Parsing with Combinators

Functional Programming (CS-210)

EPFL

Parsing

People write code using ***text*** (sequences of characters).

```
"""
(def double(n) = (n + n)
 double 4)
"""
```

Parsing

People write code using ***text*** (sequences of characters).

```
"""
(def double(n) = (n + n)
 double 4)
"""
```

But writing an interpreters (or compilers) is way easier on ***trees***.

```
Defs(
  List(("double", Fun("n", BinOp(Plus, N("n"), N("n"))))),
  Call(N("double"), C(4)))
```

Parsing

People write code using ***text*** (sequences of characters).

```
"""
(def double(n) = (n + n)
 double 4)
"""
```

But writing an interpreters (or compilers) is way easier on ***trees***.

```
Defs(
  List(("double", Fun("n", BinOp(Plus, N("n"), N("n"))))),
  Call(N("double"), C(4)))
```

This representation immediately exposes the structure of the code, while the text representation does not.

Parsing

Therefore, in such projects somebody has to write a conversion from text to trees.

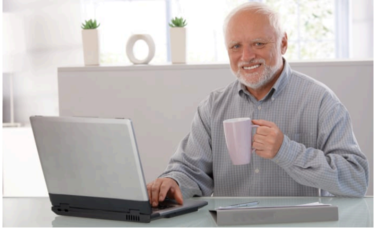
```
def parse(input: List[Char]): Expr
```

Parsing

Therefore, in such projects somebody has to write a conversion from text to trees.

```
def parse(input: List[Char]): Expr
```

If you take ***Computer Language Processing (CS-320)***, this person will be you!



Parsing

Writing such functions can be very tricky. ***Parser Combinators*** are one way to go about handling this complexity.

Parsing

Writing such functions can be very tricky. ***Parser Combinators*** are one way to go about handling this complexity.

Simple idea:

- Very simple basic parsers
- Ways to combine parsers into more complex parsers

Parser Combinator Libraries

There exist many parser combinator libraries in Scala:

Parser Combinator Libraries

There exist many parser combinator libraries in Scala:

- Scala Parser Combinators
  `github.com/scala/scala-parser-combinators`

Parser Combinator Libraries

There exist many parser combinator libraries in Scala:

- Scala Parser Combinators
  `github.com/scala/scala-parser-combinators`
- FastParse
  `www.lihaoyi.com/fastparse/`

Parser Combinator Libraries

There exist many parser combinator libraries in Scala:

- Scala Parser Combinators
  `github.com/scala/scala-parser-combinators`
- FastParse
  `www.lihaoyi.com/fastparse/`
- Scallion
  `github.com/epfl-lara/scallion`

Parser Combinator Libraries

There exist many parser combinator libraries in Scala:

- Scala Parser Combinators
  `github.com/scala/scala-parser-combinators`
- FastParse
  `www.lihaoyi.com/fastparse/`
- Scallion
  `github.com/epfl-lara/scallion`

What I will present is the ***general idea*** behind many such libraries.

Parser Combinator Libraries

There exist many parser combinator libraries in Scala:

- Scala Parser Combinators
  `github.com/scala/scala-parser-combinators`
- FastParse
  `www.lihaoyi.com/fastparse/`
- Scallion
  `github.com/epfl-lara/scallion`

What I will present is the ***general idea*** behind many such libraries.
The actual implementation may vary but the basic interface will often remain the same.

`Parser` objects

```
def parse(input: List[Char]): Expr
```

`Parser` objects

```scala
def parse(input: List[Char]): Expr
```

Turning it into a class:

```scala
case class Parser(parse: List[Char] => Expr)
```

Parser objects

```
def parse(input: List[Char]): Expr
```

Turning it into a class:

```
case class Parser(parse: List[Char] => Expr)
```

Returning the remaining input:

```
case class Parser(parse: List[Char] => (Expr, List[Char]))
```

`Parser` objects

```
def parse(input: List[Char]): Expr
```

Turning it into a class:

```
case class Parser(parse: List[Char] => Expr)
```

Returning the remaining input:

```
case class Parser(parse: List[Char] => (Expr, List[Char]))
```

Returning multiple alternatives:

```
case class Parser(parse: List[Char] => LazyList[(Expr, List[Char])])
```

Parser objects

```scala
def parse(input: List[Char]): Expr
```

Turning it into a class:

```scala
case class Parser(parse: List[Char] => Expr)
```

Returning the remaining input:

```scala
case class Parser(parse: List[Char] => (Expr, List[Char]))
```

Returning multiple alternatives:

```scala
case class Parser(parse: List[Char] => LazyList[(Expr, List[Char])])
```

Abstracting over the type of trees:

```scala
case class Parser[+A](parse: List[Char] => LazyList[(A, List[Char])])
```

Parser objects

```scala
// Method of Parser[+A]
def apply(input: List[Char]): Option[A] = {
  this
    .parse(input)
    .filter(_._2.isEmpty)
    .map(_._1)
    .headOption
}
```

Parser objects

```
def parse(input: List[Char]): Tree = {
  val parser: Parser[Tree] = ???

  parser(input).getOrElse{ throw new ParseError() }
}
```

Parser objects

```
def parse(input: List[Char]): Tree = {
  val parser: Parser[Tree] = ???  // How to build this?

  parser(input).getOrElse{ throw new ParseError() }
}
```

Example: Parser for sums

```scala
val letter: Parser[Char] = elem(_.isLetter)

val variable: Parser[SumExpr] = letter.map(Var(_))

val digit: Parser[Char] = elem(_.isDigit)

val number: Parser[SumExpr] =
  many(digit)
    .filter(_.length > 0)
    .map(ds => Num(BigInt(ds.mkString(""))))

val atom: Parser[SumExpr] = number | variable

val plus: Parser[Char] = elem('+')

val sum: Parser[SumExpr] =
  (atom ~ many(plus ~> atom)).map {
    case n ~ ns => Sum(n +: ns)
  }
```

Building Parsers

Basic Parsers

Matching a single character:

```
val item: Parser[Char] =
  Parser(input => input match {
    case c :: cs => LazyList((c, cs))
    case _ => LazyList()
  })
```

Basic Parsers

Returning a value without looking at the input:

```
def success[A](value: A): Parser[A] =
  Parser(input => LazyList((value, input)))
```

Basic Parsers

Returning a value without looking at the input:

```
def success[A](value: A): Parser[A] =
  Parser(input => LazyList((value, input)))
```

Always failing:

```
val failure: Parser[Nothing] =
  Parser(input => LazyList())
```

Building Complex Parsers

Building Complex Parsers

Filtering Out Values

Filtering out unwanted values:

```
// Method of Parser[+A]
def filter(predicate: A => Boolean): Parser[A] =
  Parser(input => this.parse(input).filter {
    case (value, _) => predicate(value)
  })
```

Filtering Out Values

Filtering out unwanted values:

```scala
// Method of Parser[+A]
def filter(predicate: A => Boolean): Parser[A] =
  Parser(input => this.parse(input).filter {
    case (value, _) => predicate(value)
  })
```

```scala
def elem(predicate: Char => Boolean): Parser[Char] =
  item.filter(predicate)
```

Filtering Out Values

Filtering out unwanted values:

```
// Method of Parser[+A]
def filter(predicate: A => Boolean): Parser[A] =
  Parser(input => this.parse(input).filter {
    case (value, _) => predicate(value)
  })
```

```
def elem(predicate: Char => Boolean): Parser[Char] =
  item.filter(predicate)
```

```
def elem(char: Char): Parser[Char] =
  elem(_ == char)
```

Transforming Values

Modifying the parsed value:

```
// Method of Parser[+A]
def map[B](function: A => B): Parser[B] =
  Parser(input => this.parse(input).map {
    case (value, rest) => (function(value), rest)
  })
```

Example:

```
val variable: Parser[SumExpr] = letter.map(Var(_))
```

Sequencing Parsers

Sequencing parsers:

```
// Method of Parser[+A]
def ~[B](that: Parser[B]): Parser[(A, B)] =
  Parser(input =>
    for {
      (leftValue, leftRest)   <- this.parse(input)
      (rightValue, rightRest) <- that.parse(leftRest)
    } yield ((leftValue, rightValue), rightRest))
```

Sequencing Parsers

Parser combinator libraries generally introduce a bit of sugar...

```
// Method of Parser[+A]
def ~[B](that: Parser[B]): Parser[A ~ B] =
                                // ^^^^^
```

Instead of pairs, they will use something like:

```
case class ~[+A, +B](_1: A, _2: B)
```

Sequencing Parsers

Parser combinator libraries generally introduce a bit of sugar...

```
// Method of Parser[+A]
def ~[B](that: Parser[B]): Parser[A ~ B] =
                                  // ^^^^^
```

Instead of pairs, they will use something like:

```
case class ~[+A, +B](_1: A, _2: B)
```

Which is simply to provide better looking pattern matching:

```
val sum: Parser[SumExpr] =
  (atom ~ many(plus ~> atom)).map {
    case n ~ ns => Sum(n +: ns)
  }
```

Sequencing Parsers

Sometimes, we wish to only keep the value from one side of a sequence and ignore the other.

Sequencing Parsers

Sometimes, we wish to only keep the value from one side of a sequence and ignore the other.

```
// Methods of Parser[+A]
def <~(that: Parser[Any]): Parser[A] = (this ~ that).map {
  case left ~ _ => left
}
```

Sequencing Parsers

Sometimes, we wish to only keep the value from one side of a sequence and ignore the other.

```
// Methods of Parser[+A]
def <~(that: Parser[Any]): Parser[A] = (this ~ that).map {
  case left ~ _ => left
}

def ~>[B](that: Parser[B]): Parser[B] = (this ~ that).map {
  case _ ~ right => right
}
```

Introducing Alternatives

Specifying alternatives:

```
// Method of Parser[+A]
def |[B >: A](that: Parser[B]): Parser[B] =
  Parser(input => this.parse(input) #::: that.parse(input))
```

Introducing Alternatives

Specifying alternatives:

```
// Method of Parser[+A]
def |[B >: A](that: Parser[B]): Parser[B] =
  Parser(input => this.parse(input) #::: that.parse(input))
```

Example:

```
val atom: Parser[SumExpr] = number | variable
```

Optional Parsers

Making a parser optional:

```
// Method of Parser[+A]
def optional: Parser[Option[A]] =
  this.map(Some(_)) | success(None)
```

Using Recursion

How to parse sums with parentheses?

Using Recursion

How to parse sums with parentheses?

`"(1+2)+(x+y)"`

Using Recursion

How to parse sums with parentheses?

`"(1+2)+(x+y)"`

Using recursion!

Using Recursion

How to parse sums with parentheses?

`"(1+2)+(x+y)"`

Using recursion!

```
lazy val parensSum = elem('(') ~> sum <~ elem(')')

lazy val atom: Parser[SumExpr] = number | variable | parensSum

val plus: Parser[Char] = elem('+')

lazy val sum: Parser[SumExpr] = {
  (atom ~ many(plus ~> atom)).map {
    case n ~ ns => Sum(n +: ns)
  }
}
```

Using Recursion

How to parse sums with parentheses?

`"(1+2)+(x+y)"`

Using recursion!

```
lazy val parensSum = elem('(') ~> sum <~ elem(')')

lazy val atom: Parser[SumExpr] = number | variable | parensSum

val plus: Parser[Char] = elem('+')

lazy val sum: Parser[SumExpr] = defer { // < Change here!
  (atom ~ many(plus ~> atom)).map {
    case n ~ ns => Sum(n +: ns)
  }
}
```

Using Recursion

```scala
def defer[A](parser: => Parser[A]): Parser[A] = {
  lazy val cached: Parser[A] = parser
  Parser(cached.parse(_))
}
```

Repeating Parsers

Repeating a parser:

```
def many[A](parser: Parser[A]): Parser[List[A]] = {
  lazy val repeated: Parser[List[A]] = defer {
    (parser ~ repeated).map { case x ~ xs => x :: xs } |
      success(List())
  }

  repeated
}
```

Using Recursion

Some libraries don't have `defer`, and instead pass arguments ***by name*** instead of *by value* for the various combinators.

```
// Methods of Parser[+A]
def ~[B](that: => Parser[B]): Parser[A ~ B] = ...

def |[B >: A](that: => Parser[B]): Parser[B] = ...
```

Handling Spaces

Spaces everywhere!

```
val input =
  " (3 + 4 ) + x + y  "
// ^  ^ ^ ^ ^ ^ ^ ^ ^^
```

Handling Spaces

Spaces everywhere!

```
val input =
  " (3 + 4 ) + x + y   "
// ^  ^ ^ ^ ^ ^ ^ ^ ^^
```

One solution:

```
val space: Parser[Char] = elem(_.isWhitespace)
```

Handling Spaces

Spaces everywhere!

```
val input =
  " (3 + 4 ) + x + y  "
// ^  ^ ^ ^ ^ ^ ^ ^ ^^
```

One solution:

```
val space: Parser[Char] = elem(_.isWhitespace)

def token[A](parser: Parser[A]): Parser[A] = parser <~ many(space)
```

Handling Spaces

Spaces everywhere!

```scala
val input =
  " (3 + 4 ) + x + y  "
// ^  ^ ^ ^ ^ ^ ^ ^ ^^
```

One solution:

```scala
val space: Parser[Char] = elem(_.isWhitespace)

def token[A](parser: Parser[A]): Parser[A] = parser <~ many(space)

val variable: Parser[SumExpr] = token(letter.map(Var(_)))
```

Handling Spaces

Spaces everywhere!

```scala
val input =
  " (3 + 4 ) + x + y  "
// ^  ^ ^ ^ ^ ^ ^ ^ ^^
```

One solution:

```scala
val space: Parser[Char] = elem(_.isWhitespace)

def token[A](parser: Parser[A]): Parser[A] = parser <~ many(space)

val variable: Parser[SumExpr] = token(letter.map(Var(_)))

lazy val parser: Parser[SumExpr] = many(space) ~> sum
```

Lexing

Another solution is to write a ***lexer*** to handle spaces, comments and more!

```
def lex(input: List[Char]): List[Token] = ...
```

Lexing

Another solution is to write a ***lexer*** to handle spaces, comments and more!

```
def lex(input: List[Char]): List[Token] = ...
```

Then, the parser operates on sequences of ***tokens*** instead of `Char`s.

```
def parse(input: List[Token]): Expr = ...
```

Lexing

Another solution is to write a ***lexer*** to handle spaces, comments and more!

```
def lex(input: List[Char]): List[Token] = ...
```

Then, the parser operates on sequences of ***tokens*** instead of `Char`s.

```
def parse(input: List[Token]): Expr = ...
```

Some parser combinators libraries support both styles, and even provide ways to write lexers using the similar combinators.

Sequencing Revisited

But wait, there's more!

Sequencing Revisited

Let's go back to sequencing...

```
// Method of Parser[+A]
def ~[B](that: Parser[B]): Parser[(A, B)] =
  Parser(input =>
    for {
      (leftValue, leftRest)   <- this.parse(input)
      (rightValue, rightRest) <- that.parse(leftRest)

    } yield ((leftValue, rightValue), rightRest))
```

Sequencing Revisited

Let's go back to sequencing...

```
// Method of Parser[+A]
def ~[B](that: Parser[B]): Parser[(A, B)] =
  Parser(input =>
    for {
      (leftValue, leftRest)   <- this.parse(input)
      (rightValue, rightRest) <- that.parse(leftRest)
                                 // ^ leftValue is unused.
    } yield ((leftValue, rightValue), rightRest))
```

Sequencing Revisited

Let's go back to sequencing...

```scala
// Method of Parser[+A]
def ~[B](that: A => Parser[B]): Parser[B] =
  Parser(input =>
    for {
      (leftValue, leftRest)   <- this.parse(input)
      (rightValue, rightRest) <- that(leftValue).parse(leftRest)
                                 // ^ leftValue is passed to that.
    } yield (rightValue, rightRest))
```

Sequencing Revisited

Let's rename ~ to something you already know:

```scala
// Method of Parser[+A]
def flatMap[B](that: A => Parser[B]): Parser[B] =
  Parser(input =>
    for {
      (leftValue, leftRest)   <- this.parse(input)
      (rightValue, rightRest) <- that(leftValue).parse(leftRest)
                                 // ^ leftValue is passed to that.
    } yield (rightValue, rightRest))
```

Parser is a Monad!

```
def unit[A](x: A): Parser[A] = success(x)
```

`Parser` is a Monad!

```
def unit[A](x: A): Parser[A] = success(x)
```

Monad laws

Associativity `p.flatMap(f).flatMap(g) == p.flatMap(f(_).flatMap(g))`

Left unit `unit(x).flatMap(f) == f(x)`

Right unit `p.flatMap(unit(_)) == p`

For-notation for Parsers

Thanks to `Parser` being a Monad, you can write sequences of parsers using for-notation.

```
val ifExpr: Parser[Expr] =
  for {
    _ <- keyword("if")
    c <- expr
    _ <- keyword("then")
    t <- expr
    _ <- keyword("else")
    e <- expr
  } yield IfExpr(c, t, e)
```

For-notation for Parsers

Thanks to `Parser` being a Monad, you can write sequences of parsers using for-notation.

```
val ifExpr: Parser[Expr] =
  for {
    _ <- keyword("if")
    c <- expr
    _ <- keyword("then")
    t <- expr
    _ <- keyword("else")
    e <- expr
  } yield IfExpr(c, t, e)
```

```
val tagged: Parser[XML] =
  for {
    o <- openTag
    c <- contents
    _ <- closeTag(o.name)
  } yield Tagged(o, c)
```

Summary

- Parsing and why it is important.

Summary

- Parsing and why it is important.
- What parser combinators are.

Summary

- Parsing and why it is important.
- What parser combinators are.
- How the various combinators can be implemented.

Summary

- Parsing and why it is important.
- What parser combinators are.
- How the various combinators can be implemented.
- Ways to handle lexical analysis.

Summary

- Parsing and why it is important.
- What parser combinators are.
- How the various combinators can be implemented.
- Ways to handle lexical analysis.
- That `Parser` is a Monad!

Summary

- Parsing and why it is important.
- What parser combinators are.
- How the various combinators can be implemented.
- Ways to handle lexical analysis.
- That `Parser` is a Monad!

Take a look at the parser for the interpreter language!