Functional Programming

7 Function Chaining and Composition

"Patterns of Combination and Composition"

- In functional programming we observe similar or equal patterns for function combination and composition

- Important structures for on combination and composition are
  - Functors
  - Monads
  - Applicatives
  - Arrows
  - ...

from Catagory Theory

7 Function Chaining and Composition

Functors and Monads

Function chaining with for-comprehension

Function composition

Case study: Parser combinators

FUNCTOR

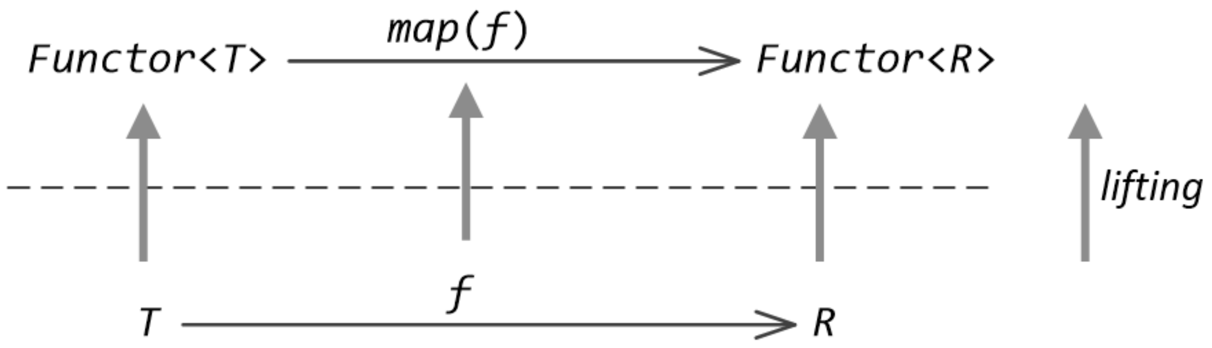
A *Functor* is a structure of generic elements

- a container with elements of generic type **T**
- a method **map** with a function parameter **f** and
  a result which again is of the same functor type with a different element type **R**

```
map : Functor[T] x (T => R) => Functor[R]
```

with a functor a function f is lifted to the level of a functor where

- function **f** is used for mapping elements
- at the level of the functor a **functor-specific** operation is realized by **map**



Functors

Functors already seen

- List[A]

```
trait List[+A] :
  def map[B](f: A => B): List[B] = ...
```

- Option[A]

```
sealed abstract class Option[+A] :
  def map[B](f: A => B): Option[B] = ...
```

- Try[A]

```
trait Try[+T] :
  def map[R](fn: T => R) : Try[R] = ...
```

- have generic type parameter for elements
- **map** maps to same functor

FUNCTOR LAWS

Identity law

functor.map(id) == functor

with ***id*** the identiy function: ***id***(***x***) = x

Composition law

functor.map(f ∘ g) == functor.map(f).map(g)

with ***f* ∘ *g*** is function composition: **(*g* ∘ *f*)(*x*) = *g*(*f*(*x*))**

From the two laws if follows that **map** is **structure-preserving**

➔ original functor and resulting functor have same structure

- **List**
  - resulting List has same number of elements and mapped elements are in same order
- **Option**
  - None mapped to None, Some mapped to Some

Monad

A Monad[A] is structure of elements of type A

- is a container of elements of generic type **A**
- it is a ***Functor*[A]** of elements of type **A**
- it has methods ***unit*** and ***flatMap*** as follows

  - **unit** takes value of some type **A** and wraps it into a **Monad**

```
unit : a -> Monad[A]
```

unit comes in different forms

  - **flatMap** takes a **Monad** with contained values of type **A**, a function which maps values of type **A** into a **Monad** with values of type **B** and returns a **Monad** with values of type **B**

```
flatMap : Monad[A] x (A -> Monad[B]) -> Monad[B]
```

Answer of GhatGPT to "What is a monad?"

In computer science, a monad is a design pattern that is used to structure and manage computations in a functional programming paradigm. It provides a way to encapsulate and sequence operations, while also allowing for composition and chaining of these operations.

At its core, a monad consists of three components:

1.*Type constructor*: It defines the type of the monad and represents a container or context for a value or computation. It is usually denoted as **M** or **Monad**.

2.*Return function* (unit or pure): It takes a value and lifts it into the monad, effectively encapsulating it within the monadic context. It has the signature **a -> M a**, where **a** is the type of the value.

3.*Bind function* (flatMap, >>=): It allows sequencing of operations within the monad. It takes a monadic value, applies a computation to it, and returns a new monadic value. The bind function has the signature **M a -> (a -> M b) -> M b**, where **a** and **b** are types and the second argument is a function that maps a value **a** to a new monadic value **M b**.

The monad pattern enables developers to write code in a sequential style, even when dealing with computations that involve side effects or complex data transformations. Monads provide a way to manage the flow of data and computations, handle error conditions, and control effects such as input/output or state changes.

Prominent examples of monads in functional programming languages include the Maybe monad (handles optional values and null safety), the List monad (represents a collection of values), and the IO monad (manages input/output operations). Each monad implementation defines its own return and bind functions specific to its behavior and requirements.

Monads have proven to be a powerful tool for structuring functional programs, providing a way to handle complexity, maintain code readability, and separate concerns in a modular fashion.

Example Monads

Monads already seen

	unit	flatMap
Iterable (List)	`Iterable(elem, ...)`	`def flatMap[B](f: A => Iterable[B]): Iterable[B]`
Option	`Some(x)`	`def flatMap[B](f: A => Option[B]): Option[B]`
Try	`Success(r)` `Failure(e)`	`def flatMap[U](f: T => Try[U]): Try[U]`

Monad List

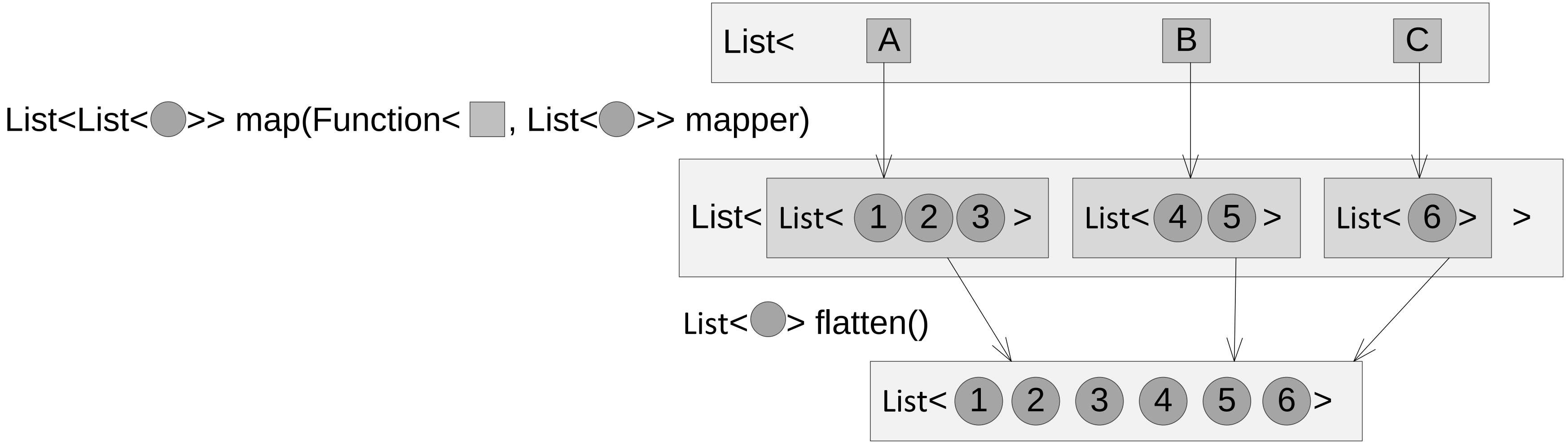
Illustration of flatMap for Lists

```
val lst = List("A", "B", "C")
```

```
val flatLst = lst.flatMap(c => {
  c match {
    case "A" => List(1, 2, 3)
    case "B" => List(4, 5)
    case "C" => List(6)
  }
})
```

equivalent

```
val listOfLists = lst.map(c => {
  c match {
    case "A" => List(1, 2, 3)
    case "B" => List(4, 5)
    case "C" => List(6)
  }
})
val flatList = listOfLists.flatten
```



Monad Laws

Identity law

of(x).flatMap(f) == f(x)

Associativity law

monad.flatMap(f).flatMap(g) == monad.flatMap(x -> f(x).flatMap(g))

flatMap(g) applied to result monad.flatMap(f)

flatMap(g) applied to result of f(x)

Associativity law: Illustration

Associativity law

```
monad.flatMap(x => f(x)).flatMap(y => g(y)) == monad.flatMap(x => f(x).flatMap(y => g(y)))
```

flatMap(g) applied to result monad.flatMap(f)

flatMap(g) applied to result of f(x)

Example:

- For 1 .. n, list of pairs of [(1, 1), (2, 1), (2, 2), ..., (n, 1), (n, 2), ... (n, n)]

does not compile

```
val pairs2 : Iterable[(Int, Int)] =
  (1 to n).flatMap(x => (1 to x)).flatMap(y => Iterable((x, y)))
```

x => f(x)

y => g(y)

x **not** in scope

```
val pairs : Iterable[(Int, Int)] =
  (1 to n).flatMap(x => (1 to x).flatMap(y => Iterable((x, y))))
```

y => g(y)

x => f(x).flatMap(y=> g(y))

x in scope

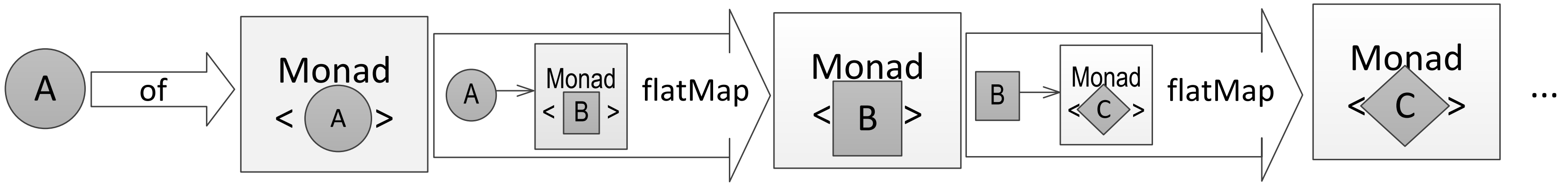
Monad is a Functor

Expressing map by flatMap

```
map : Monad[A] x (A -> B) -> Monad[B]
map(monad, fn) = flatMap(monad, a -> unit(fn(a)))
```

Relevance of Monads

Monads allow building chains of operations



- where the way how results are composed is specific to the Monad

There are two levels of computation

- mapping of elements by mapping functions
- composing results by flatMap (plus map, filter and and others)

Well Known Monads

- **Iterable**: all collections
- **Option**: with alternatives Some and None
- **Try**: for exception handling similar to Option but with exception object
- **Future**: for asynchronous computations, see Reactive Programming by H. Heinzelreiter
- **Stream**: for lazy collection processing
- **Reactive Streams**: see Reactive Programming by H. Heinzelreiter
- **Reader**: for providing a value for the computation chain, e.g., variable bindings
- **Writer**: for accumulating information
- **State**: for passing state information through computation chain
- **Parser**: functional parser combinators
- **Gen**: random value generators
- ...

In Java: **CompletableFuture**

Example: Evaluation of Expressions

with different Monads

- eval with **Id**
- eval with **Option**
- eval with **Try**
- eval with **Set**
- eval with **Future**

see Exercise 7

7 Function Chaining and Composition

For-Comprehension

for-expressions translated to chain of flatMap, map and withFilter

```scala
for (x <- e1) yield e2
```
⇒
```scala
e1.map(x => e2)
```

```scala
for (i <- 1 to 7) yield i * i
```
⇒
```scala
(1 to 7).map(i => i * i)
```

```scala
for (x <- e1 if c;
     s) yield e2
```
⇒
```scala
for (x <- e1.withFilter(c);
     s) yield e2
```

```scala
for (i <- 1 to 7 if i % 2 == 0)
    yield i * i
```
⇒
```scala
for (i <- (1 to 7).withFilter(i => i % 2 == 0))
    yield i * i
```

with continuation of translation

```scala
(1 to 7).withFilter(i => i % 2 == 0).map(i => i * i)
```

```scala
for ( x <- e1;
      y <- e2; s
    ) yield e3
```
⇒
```scala
e1.flatMap(x =>
             for (y <- e2; s
) yield e3)
```

```scala
for ( i <- 1 to 7;
      j <- 1 to i
    ) yield (i, j)
```
⇒
```scala
(1 to 7).flatMap(i =>
  for (j <- (1 to i)
yield (i, j))
```

```scala
(1 to 7).flatMap(i =>
   (1 to i).map(j =>
     (i, j))
```

For-Comprehension

Examples:

- **List** Monad:

```
List(1, 2, 3, 4).withFilter(i => i % 2 == 0)
                      .flatMap(j => (1 to i))
                          .map(j => (i, j))
```

with **for**:

```
 for {
    i <- List(1, 2, 3, 4) if i % 2 == 0;
    j <- 1 to i
 } yield (i, j)
```

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

- **Option** Monad:

```
val roomOpt = stds.get("Hans").
                  flatMap(id => courses.get(id)).
                    flatMap(course => rooms.get(course)).
                      filter(room.startsWith("HS")
```

```
val ssnToPerson : Map[Long, Person] = …
val pers2StdId : Map[Person, Long] = …
val stdId2Course : Map[Long, Course] = …
val course2Room : Map[Course, Room] = …
```

with **for**:

```
val roomOpt =
  for (
    id <- stds.get("Hans");
    course <- courses.get(id);
    room <- rooms.get(course) if room.startsWith("HS")
  ) yield room
```

```
Some(HS18)
```

Example: Evaluation of Expressions

eval with Double

```
def eval(expr: Expr, bds: Map[String, Double]): Double =
  expr match {
    case Lit(v) => v
    case Var(n) => bds(n)

    case Add(l, r) => {
      val lr = eval(l, bds)
      val rr = eval(r, bds)
      lr + rr
    }
    case Mult(l, r) => {
      val lr = eval(l, bds)
      val rr = eval(r, bds)
      lr * rr
    }
    case Min(s) => {
      val sr = eval(s, bds)
      -sr
    }
    case Rec(s) => {
      val sr = eval(s, bds)
      1.0 / sr
    }
  }
```

eval with Option[Double] and for

```
def eval(expr: Expr, bds: Map[String, Double]): Option[Double] =
  expr match {
    case Lit(v) => Some(v)
    case Var(n) => bds.get(n)

    case Add(l, r) => for {
        lr <- eval(l, bds)
        rr <- eval(r, bds)
      } yield lr + rr

    case Mult(l, r) => for {
        lr <- eval(l, bds)
        rr <- eval(r, bds)
      } yield lr * rr

    case Min(s) => for {
        sr <- eval(s, bds)
     } yield -sr

    case Rec(s) => for {
        sr <- eval(s, bds)
        rec <- if sr == 0.0 then None else Some(1.0 / sr)
      } yield rec
  }
```

7 FUNCTION CHAINING AND COMPOSITION

JKU

Function composition

- **Function composition operators** are **functions** which **take functions as arguments** and **return functions as results**
- **Function composition** allows creating **more complex functions** from **simpler functions**
- **Function composition** allows **creation of programs** by **programs**

Functions are programs!

Composition methods in Function1

- trait **Function1** with composition methods
  - **combine** and
  - and **andThen**
- which combine **this function object** with as **second function object** provided as parameter **g**

simplified

```scala
trait Function1[-T1, +R] extends AnyRef { self =>
  def apply(v1: T1): R

  def compose[A](g: A => T1): A => R = { x => this.apply(g(x)) }
  def andThen[A](g: R => A): T1 => A = { x => g(this.apply(x)) }
}
```

combine and **andThen** differ in the order of application

```scala
val wordsFn : String => List[String] =
  line => line.split(" ").toList

val sortFn : List[String] => List[String] =
  words => words.sorted

val wordsSortFn : String => List[String] =
  wordsFn.andThen(sortFn)

println(wordsSortFn("functional programming is great"))
```

```
List(functional, great, is, programming)
```

Scala Language Feature: Extension Methods

- Extension methods allow defining methods for a type external to the type definition
- Can be called like normal methods for objects of type

Example: **toInt**, **toDouble**, ... for class **String**

called object

extended type

```
extension (s: String)
  def toInt : Int = Integer.parseInt(s)
  def toLong : Long = java.lang.Long.parseLong(s)
  def toDouble : Double = java.lang.Double.parseDouble(s)
```

extension methods

```
"123".toInt
"0.5".toDouble
```

Scala Language Feature: Extension Methods

- Extension methods for any type supported, also generic types

Example: Extension methods for generic List[T]

```
object ListOps :
  extension[T](l : List[T])
    def isLong : Boolean = l.size > 10
    def isShort : Boolean = l.size <= 10
```

```
import ListOps.*
val lst = List(1, 2, 3, 4, ... )

if (lst.isLong) then ...
```

Composing Boolean Predicates

Boolean predicates are functions of type

```
A => Boolean
```

We write composition operators for combining predicates

- as extension functions for type **`A => Boolean`**

```
type Predicate[-T] = T => Boolean
extension[A] (p: Predicate[A])
  def &&(p2: Predicate[A]) : Predicate[A] = a => p(a) && p2(a)
  def ||(p2: Predicate[A]) : Predicate[A] = a => p(a) || p2(a)
  def not : Predicate[A] = a => !p(a)
```

```
def containsFn[A](a: A) : Predicate[List[A]] =
  lst => lst.contains(a)
def isEmptyFn[A] : Predicate[List[A]] =
  lst => lst.isEmpty
def notEmptyAndContainsFn[A](a: A) : Predicate[List[A]] =
  isEmptyFn.not && containsFn(a).not

println(notEmptyAndNotContainsFn(7)(List(1, 2, 3, 4, 5, 6, 8)))
```

Trait Ordering

Trait Ordering for comparing elements

- Methods **on**, **orElse**, **orElseBy**, etc. combine **Ordering** objects

subtype of Java's **java.util.Comparator**

simplified

```
trait Ordering[T] extends Comparator[T] with PartialOrdering[T] with Serializable :
  def compare(x: T, y: T): Int

  override def lteq(x: T, y: T): Boolean = compare(x, y) <= 0
  override def gteq(x: T, y: T): Boolean = compare(x, y) >= 0
  override def lt(x: T, y: T): Boolean = compare(x, y) < 0
  override def gt(x: T, y: T): Boolean = compare(x, y) > 0

  override def reverse: Ordering[T] = new Ordering.Reverse[T](this)

  def on[U](f: U => T): Ordering[U] = new Ordering[U] {
    def compare(x: U, y: U) = this.compare(f(x), f(y))
  }
  def orElse(other: Ordering[T]): Ordering[T] = (x, y) => {
    val res1 = this.compare(x, y)
    if (res1 != 0) res1 else other.compare(x, y)
  }
  def orElseBy[S](f: T => S)(implicit ord: Ordering[S]): Ordering[T] = ...
  ...
```

Companion Object Ordering

Creating Ordering

```
oject Ordering {
  ...
  def by[T, S](f: T => S)(implicit ord: Ordering[S]): Ordering[T] = new Ordering[T] {
    def compare(x: T, y: T) = ord.compare(f(x), f(y))
    ...
  }
  ...
```

Trait Ordering

Example application

```
val orderByLastName : Ordering[Person] = Ordering.by(person => person.lastName)
val orderByFirstName : Ordering[Person] = Ordering.by(person => person.firstName)
val orderByBorn : Ordering[Person] = Ordering.by(person => person.born)

val personOrdering =
  orderByLastName
    .orElse(orderByFirstName)
    .orElse(orderByBorn)

val personOrderingRev : Ordering[Person] =
  Ordering.by[Person, String](person => person.lastName)
    .orElseBy(_.firstName)
    .orElseBy(_.born)
    .reverse

val huberFranz1999 = Person("Franz", "Huber", 1990)
val huberFranz1998 = Person("Franz", "Huber", 1991)

println(personOrdering.lt(huberFranz1998, huberFranz1999))
println(personOrderingRev.lt(huberFranz1998, huberFranz1999))
```

7 Function Chaining and Composition

Functors and Monads

Function chaining with for-comprehension

Function composition

Case study: Parser combinators

Parser Combinators

Prominent example for functional programming

- Parser as function objects
  - reads input
  - returns parse result and rest of input

```
type Parser[T] : Input => Result[T]
```

- creating parser by composing simpler parsers
  - sequence: ~
  - alternative: |
  - option: **opt**
  - repetition: **rep**

Combinators analogous to EBNF

: Parser[(Char, Char)]

: Parser[Char]

```
char('x') ~ ( char('y') | (char('z') )
```

: Parser[Char]

: Parser[Char]

: Parser[Char]

<u>EBNF:</u>

'x' ('y' | 'z')

Parser Combinators: Example Implementation

- **Parsers**
  - Input = **String**
  - Parser of characters and words
  - returns **`Result[T]`**
    - either **`Success[T]`** with result of type **`T`** and rest of input
    - or **`Failure`** with error message and input causing failure

```
trait Result[+T] :
  val rest: String
case class Success[+T](result: T, rest: String) extends Result[T]
case class Failure(message: String, rest: String) extends Result[Nothing]
```

```
trait Parser[+T] extends (String => Result[T]) :
  def apply(input: String) : Result[T] // inherited
```

Parser are functions
String => Result[T]

apply method for doing parsing

Parser Combinators: Example Implementation

Parser of a single character

- **char**: returns a parser object for parsing a given character

```
def char(c: Char): Parser[Char] =
    input =>
        if (input.isEmpty) Failure("Input empty", input)
        else if (c != input.charAt(0)) Failure(s"Char $c expected", input)
        else Success(c, input.substring(1))
```

```
val xParser: Parser[Char] = char('x')
val yParser: Parser[Char] = char('y')
val zParser: Parser[Char] = char('z')

val xR: Result[Char] = xParser("xyz")
println(xR)
```

```
Success(x,yz)
```

```
val yR: Result[Char] = yParser(xR.rest)
println(yR)
```

```
Success(y,z)
```

```
val zR: Result[Char] = zParser(yR.rest)
println(zR)
```

```
Success(z,)
```

```
val xFailed: Result[Char] = xParser("abc")

println(xFailed)
```

```
Failure(Input does not start with x,abc)
```

```
val yFailed: Result[Char] = yParser("")

println(yFailed)
```

```
Failure(Input empty,)
```

Parser Combinators: Example Implementation

Monad Parsr:

- **flatMap**: Applying parsers in series

```
trait Parser[+T] extends (String => Result[T]) :
  thisParser: Parser[T] =>

  def flatMap[R](f: T => Parser[R]): Parser[R] =
      input =>
        thisParser(input) match {
          case Success(t, rest) => f(t)(rest)
          case f@Failure(_, _) => f
        }
```

alias to this

first apply this parser

if success then apply second parser

otherwise return failure

- **unit** methods in companion object
  - **success:** creating parser with **Success** with value and same input
  - **failure:** creating parser with **Failure** with message and same input

```
object Parser {
  def success[T](t: T) : Parser[T] =
      input => Success(t, input)
  def failure[T](message: String): Parser[T] =
      input => Failure(message, input)
}
```

Parser Combinators: Example Implementation

Parser combinator methods:

- **map**: flatMap with success of mapped value

```
trait Parser[+T] extends (String => Result[T]) :
  thisParser: Parser[T] =>
  …
  def map[R](f: T => R): Parser[R] =
    thisParser.flatMap(t => success(f(t)))
```

- **filter:** flatMap with success if predicate fulfilled

```
  def filter(pred: T => Boolean): Parser[T] =
    thisParser.flatMap(t => if pred(t) then success(t)
                            else failure("No such element"))
```

```
val anyChar : Parser[Char] =
  input => if (input.isEmpty) Failure("Input empty", input)
           else Success(input.charAt(0), input.substring(1))

def char(c: Char): Parser[Char] = anyChar filter { a => a == c}
```

```
val xParser: Parser[Char] = char('x')
```

Parser Combinators: Example Implementation

Parser combinator methods:

- **|**: operator for alternatives

```
trait Parser[+T] extends (String => Result[T]) :
  thisParser: Parser[T] =>
  …
  def |[U >: T](otherParser: => Parser[U]): Parser[U] =
  input => thisParser(input) match {
      case s@Success(r, rest) => s
      case Failure(_, _) => otherParser(input)
    }
```

first apply this parser

if this fails
apply other parser

```
val u_Or_vParser : Parser[Char] = char('u') | char('v')

val u_Or_vResult = u_Or_vParser("ux")
println(u_Or_vResult)

val u_Or_vResult2 = u_Or_vParser("vx")
println(u_Or_vResult2)
```

```
Success(u,x)
```

```
Success(v,x)
```

Parser Combinators: Example Implementation

Parser combinator methods:

- **opt**: optional parser with Option as result

```
trait Parser[+T] extends (String => Result[T]) :
  thisParser: Parser[T] =>
  …
  def opt: Parser[Option[T]] =
    thisParser.map(t => Some(t)) | success(None)
```

with result **Option[T]**

if this parser fails then result with **None**

```
val opt_u_Parser : Parser[Option[Char]] = char('u').opt

val opt_u_Result = opt_u_Parser("ux")
println(opt_u_Result)    //                              Success(Some(u),x)

val opt_u_Result2 = opt_u_Parser ("x")
println(opt_u_Result2)   //                              Success(None,x)
```

Parser combinator methods:

- **rep**: repetition of this parser with List as result

```
trait Parser[+T] extends (String => Result[T]) :
  thisParser: Parser[T] =>
  …
  def rep: Parser[List[T]] =
    new Parser[List[T]] {
      repParser: Parser[List[T]] =>
      override def apply(input: String): Result[List[T]] =
        thisParser(input) match {
          case Failure(_, _) => Success(List(), input)
          case s@Success(r, rest) =>
            repParser(rest) match {
              case Success(rs, rest2) => Success(r :: rs, rest2)
              case Failure(_, rest2) => Success(List(), rest2)
            }
        }
    }
```

if this parser fails then result empty list

recursive call of the repetition parser

```
val xRepParser: Parser[List[Char]] = char('x').rep
val xRepResult = xRepParser("xxxxv")
println(xRepResult)
```

```
Success(List(x, x, x, x),v)
```

```
val vRepParser: Parser[List[Char]] = char('v').rep
val vRepResult = xRepParser("xxxxv")
println(vRepResult)
```

```
Success(List(), xxxxv)
```

Parser Combinators: Example Implementation

Example application

```
val u_Or_vxyParser = (char('u') | char('v')) ~ char('x').opt ~ char('y')
val u_Or_vxyResult = u_Or_vxyParser("uxy")
println(u_Or_vxyResult)
```

```
Success(((u,Some(x)),y),)
```

```
val u_Or_vxyyyyParser = (char('u') | char('v')) ~ char('x').opt ~ char('y').rep ~
char('z')
val u_Or_vxyyyyResult = u_Or_vxyyyyParser("vxyyyyz")
println(u_Or_vxyyyyResult)
```

```
Success((((v,Some(x)),List(y, y, y, y)),z),)
```

Parser Combinators: Example Token Parser

- Parser for elements from list

```
def oneOutOf[E](pe: Parser[E], outOf: List[E]): Parser[E] =
  pe.filter(r => outOf.contains(r))
def charOutOf(cs: List[Char]) = oneOutOf(any, cs)
```

- Parser for letters

```
val letter = charOutOf("abcdefghijklmnopqrstuvwxyz".toList)
val digit = charOutOf("0123456789".toList)
```

- Parser for words

```
val anyWord: Parser[String] = letter.rep.map(chars => new String(chars.toArray))
def word(w: String) : Parser[String] = anyWord.filter(t => t == w)
```

- Parser of Booleans

```
val trueParser : Parser[Boolean] = word("true").map(t => true)
val falseParser : Parser[Boolean] = word("false").map(f => false)
val bool : Parser[Boolean] = trueParser | falseParser
```

```
val s1 = bool("true false")
println(s1)
val s2 = bool(s1.rest)
println(s2)
```

```
Success(true, false)
```

```
Success(false,)
```

Parser Combinators: Example Token Parser

- Parser for digit

```
val digit = charOutOf("0123456789".toList)
```

- Parser for digits and ints

```
val digits = digit.rep.map(ds => new String(ds.toArray))

val int: Parser[Int] = digits.map { ds => ds.toInt }
```

```
val s3 = int("123")
println(s3)
```

```
Success(123,)
```

SCALA LIBRARY SCALA.UTIL.PARSING.COMBINATOR

Combinators:

Sequence:	**a ~ b**	: Parser[A ~ B]
Sequence with left result:	**r <~ c**	: Parser[R]
Sequence with right result :	**c ~> r**	: Parser[R]
Option:	**opt(a)**	: Parser[Option[A]]
Repitition:	**rep(a)**	: Parser[List[A]]
Repitition with separator:	**repsep(a, separator)**	: Parser[List[A]]
mapping (= semantic action):	**^^ (T => U)**	: Parser[U]

Scala Library scala.util.parsing.combinator

Predefined parsers

- RegexParsers: Regular expressions

```
trait RegexParsers extends scala.util.parsing.combinator.Parsers {
  implicit def literal(s : String): Parser[String]
  implicit def regex(r : Regex) : Parser[String]
}
```

Implicit conversions from strings to parsers

- JavaTokenParsers: Java identifiers und literals

```
trait JavaTokenParsers extends RegexParsers {
  def ident : Parser[String]
  def wholeNumber : Parser[String]
  def decimalNumber : Parser[String]
  def stringLiteral : Parser[String]
  def floatingPointNumber : Parser[String]
}
```

Scala Language Feature: Implicit Conversions

Implicitly creates an object of one type into an object of other type

- if method not applicable for object
- look for implicit conversion method so that method gets applicable on converted object
- apply conversion and apply method

Scala 2 approach

- implicit method

```
case class ParsableString(s: String) :
  def parseInt = Integer.parseInt(s)
  def parseDouble = java.lang.Double.parseDouble(s)

implicit def stringToParsable(s: String) : ParsableString = ParsableString(s)
```

```
val i : Int = "123".parseInt
```

mostly replaced by extension method in Scala 3

Scala 3 approach

- Conversion function

```
abstract class Conversion[-T, +U] extends Function1[T, U]:
  def apply(x: T): Ung = ParsableString(s)
```

```
given stringToParsableFn : Conversion[String, ParsableString] =
  s => ParsableString(s)
```

```
val i : Int = "123".parseInt
```

JKU

Scala Library scala.util.parsing.combinator

- Application example: JSON parser

```
import java.io.FileReader
import scala.util.parsing.combinator.JavaTokenParsers

object JSONParser extends JavaTokenParsers {
  def obj: Parser[List[String ~ String ~ Any]] = "{"~> repsep(member, ",") <~"}"
  def member: Parser[String ~ String ~ Any] = stringLiteral~":"~value
  def value: Parser[Any] = obj | arr | stringLiteral |  floatingPointNumber | "null" | "true" | "false"
  def arr: Parser[List[Any]] = "["~> repsep(value, ",") <~"]"
}
```

```
val reader = new FileReader("address-book.json")
val parseResult : ParseResult[Any] = parseAll(value, reader)
println(parseResult)
```

Input:

```
{
  "address book": {
    "name": "John Smith",
    "address": {
      "street": "10 Market Street",
      "city"  : "San Francisco, CA",
      "zip"   : 94111
    },
    "phone numbers": [
      "408 338-4238",
      "408 111-6892"
    ]
  }
}
```

Parse result:

```
List((("address book"~:)~List((("name"~:)~"John Smith"),
(("address"~:)~List((("street"~:)~"10 Market Street"), (("city"~:)~"San Francisco, CA"),
(("zip"~:)~94111))), (("phone numbers"~:)~List("408 338-4238", "408 111-6892")))))
```

Scala Library scala.util.parsing.combinator

- Application example: JSON parser
  - Mapping of initial parse result

```
object JSONParser2 extends JavaTokenParsers {
  def obj: Parser[Map[String, Any]] =
    ("{"~> repsep(member, ",") <~"}").map(ms => (Map() ++ ms))
  def member: Parser[(String, Any)] =
    (stringLiteral ~ ":" ~ value).map(nameValue => nameValue match {
                                  case name~":"~value => (name, value)
                                 })
  def value: Parser[Any] =
      obj |
      arr |
      stringLiteral |
      floatingPointNumber.map(_.toDouble) |
      "null".map(x => null) |
      "true".map(x => true) |
      "false".map(x => false)
  def arr: Parser[List[Any]] =
    ("["~> repsep(value, ",") <~"]").map(l => l)
}
```

```
Map(
  "address book" -> Map(
    "name" -> "John Smith",
    "address" -> Map(
       "street" -> "10 Market Street",
       "city" -> "San Francisco, CA",
       "zip" -> 94111.0),
    "phone numbers" ->
       List(
          "408 338-4238",
          "408 111-6892"
       )
    )
)
```

Scala Library scala.util.parsing.combinator

- Application example: JSON parser
  - Semantic actions with ^^

```
object JSONParserSemActions extends JavaTokenParsers {

  def obj: Parser[Map[String, Any]] = "{"~> repsep(member, ",") <~"}" ^^ (Map() ++ _)

  def member: Parser[(String, Any)] = stringLiteral~":"~value ^^ { case name~":"~value => (name, value) }

  def value: Parser[Any] =
        obj |
        arr |
        stringLiteral |
        floatingPointNumber ^^ (_.toDouble) |
        "null"              ^^ (x => null) |
        "true"              ^^ (x => true) |
        "false"             ^^ (x => false)

  def arr: Parser[List[Any]] = "["~> repsep(value, ",") <~"]" ^^ (List() ++
```

```
Map(
  "address book" -> Map(
    "name" -> "John Smith",
    "address" -> Map(
       "street" -> "10 Market Street",
       "city" -> "San Francisco, CA",
       "zip" -> 94111.0),
    "phone numbers" ->
       List(
          "408 338-4238",
          "408 111-6892"
       )
    )
)
```