Parser combinators in Scala

Bernie Pope

Outline

- The combinator style.
- From grammars to code.
- Types.
- Refining the output.
- Performance.

The combinator style

- Bricks: a small set of primitive methods.
- Cement: a small set of combinator methods.
- Parsers with complex behaviour are built up by combining parsers with simpler behaviour.
- For example, if `p` is an existing parser that accepts input `i`, then `p*` is a new parser which accepts zero or more occurrences of `i`. The `*` method is a parser combinator.
- Benefits and encourages a declarative style of programming.

From grammars to code

EBNF grammar for XML (abridged)

```
document    = declaration space? element space?
declaration = ‘<?xml’ version encoding? standalone? space? ‘?>’
version     = space ‘version’ equals versionNum
encoding    = space ‘encoding’ equals encodingName
standalone  = space ‘standalone’ equals (yes | no)
element     = nonEmpty | empty
nonEmpty    = startTag content endTag
empty       = ‘<’ name attributes space? ‘/>’
startTag    = ‘<’ name attributes space? ‘>’
endTag      = ‘</’ name space? ‘>’
attributes  = (space attribute)*
content     = charData? (element charData?)*
attribute   = name equals string
yes         = ‘"yes"’ | ‘'yes'’
no          = ‘"no"’  | ‘'no'’
equals      = space? ‘=’ space?
```

EBNF grammar for XML (abridged)

```
document        declaration space  element space
declaration             version encoding  standalone  space
version         space           equals versionNum
encoding        space            equals encodingName
standalone      space              equals  yes   no
element         nonEmpty   empty
nonEmpty        startTag content endTag
empty               name attributes space
startTag            name attributes space
endTag               name space
attributes       space attribute
content         charData   element charData
attribute       name equals string
yes
no
equals          space     space
```

EBNF grammar for XML (abridged)

```
element      = nonEmpty
nonEmpty     =              content



content      =                  element
```

EBNF grammar for XML (abridged)

```
<?xml                                         ?>
      version
      encoding
      standalone

<                          />
<                          >
</               >

"yes"     'yes'
"no"      'no'
       =
```

Terminals

EBNF grammar for XML (abridged)

```
declaration•space?•element•space?
‘<?xml’•version•encoding?•standalone?•space?•‘?>’
space•‘version’•equals•versionNum
space•‘encoding’•equals•encodingName
space•‘standalone’•equals•(         )

startTag•content•endTag
‘<’•name•attributes•space?•‘/>’
‘<’•name•attributes•space?•‘>’
‘</’•name•space?•‘>’
(space•attribute)*
charData?•(element•charData?)*
name•equals•string


space?•‘=’•space?
```

Sequences

EBNF grammar for XML (abridged)

```
                          yes | no
nonEmpty | empty

'"yes"' | "'yes'"
'"no"'  | "'no'"
```

Alternatives

EBNF grammar for XML (abridged)

```
space?            space?
     encoding? standalone? space?

space?
space?
space?

charData?         charData?

space?   space?
```

Options

EBNF grammar for XML (abridged)

```
(space attribute)*
          (element charData?)*
```

Repetitions

Parser for XML (abridged)

```
class XMLParser extends RegexParsers {
  override def skipWhitespace : Boolean = false

  def document    = declaration~(space?)~element~(space?)
  def declaration = "<?xml"~version~(encoding?)~(standalone?)~(space?)~"?>"
  def version     = space~"version"~equals~versionNum
  def encoding    = space~"encoding"~equals~encodingName
  def standalone  = space~"standalone"~equals~(yes | no)
  def element     = nonEmpty | empty
  def nonEmpty    = startTag~content~endTag
  def empty       = "<"~name~attributes~(space?)~"/>"
  def startTag    = "<"~name~attributes~(space?)~">"
  def endTag      = "</"~name~(space?)~">"
  def attributes  = (space~attribute *)
  def content     = (charData?)~(element~(charData?) *)
  def attribute   = name~equals~string
  def yes         = "\"yes\"" | "'yes'"
  def no          = "\"no\"" | "'no'"
  def equals      = (space?)~"="~(space?)

  // ... some rules elided ...
}
```

Parser for XML (abridged)

```
document      declaration  space   element  space
declaration           version  encoding    standalone    space
version       space           equals versionNum
encoding      space            equals encodingName
standalone    space              equals  yes   no
element       nonEmpty   empty
nonEmpty      startTag content endTag
empty             name attributes  space
startTag          name attributes  space
endTag             name  space
attributes     space attribute
content        charData    element  charData
attribute     name equals string
yes
no
equals         space        space
```

Parser for XML (abridged)

```
element         nonEmpty
nonEmpty                 content

content                               element
```

Recursion

Parser for XML (abridged)

```
<?xml                                                ?>
      version
      encoding
      standalone


<                            />
<                            >
</                 >


\"yes\"     'yes'
\"no\"     'no'
        =
```

Terminals

Parser for XML (abridged)

```
declaration~(space?)~element~(space?)
"<?xml"~version~(encoding?)~(standalone?)~(space?)~"?>"
space~"version"~equals~versionNum
space~"encoding"~equals~encodingName
space~"standalone"~equals~(          )

startTag~content~endTag
"<"~name~attributes~(space?)~"/>"
"<"~name~attributes~(space?)~">"
"</"~name~(space?)~">"
 space~attribute
(charData?)~(element~(charData?) *)
name~equals~string


(space?)~"="~(space?)
```

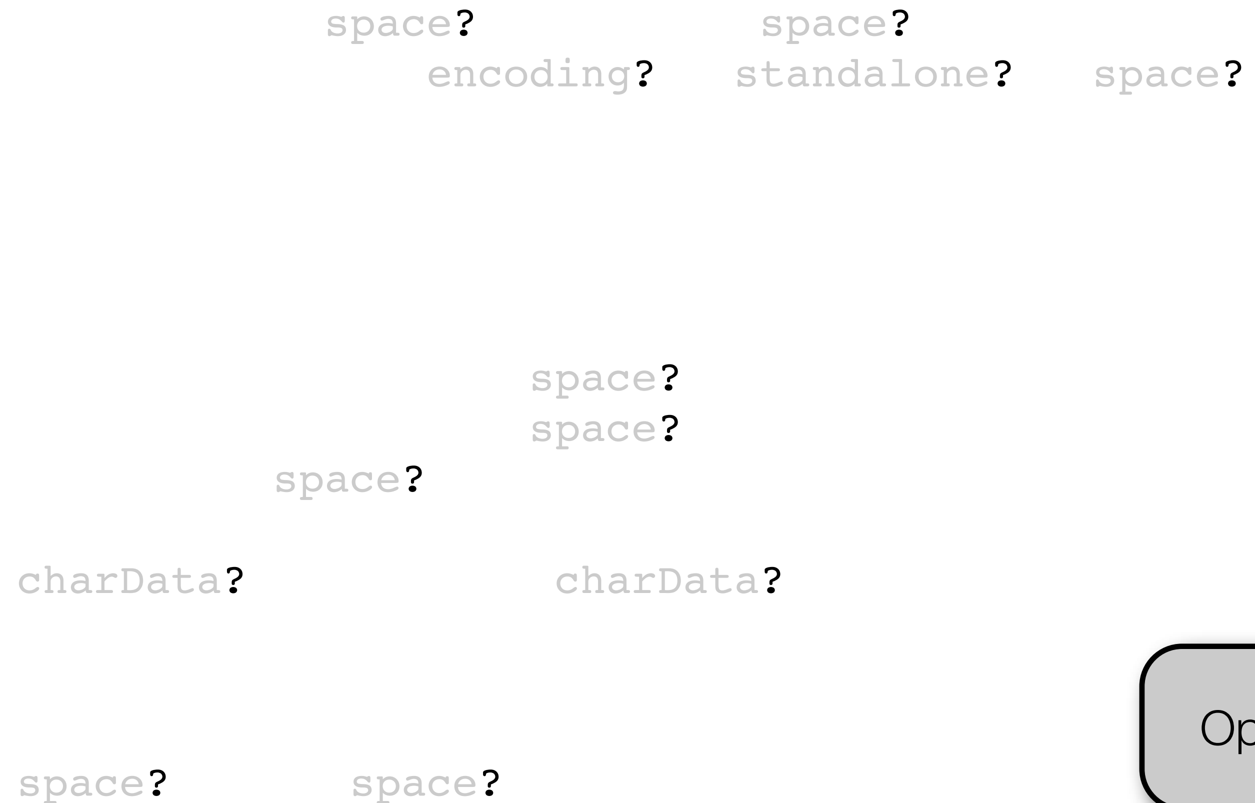
Sequences

Parser for XML (abridged)

```
                                    yes | no
nonEmpty | empty

"\"yes\"" | "'yes'"
"\"no\"" | "'no'"
```

Alternatives

Parser for XML (abridged)



Parser for XML (abridged)

```
space~attribute *
          element~(charData?) *
```

Repetitions

Running the parser

```
object Main extends XMLParser {
   def main(args: Array[String]) {
      val reader = new FileReader(args(0))
      println(parseAll(document, reader))
   }
}
```

Input file:

```
<?xml version="1.0"?><test answer="42" />
```

Output:

```
[1.42] parsed: ((((((((<?xml~((( ~version)~((None~=)~None))
~(("~1.0)~")))~None)~None)~None)~?>)~None)~((((<~test)~List((
~((answer~((None~=)~None))~(("~42)~"))))))~Some( ))~/>))~None)
```

Types

What is a Parser?

- `abstract class Parser[+T] extends (Input => ParseResult[T])`
- `type Input = Reader[Elem]`
- `Elem` is the (abstract) type of input tokens.
- `Reader` is a stream of values annotated with source coordinates.
- `ParseResult` encodes the success or failure of a parse.
- `parseAll [T](p:Parser[T], in:java.io.Reader): ParseResult[T]`

Combinator types

For `Parser[T]` we have:

- `def ~ [U](p: => Parser[U]): Parser[~[T, U]]`
- `def | [U >: T](q: => Parser[U]): Parser[U]`
- `def rep[T](p: => Parser[T]): Parser[List[T]]`
- `def opt[T](p: => Parser[T]): Parser[Option[T]]`

For `RegexParsers` we have:

- `type Elem = Char`
- `implicit def literal(s: String): Parser[String]`
- `implicit def regex(r: Regex): Parser[String]`

Combinator types

```
def ~ [U](p: => Parser[U]): Parser[~[T, U]]
```

```
case class ~[+a, +b](val _1 : a, val _2 : b) extends Product
```

Combinator types

`U` is a supertype of `T`

```
def | [U >: T](q: => Parser[U]): Parser[U]
```

Homework: why doesn’t | have this type instead?

```
def | [U](q: => Parser[U]): Parser[Either[T,U]]
```

Combinator types

`*` is a postfix operator which is a synonym for `rep`.
Notice the list type in the output.

```
def rep[T](p: => Parser[T]): Parser[List[T]]

def opt[T](p: => Parser[T]): Parser[Option[T]]
```

`?` is a postfix operator which is a synonym for `opt`.
Notice the option type in the output.

Combinator types

```
def string       = doubleString | singleString
def charData     = "[^<]+".r
def space        = """\s+""".r
def name         = """(:|\w)((\-|\.|\d|:|\w))*""".r
def doubleString = "\""~"""[^"]*""".r~"\""
def singleString = "'"~"[^']*".r~"'"
```

`implicit` methods allow Scala to convert string and regex literals into parsers that accept those literals.

```
implicit def literal(s: String): Parser[String]

implicit def regex(r: Regex): Parser[String]
```

A minor typing frustration

Actually, we need to write the result type for 'content'. Otherwise, scalac says:

```
error: recursive method element needs result type
```

```
def content : Parser[Any] = (charData?)~(element~(charData?) *)
```

Refining the output

Too much unnecessary information in the output

Recall the output from before:

```
[1.42] parsed: ((((((((<?xml~((( ~version)~((None~=)~None))
~(("~1.0)~")))~None)~None)~None)~?>)~None)~((((<~test)~List((
~((answer~((None~=)~None))~(("~42)~"))))))~Some( ))~/>))~None)
```

Why bother keeping the whitespace and punctuation?

Combinators for left and right selection

Combine two parsers and discard the result of the left one:

```
def ~> [U](p: => Parser[U]): Parser[U]
```

Combine two parsers and discard the result of the right one:

```
def <~ [U](p: => Parser[U]): Parser[T]
```

Combinators for left and right selection

```
def document     = declaration~>(space?)~>element<~(space?)
def declaration  = "<?xml"~version~(encoding?)~(standalone?)~(space?)~"?>"
def version      = space~"version"~equals~string
def encoding     = space~"encoding"~equals~string
def standalone   = space~"standalone"~equals~(yes | no)
def element      = nonEmpty | empty
def nonEmpty     = startTag~content~endTag
def empty        = "<"~>name~attributes<~(space?)<~"/>"
def startTag     = "<"~>name~attributes<~(space?)<~">"
def endTag       = "</"~>name<~(space?)<~">"
def attributes   = (space~>attribute *)
def content : Parser[Any] = (charData?)~(element~(charData?) *)
def attribute    = (name<~equals)~string
def yes          = "\"yes\"" | "'yes'"
def no           = "\"no\"" | "'no'"
def equals       = (space?)~"="~(space?)
def string       = doubleString | singleString
def charData     = "[^<]+".r
def space        = """\s+""".r
def name         = """(:|\w)((\-|\.|\d|:|\w))*""".r
def doubleString = "\""~>"""[^"]*""".r<~"\""
def singleString = "'"~>"[^']*".r<~"'"
```

Combinators for left and right selection

```
def document      = declaration~>(space?)~>element<~(space?)
def declaration   = "<?xml"~version~(encoding?)~(standalone?)~(space?)~"?>"
def version       = space~"version"~equals~string
def encoding      = space~"encoding"~equals~string
def standalone    = space~"standalone"~equals~(yes | no)
def element       = nonEmpty | empty
def nonEmpty      = startTag~content~endTag
def empty         = "<"~>name~attributes<~(space?)<~"/>"
def startTag      = "<"~>name~attributes<~(space?)<~">"
def endTag        = "</"~>name<~(space?)<~">"
def attributes    = (space~>attribute *)
def content : Parser[Any] = (charData?)~(element~(charData?) *)
def attribute     = (name<~equals)~string
def yes           = "\"yes\"" | "'yes'"
def no            = "\"no\"" | "'no'"
def equals        = (space?)~"="~(space?)
def string        = doubleString | singleString
def charData      = "[^<]+".r
def space         = """\s+""".r
def name          = """(:|\w)((\-|\.|\d|:|\w))*""".r
def doubleString = "\""~>"""[^"]*""".r<~"\""
def singleString = "'"~>"[^']*".r<~"'"
```

Mind your precedence and associativity

```
def attribute = name<~equals~string
```

Parses as:

```
def attribute = name<~(equals~string)
```

But that would discard the string!

A much more refined output

Here's the output from the modified parser:

```
[1.42] parsed: (test~List((answer~42)))
```

Modifying the output

- Suppose that we want the parser to build a result of a different type than the default.
- For example we might want to build an XML tree using the data types in `scala.xml`.
- Parser[T] provides various combinators for this purpose, for example:
  - `def ^^ [U](f: T => U): Parser[U]`
  - `def >> [U](f: T => Parser[U]): Parser[U]`

Modifying the output

```
def empty         = "<"~>name~attributes<~(space?)<~"/>" ^^ mkEmpty

def attributes    = (space~>attribute *) ^^ mkAttributes



private def mkAttributes = (list : List[String~String]) =>
   ((Null:MetaData) /: list.reverse) {
      case (atts,key~value) => new UnprefixedAttribute(key,value,atts)
}


private def mkEmpty : String~MetaData => Node = {
   case name~atts => Elem(null, name, atts, TopScope)
}
```

Modifying the output

```
def nonEmpty     = startTag~content~endTag >> mkNonEmpty
```

```
private type NonEmpty = String~MetaData~List[Node]~String

private def mkNonEmpty : NonEmpty => Parser[Node] = {
   case startName~atts~children~endName =>
      if (startName == endName)
         success (Elem(null, startName, atts, TopScope, children:_* ))
      else
         err("tag mismatch")
}
```

Modifying the output

```
def nonEmpty     = startTag~content~endTag >> mkNonEmpty
```

Notice that **`mkNonEmpty`** returns a **`Parser`** as its result.

```
private type NonEmpty = String~MetaData~List[Node]~String

private def mkNonEmpty : NonEmpty => Parser[Node] = {
   case startName~atts~children~endName =>
      if (startName == endName)
         success (Elem(null, startName, atts, TopScope, children:_* ))
      else
         err("tag mismatch")
}
```

Modifying the output

```
def nonEmpty     = startTag~content~endTag >> mkNonEmpty
```

```
def success[T](v: T) : Parser[T]
```

```
private type NonEmpty = String~MetaData~List[Node]~String

private def mkNonEmpty : NonEmpty => Parser[Node] = {
   case startName~atts~children~endName =>
      if (startName == endName)
         success (Elem(null, startName, atts, TopScope, children:_* ))
      else
         err("tag mismatch")
}
```

Modifying the output

```
def nonEmpty     = startTag~content~endTag >> mkNonEmpty
```

```
def err(msg: String) : Parser[Nothing]
```

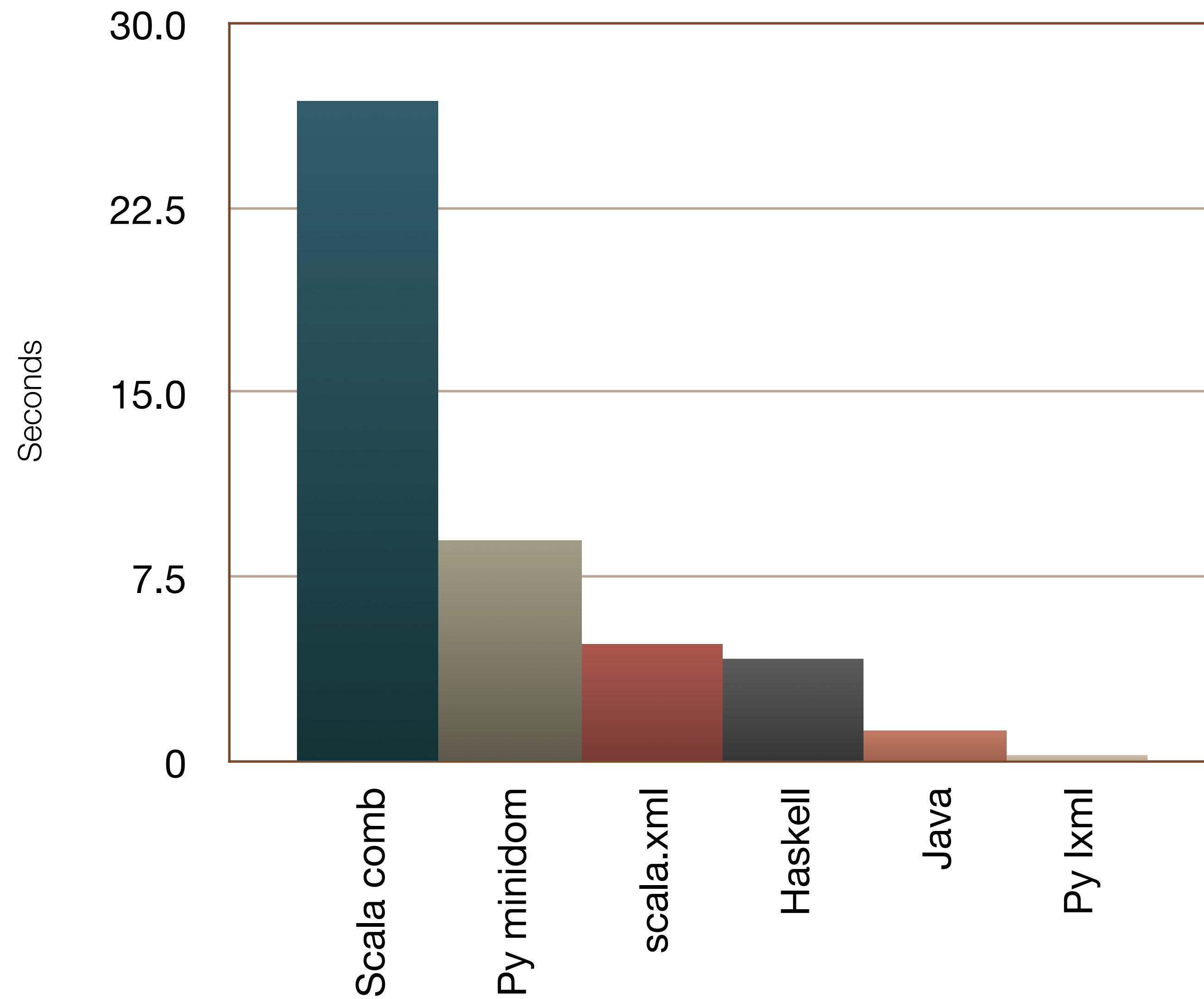
```
private type NonEmpty = String~MetaData~List[Node]~String

private def mkNonEmpty : NonEmpty => Parser[Node] = {
   case startName~atts~children~endName =>
      if (startName == endName)
         success (Elem(null, startName, atts, TopScope, children:_* ))
      else
         err("tag mismatch")
}
```

Performance

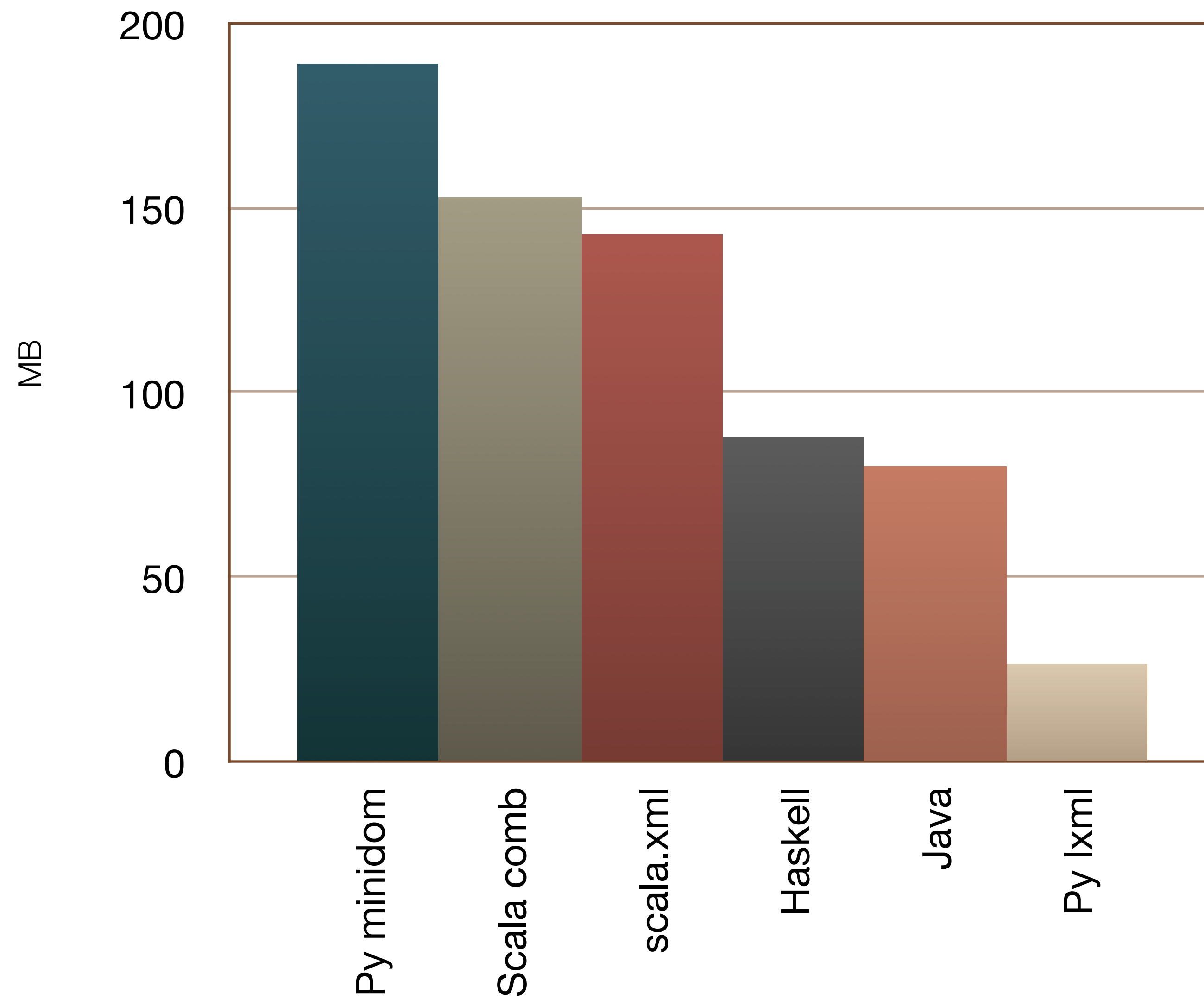
How well does the parser perform?

- Non scientific test: parse a 3MB iTunes file and pretty print the result.
- Compare against the following:
  - Haskell using `Parsec` (see handout).
  - Python using `lxml` (binding to `libxml2`).
  - Python using `xml.dom.minidom`.
  - Scala using `scala.xml`.
  - Java using `javax.xml.parsers`.

Runtime performance



Memory usage



Why bad performance from Scala combinators?

- The combinator for parsing alternatives '|' is backtracking.
- Consider the parser: `p | q`
- First try to parse `p`. If it fails try to parse `q`.
- What happens if `p` and `q` share their left prefix?
- `p` can do work which is repeated in `q`.
- The solution is to left factor the grammar.
- Compare with Parsec in Haskell which is not backtracking (by default).

Homework

- Extend the parser to support more XML (comments, CDATA, entities).
- Left factor the grammar and check for performance improvements.
- Modify the parser to produce better error messages.
- Read chapter 31 of “Programming in Scala” (the stairway book).
- Read the source code of `scala.util.parsing.combinator`.