errors with Option Type has one remaining issue
    - i.e. there is no info with the error in None → if got multiple cases to check multiple errors, we dont if the error is us trying to divide by 0 or sth else

```scala
def eval(e: MathExp): Either[ErrMsg,Int] = e match {
    case MathExp.Plus(e1, e2) =>
    eval(e1) match {
      case Left(m) => Left(m)
      case Right(v1) => eval(e2) match {
        case Left(m) => Left(m)
        case Right(v2) => Right(v1 + v2)
      }
    }
// cases omitted for Minus and Mult
case MathExp.Div(e1, e2) =>
  eval(e1) match {
    case Left(m) => Left(m)
    case Right(v1) => eval(e2) match {
      case Left(m) => Left(m)
      case Right(0) => Left(s"div by zero caused by ${e.toString}")
      case Right(v2) => Right(v1 / v2)
  }
}
case MathExp.Const(i) => Right(i)
}
eval(Div(Const(1), Minus(Const(2), Const(2))))
// yields Left(div by zero caused by Div(Const(1),Minus(Const(2),Const(2))))
```

Derived Type Class

back-to-top

- A trait that extends trait
- for what? Dont need nearly identical boilerplate for every case class

```scala
trait Rectangle[A] {

}

trait Square[A] extends Eq[A] {

}
```

- Every `Square` is also a `Rectangle`, but not every `Rectangle` is a `Square`

Example of Derived Type Class

```scala
case class Person(name: String, age: Int)

given Show[Person] with
  def show(p: Person): String =
    s"Person(name=${p.name}, age=${p.age})"

// Implicit Instantiation --> "Whenever you need a Show[Person], use it like this"

val p = Person("Ada", 30)
println(summon[Show[Person]].show(p))
// Output: Person(name=Ada, age=30)
```

Now you add a field to the model: Person has country and you did not change the Show

```scala
case class Person(name:String, age:Int, country: String)

val p2 = Person("Ada", 30, "UK")
println(summon[Show[Person]].show(p2))
```

Output: `Person(name=Ada, age=30)`
The country is missing from the output!

Manual Solution : Edit the Show instance to include the new field

```scala
given Show[Person] with
  def show(p: Person): String =
    s"Person(name=${p.name}, age=${p.age}, country=${p.country})"
```

Now imagine you have 500 new field to add.

(Continued) Derived show implementation

back-to-top

Using the `derives` keyword → indicating it is a derived type class

```scala
case class Person(name: String, age: Int) derives Show
val p = Person("Ada", 30)
println(summon[Show[Person]].show(p))
// Person(Ada, 30)
```

Now if you need to add a field

```scala
case class Person(name: String, age: Int, country: String) derives Show
val p = Person("Ada", 30, "UK")
println(summon[Show[Person]].show(p))
// Person(Ada, 30, UK)
```

- tells Scala to automatically generate (derive) an implicit `Show[Person]` instance for you.

Applicative Functor

back-to-top

- Sits between `Functor` and `Monad`
- Allows us to apply functions that take multiple arguments to values

```scala
// General 'formula'
trait Applicative[F[_]] extends Functor[F] {
  def pure[A](a: A): F[A]
  def ap[A, B](ff: F[A => B])(fa: F[A]): F[B]
  def map[A, B](fa: F[A])(f: A => B): F[B] = ap(pure(f))(fa)
}
```

1. `pure` : put a plain value into the context
2. `map2/ap` : apply a function to values that are inside contexts

Think of context as a box around a value

- `Option[A]` = maybe-a-value box
- `List[A]` = many-values box
- `Future[A]` = value-arrives-later box

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

Applicative Functor allows you to use the same `add` but on numbers that are inside context without opening the context

Difference with Functor

`Functor` = you have one box `F[A]`. You can run a 1-arg function on what's inside that one box.

`Applicative` = you have several boxes `F[A]`, `F[B]`, … You can run a normal multi-arg function on the contents of all those boxes at once (as long as they don't depend on each other).

Example

To list all outfits, pair every shirt with every pant

```scala
def map2[A, B, C](fa: List[A], fb: List[B])(f: (A, B) => C): List[C] =
  for { a <- fa; b <- fb } yield f(a, b)


val shirts = List("Blue", "White")
val pants  = List("Jeans", "Khakis")

val outfits = map2(shirts, pants)((s, p) => s"$s + $p")
// List("Blue + Jeans", "Blue + Khakis", "White + Jeans", "White + Khakis")
```

Explanation:

map2: take two wrapped values (lists) and a pure function → combine all values inside using that function.

1. Parameter #1: the containers

```scala
(fa: List[A], fb: List[B])
```

`fa` is a list of shirts
`fb` is a list of pants

2. Paramter #2: the combining function

```
(f: (A, B) => C)
```

Takes one A and one B and produces a C (e.g. a full outfit).

```
for { a <- fa; b <- fb } yield f(a, b)

// Scala automatically rewrites as:

fa.flatMap(a => fb.map(b => f(a, b)))
```

Part	What it does
`a <- fa`	Pull out each element `a` from the first list `fa` (“outer loop”).
`flatMap`	Handles iterating the outer list and *flattening* the nested lists produced later.
`b <- fb`	For each `a` , iterate through all elements `b` of the second list (`fb`).
`map`	Transforms each `b` into `f(a, b)` and returns a `List[C]` .
`yield f(a,b)`	Collects each computed value into the resulting list.

Execution:

1. s = "Blue"
   → map over pants: "Blue + Jeans", "Blue + Khakis"
2. s = "White"
   → map over pants: "White + Jeans", "White + Khakis"

Then flatMap joins them into:

```
List("Blue + Jeans", "Blue + Khakis", "White + Jeans", "White + Khakis")
```

Applicative Laws

back-to-top

Identity

```
ap(pure(x => x))(v) == v
```

If you lift the *identity function* `(x => x)` into the Applicative and apply it to any wrapped value `v`, nothing changes

Homomorphism

Interchange

Composition

Monad

Monad = Applicative + the power to let each next step depend on the previous result (via flatMap / bind).

- `Functor` : you can transform the thing inside one box (map).
- `Applicative` : you can use a normal multi-arg function on several independent boxes (map2, mapN).
- `Monad` : you can decide the next box based on the previous box's value (flatMap / bind).

In other words: later steps may depend on earlier results.

Think: “run step 1; see its result; based on that, choose step 2.”

```
// General 'Formula'

trait Monad[F[_]] /* extends Applicative[F] */ {
  def pure[A](a: A): F[A]
  def flatMap[A,B](fa: F[A])(f: A => F[B]): F[B]  // aka bind
  // map & ap can be derived if you have flatMap + pure
}
```

Example with Option

```scala
def parseInt(s: String): Option[Int] =
  s.toIntOption

def safeDiv(a: Int, b: Int): Option[Int] =
  if b == 0 then None else Some(a / b)

// Monad style (flatMap = “then” that can *depend* on the previous value)
val result: Option[Int] =
  parseInt("42").flatMap { n =>
    parseInt("7").flatMap { d =>
      safeDiv(n, d)              // choice depends on d
    }
  }
// for-comprehension sugar:
val result2 =
  for {
    n <- parseInt("42")
    d <- parseInt("7")
    q <- safeDiv(n, d)           // we can *branch* here
  } yield q
```

We use `d` to decide to continue `safeDiv` → that is the Monad dependency.

Applicative cannot express that branching.

Functor, Applicative and Monad

`Functor` lets you map over a wrapped value,

`Applicative` lets you apply wrapped functions to wrapped values, and

`Monad` lets you sequence computations where the next step can depend on the previous result.

TV Analogy

Functor: “I can transform what’s on a TV.”

Applicative: “I can apply wrapped presets to wrapped TVs (combine effects).”

Monad: “I can run a sequence where each next remote action is chosen using the previous result.”

When do I use which?

- Functor for when you need to map over one container
- Applicative when you need to combine several containers independently
- Monad when there is a Step B that depends on Step A

Syntax Analysis (5A)

back-to-top

- A compiler turns source text into target code by moving through distinct stages
- `Source Text → [Lexing] → [Parsing] → [Semantic Analysis] → [Optimization] → [Code Generation]`

Lexing

back-to-top

- **Input**: Source program in string (a list of characters)
- **Output**: a list of lexical tokens
- **Method**: Break things up into a sequence of tokens and ignore irrelevant things like whitespace and comments
- Fails when something can't be recognised as a lexical token

`tokens` are terminals of the grammar ("terminals = lexical tokens").

```
Grammar Notation
          (JSON) J ::= i | 's' | [] | [IS] | {NS}
        (Items) IS ::= J , IS | J
(Named Objects) NS ::= N, NS | N
  (Named Object) N ::= 's' : J
```

- Left Hand Side → Non-Terminals
- Right Hand Side → Terminals (Tokens) and Non-Terminals (Grammar Variables)
- `i` denotes an integer
- `'s'` denotes a string

Example (Simple)

```
val x = 2
```

We put the above into a lexer and it would produce:

```
[val][identifier:x][=][number:42]
```

Example (JSON)

Lexer Token Data Type Enum

All possible tokens the lexer can output

```
enum LToken {
  case IntTok(v:Int)
  case StrTok(v:String)
  case SQuote
  case LBracket; case RBracket
  case LBrace;  case RBrace
  case Colon;   case Comma
  case WhiteSpace
}
```

```
{'k1':1,'k2':[]}
```

We put the above into a lexer and it would produce:

```
List(LBRace, SQuote, StrTok("k1"), SQuote, Colon, IntTok(1), Comma, SQuote,
     StrTok("k2"), SQuote, Colon, LBracket, RBracket, RBrace)
```

Notice this illustrates how characters are grouped into tokens—e.g., all digits `1 became` IntTok(1) `; quoted` k1/k2 `became` StrTok("k1") `,` StrTok("k2")`, with surrounding quotes as SQuote.

The lexer does not understand the structure of the code, it doesnt know anything, it just groups characters into meaningful chunks for the parser to make sense of it.

Parsing

- `Source Text → [Lexing] → `**`[Parsing]`**` → [Semantic Analysis] → [Optimization] → [Code Generation]`
- **Input**: A list of lexical tokens
- **Output**: A parse tree
- **Method**:
- Fails when the input cannot be parsed by grammar rule

Example

```
1 + 2 * 3
```

↓ Lexing ↓

```
[NUMBER:1] [PLUS:+] [NUMBER:2] [STAR:*] [NUMBER:3]
```

↓ Parsing ↓
Takes the tokens and builds a tree that represents the structure (syntax) of the expression, according to grammar rules

```
     (+)
    /   \
 (1)    (*)
        / \
      (2) (3)
```

This is called an Abstract Syntax Tree (AST).

The root is `+`
The left child is `1`
The right child is `*`, which has children `2` and `3`

Continued from Lexer Example (JSON)

JSON Enum for Parser

```
enum Json {
  case IntLit(v:Int)
  case StrLit(v:String)
  case JsonList(vs: List[Json])
  case JsonObject(flds: Map[String, Json])
}
```

After passing the lexer output into the parser, the parser will output the following:

```
// Initial pre-lexer input
{'k1':1,'k2':[]}

val expected = Some(JsonObject(
  Map(
    "k1" -> IntLit(1),
    "k2" -> JsonList(Nil)
  )
))
```

`Some(...)` → the parser succeeded (it returns `Option[Json]`).

If parsing failed, you'd get None.

`JsonObject( Map( ... ) )` → the root AST node is an object with fields stored in a Scala Map[String, Json].

`"k1" -> IntLit(1)` → field k1 has an integer literal value 1.

`"k2" -> JsonList(Nil)` → field k2 has a list value that is empty (Nil means empty list), i.e. [].

Deep Dive and How it Works

A **dispatcher** for the parser decides which grammar rule to use by peeking at the first token (the 'lookahead')

```
def parse(toks:List[LToken]):Option[Json] = toks match {
  case Nil => // Done? what to return?
  case (t::ts) if t is digit => {
    val i = parse_an_int(toks); Some(IntLit(i)) }
  case (t::ts) if t is '\'' => {
    val s = parse_a_str(toks); Some(StrLit(s)) }
  case (t::ts) if t is '[' => {
    val l = parse_a_list(toks); Some(JsonList(l)) }
  case (t::ts) => {
    val m = parse_a_map(toks); Some(JsonObject(m)) }
}
```

The Parser receives the following output from lexer:

```
LBrace, SQuote, StrTok("k1"), SQuote, Colon, IntTok(1),
Comma, SQuote, StrTok("k2"), SQuote, Colon, LBracket, RBracket, RBrace
```

Then it looks at the first token `LBrace` → it falls into the object/map branch → use `parse_a_map`

In other words, the parser knows that "this is a JSON object starting now"

1. Dispatch: LBrace tells the parser to call the object routine (e.g., parse_a_map).
2. Consume & discard: the parser consumes the LBrace token (moves the cursor forward). The brace itself is not kept in the AST.
3. Parse contents: inside, it parses one or more "key" : value pairs, separated by commas.
4. Close: it expects and consumes the matching RBrace.
5. Build AST: it returns a `JsonObject(Map[String, Json])` built from those pairs.

```
val m = parse_a_map(toks)
Some(JsonObject(m))
```

FIRST TOKEN PARSED

- Parse first pair key "k1"

Expect `'s'` : consume `SQuote` , `StrTok("k1"), SQuote` → key = `"k1"` .

- Parse colon

Consume Colon.

.

.

.

- Close object

Lookahead: RBrace → consume it; object ends

Top Down Parsing

back-to-top

Big Picture:

- Solves grammar ambiguity and left recursion

Grammar Ambiguity → what does the 2 E mean?

```
E ::= E + E | E * E | i

E, T, F are Non-terminal expression (will expand further)
+,-,id,num etc are terminal (literal token from the lexer)

1. Flatten
E ::= E + E
E ::= E * E
E ::= i

2. Resolve Ambiguity by introducing more non-terminals
E ::= T + E
E ::= T
T ::= T * F
T ::= F
F ::= i
```

Left Recursion

```
...
T ::= T * F //will keep looping
...

1. Resolve left recursion

----------------------------------------------------------------
Formula:
A ::= A α | β

turns into

A  ::= β A'
A' ::= α A' | ε

A ↔ T
α ↔ * F (everything after the leading T in the left-recursive alt)
β ↔ F (the alternative that doesn’t start with T)
----------------------------------------------------------------


...
T  ::= F T'
T' ::= * F T' | ε
...
```

What about the rest? (FIRST-FIRST Conflict)

```
After resolving the left recursion, you have this:
E  ::=  T + E
E  ::=  T
T  ::=  F T'
T' ::=  * F T' | ε
T  ::=  F
F  ::=  i

1. Remove the extra T::=F
E  ::=  T + E
E  ::=  T
T  ::=  F T'
T' ::=  * F T' | ε
F  ::=  i

2. Resolve FIRST-FIRST conflict --> E is T + E or E?
----------------------------------------------------------------
Formula:
A ::= α β1 | α β2 (E  ::=  T + E | T)

turns into

A  ::= α A'
A' ::= β1 | β2

A ⇔ E
α ⇔ T
β1 ⇔ +E
β2 ⇔ ε
----------------------------------------------------------------

Final Grammar:
E  ::=  T E'
E'  ::=  + E E' | ε
T  ::=  F T'
T' ::=  * F T' | ε
F  ::=  i
```

FIRST-FOLLOW Conflict

```
Initial Grammar
S ::= A a
A ::= a | ε


-----------------------------------------------------------------
A ::= α | ε
P ::= γ1 A γ2
// with  FIRST(α) ∩ FOLLOW(A) ≠ ∅   (conflict)

turns into

P ::= γ1 α γ2 | γ1 γ2


P ⇔ S
A ⇔ A
a ⇔ a
γ1 ⇔ ε (nothing before A)
γ2 ⇔ a (the terminal after A)
-----------------------------------------------------------------

Final Output
S ::= a a | a
```

LL(1) Predictive Parsing Table (Step-by-step)

Grammar (Clean - no left recursion):

```
E  → T E'
E' → + T E' | ε
T  → i
```

Key FIRST:

- FIRST(X) is the set of terminals that **can appear first** when X expands, it never includes nonterminals or later tokens in a sequence

- For FIRST(E) for e.g., the grammar is E → T E', so the **first one that appears** is T, therefore:

$$\text{FIRST(E) = FIRST(T) = \{i\}}$$

- For FIRST(E'), the grammar is E1 → + T E' | ε, so the **first ones that** appears is + and ε, therefore:

$$\text{FIRST(E') = + FIRST(E') = } \varepsilon$$

- For FIRST(T) the grammar is T → i, so the first one that appears is i, therefore:

$$\text{FIRST(T) = \{i\}}$$

Together:

```
FIRST(E) = {i}
FIRST(E') = {+, ε}
FIRST(T) = {i}
```

Key FOLLOW (highlights):

- FOLLOW(X) is the set of terminals that can immediately appear to the right of nonterminal X in some sentential form. plus `$` if A can reach the end of input
- `sentential` form is any string you can get by starting from the grammar's start symbol

- Initialise: FOLLOW(Start) = `{$}` → FOLLOW(E) = `{$}`

- For FOLLOW(E'), the grammar is E → T E', therefore:

$$\text{FOLLOW(E') = FOLLOW(E) = \{\$\}}$$

- For FOLLOW(T), there is 2 grammar
  - E' → + T E'and
  - E → T E',
    therefore:

$$\text{FOLLOW(T) = \{+, FOLLOW(E)\} = \{+, \$\}}$$

Together,

```
FOLLOW(E) = {$}
FOLLOW(E') = {$}
FOLLOW(T) = {+, $}
```

Predictive table

(rows = nonterminals, cols = lookahead):

Recall the grammar:

```
E  → T E'
E' → + T E' | ε
T  → i
```

First Follow Keys

```
FIRST(E) = {i}
FIRST(E') = {+, ε}
FIRST(T) = {i}
FOLLOW(E) = {$}
FOLLOW(E') = {$}
FOLLOW(T) = {+, $}
```

Rows are E, E', T
Columns are T, E', +, ε, i

Construction Rules:

1. For every terminal `a` in FIRST(α), put `A → α` in table cell M[A, a].
2. If ε ∈ FIRST(α), then for every terminal b in FOLLOW(A) (including $), put `A → α` into M[A, b]

Each table cell must get at most one production. If two productions compete for the same cell, the grammar is not LL(1) and needs rewriting (usually by left factoring or removing left recursion).

Constructing the table

1. FIRST(E) = i (α = E, a = i)
   Find which non-terminal → i (A → a)
   T → i, but we cannot put T → i into M[E,i],

E → T E' can
So we put E → T E' into M[E,i]

2. FIRST(E') = + (α = E', a = +)
Find which non-terminal → + (A → a)
E' → + T E'
So we put `E' → + T E'` into M[E',+] (M[A,a])

3. FIRST(E') = ε (α = E', a = ε)
Find which non-terminal → ε (A → a)
E' → ε
So we put `E' → ε` into M[E',ε] (M[A,a])

3. FIRST(T) = i (α = E, a = i)
Find which non-terminal → i (A → a)
T → i
So we put `T → i` into M[T,i]

Non-terminal	i	+	$
E	E → T E'		
E'		E' → + T E'	E' → ε
T	T → i		

Naive Top Down Parsing

Think "match the next symbol, recurse." If next grammar symbol is a terminal, check the next token; if it's a nonterminal, try each alternative until one succeeds; accept ε (empty) only when allowed.

Subjected to Left Recursion loops, excessive backtracking, limited lookahead, no error recovery

Predictive Top Down Parsing (better alternative)

- Builds a parse from the start symbol downward, using 1 lookahead token to predict exactly which production to apply—no backtracking.

Parsec

back-to-top

- Parser as a Monad: sequencing without boilerplate
- Wraps the function and give it `map` and `flatMap` , so you can chain steps in order :

1. Parse this
2. Then parse that
3. Combine results

Naive Example

```
// Grammar Rules
(1) T ::= xx //T can be 2 x tokens in a row
(2) T ::= yx //T can be a Y token followed by a X token

// These represents the possible tokens your parser can see
// Data Structure
enum LToken{
  case XTok
  case YTok
}

// Parse result types
enum T {
  case XX
  case YX
}
```

Naive Implementation

back-to-top

We will need one function for every gramma rule (or pattern) :

- One function to parse XX (two XTok in a row)
- One function to parse YX (YTok followed by XTok)

```
enum Result[A] {
  case Failed(msg:String)
  case Ok(v:A)
}

def item(toks:List[LToken]):Result[(LToken, List[Token])] = toks match {
  case Nil => Failed("item() is called with an empty input")
  case (t::ts) => Ok((t, ts))
  }
def sat(toks:List[LToken])(p:LToken => Boolean):Result[(LToken, List[Token])] = toks match {
  case Nil => Failed("sat() is called with an empty input")
  case (t::ts) if p(t) => Ok((t, ts))
  case (t::ts) => Failed("""sat() is called with an input that
  does not satisfy the input predicate.""")
  }
```

- `Sat` is called twice
- Alot boiler plate codes for `parseXX` and `parseYX`

Monad Option (Better)

back-to-top

refer to parsec.scala for output

Step 0: Monad Boilerplate

```scala
case class Parser[T, A](p: List[T] => Result[(A, List[T])]) {
  def map[B](f: A => B): Parser[T, B] = Parser { toks =>

    // Consume
    p(toks) match {
      case Failed(err)    => Failed(err)
      // f(a) turns A into B
      case Ok((a, toks1)) => Ok((f(a), toks1))
    }
  }

  def flatMap[B](f: A => Parser[T, B]): Parser[T, B] = Parser { toks =>
    p(toks) match {
      case Failed(err)    => Failed(err)
      case Ok((a, toks1)) => f(a).p(toks1)
    }
  }
}
```

Step 1: Define basic token parsers

```scala
def xTok: Parser[LToken, LToken] =
  Parser {
    case XTok :: rest => Ok((XTok, rest))
    case _            => Failed("Expected XTok")
  }

def yTok: Parser[LToken, LToken] =
  Parser {
    case YTok :: rest => Ok((YTok, rest))
    case _            => Failed("Expected YTok")
  }
```

Suppose xTok sees input [XTok, YTok]. It returns Ok((XTok, [YTok])). When flatMap chains to another parser, that next parser receives the remaining [YTok].

xTok succeeds only when the list starts with XTok; otherwise it fails. Same idea for yTok. These are the atomic building blocks.

Step 2: Compose for `xx` and `yx` using flatmap/map

This part "brute forces" functions that matches specific sequence of tokens, i.e. if the tokens match the expected pattern, the parser returns the corresponding result from your enum, if not it will just fail

```scala
// T ::= xx
val parseXX: Parser[LToken, T] =
  for {
    _ <- xTok
    _ <- xTok
  } yield T.XX

// T ::= yx
val parseYX: Parser[LToken, T] =
  for {
    _ <- yTok
    _ <- xTok
  } yield T.YX
```

Step 3: Combine all alternatives (not xx and yx)

```scala
def or[A](p1: Parser[LToken, A], p2: Parser[LToken, A]): Parser[LToken, A] =
  Parser { toks =>
    p1.p(toks) match {
      case Failed(_) => p2.p(toks)
      case ok       => ok
    }
  }

val parseT: Parser[LToken, T] = or(parseXX, parseYX)
```

parseXX fails right away if first token isn't XTok.
`or` then runs parseYX, which consumes YTok and XTok, leaving [XTok], and returns T.YX.

Dealing with Left Recursion

back-to-top

Solution : Combinators from Parsec Library

```
def optional[T, A](pa: Parser[T, A]): Parser[T, Either[Unit, A]] = {
  val p1: Parser[T, Either[Unit, A]] = for (a <- pa) yield (Right(a))
  val p2: Parser[T, Either[Unit, A]] = Parser(toks => Ok((Left(()), toks)))
  choice(p1)(p2)
}
```

Many1 and Many

```
def optional[T, A](pa: Parser[T, A]): Parser[T, Either[Unit, A]] = {
  val p1: Parser[T, Either[Unit, A]] = for (a <- pa) yield (Right(a))
  val p2: Parser[T, Either[Unit, A]] = Parser(toks => Ok((Left(()), toks)))
  choice(p1)(p2)
    }

// one or more
def many1[T, A](p: Parser[T, A]): Parser[T, List[A]] = for {
  a <- p
  as <- many(p)
    } yield (a :: as)

// zero or more
def many[T, A](p: Parser[T, A]): Parser[T, List[A]] = ...
```

Math Exp Example

back-to-top

Grammer

```
E::= T + E
E::= T
T::= T * F
T::= F
F::= i
```

The Parser

```
def parseExp: Parser[LToken, Exp] = choice(parsePlusExp)(parseTermExp)

def parsePlusExp: Parser[LToken, Exp] = for {
  t <- parseTerm
  plus <- parsePlusTok
  e <- parseExp
} yield PlusExp(t, e)
```

The problem: `T ::= T * F` the parser looks at the rule `T is T * F`, so it starts trying to parse a T. But the first thing on the right-hand side is another T, so it repeats the same rule immediately—`T is T * F`—and tries to parse a T again. That loop never consumes a token, so it never gets past the first symbol.

The solution: Edit the Grammar

```
E::= T + E
E::= T
///////////////////////////////////////
T ::= T * F // left recursive!! --> No tokens are ever matched and the parser loops forever
// le-grammar
T   ::= FT' // parses the first factor
T'  ::= *FT' // handles each additional * factor
T'  ::= epsilon // lets you stop when there are no more multiplications
///////////////////////////////////////
T ::= F
F ::= i
```

```
// T ::= FT'
def parseTermLE:Parser[LToken, TermLE] = for {
  f <- parseFactor
  tp <- parseTermP
} yield TermLE(f, tp)


def parseTermP:Parser[LToken, TermLEP] = for {
omt <- optional(parseMultTermP)
} yield { omt match {
  case Left(_) => Eps
  case Right(t) => t
}}
def parseMultTermP:Parser[LToken, TermLEP] = for {
asterix <- parseAsterixTok
  f <- parseFactor
  tp <- parseTermP
} yield MultTermLEP(f, tp)
```

Another Example

```
// Grammar
// X ::= XXa | Y
// Y ::= Yb | c

// Step 1 : Flatten
// X ::= XXa
// X ::= Y
// Y ::= Yb
// Y ::= c

// Step 2 : Eliminate Left Recursion
// X ::= XXa | Y
// Y ::= cY'
// Y' ::= bY' | ε

// Step 3 : Eliminate left recursion for X
// X  ::= YX'
// X' ::= XaX'
// X' ::= ε
// Y  ::= cY'
// Y' ::= bY'
// Y' ::= ε

// Left recursion is only when the leftmost symbol is the same as the non-terminal being definec
// i.e. X ::= Xa... is left recursive because it starts with X
// but X ::= YX is not left recursive because it starts with Y although it ends with X
```