Scala for TAPL'ers
3

Adriaan Moors
<adriaan cs kuleuven be>
09/11/2007

Parsing

Parsing

- Check whether input has a given structure
  - Typically do more than just *checking*
    - Produce AST
- Acceptable structure is defined by a *grammar*

Grammar (in BNF)

```
term = '\' ident '.' term
     | ident
     | term term
```

Recursive descent

```
term = '\' ident '.' term
     | ident
     | term term
```

```
def ident(input: String): Boolean = ...
def term(input: String): Boolean
  = (input(0) == '\\' && ident(input.substring(1)) &&
     input(i) == '.' && term(input.substring(i+1)))
  || ident(input)
  || term(input) && term(inputRest)
```

Recursive descent

```
term = '\' ident '.' term
     | ident
     | term term
```

```
def ident(input: String): Boolean = ...
def term(input: String): Boolean
  = (input(0) == '\\' && ident(input.substring(1)) &&
     input(i) == '.' && term(input.substring(i+1)))
  || ident(input)
  || term(input) && term(inputRest)
```

Don't know `i` and `inputRest`

Can you spot other (more serious) problems?

First problem: managing input

Essence of this problem:

```
twoIdents = ident ident
```

```
def twoIdents(input: String) = ident(input) && ident(rest)
```

First problem: managing input

Essence of this problem:

```
twoIdents = ident ident
```

```
def twoIdents(input: String) = ident(input) && ident(rest)
```

Beside success, track how much input was consumed:

First problem: managing input

Essence of this problem:

```
twoIdents = ident ident
```

```
def twoIdents(input: String) = ident(input) && ident(rest)
```

Beside success, track how much input was consumed:

```
def twoIdents(input: String): Option[String]
```

First problem: managing input

Essence of this problem:

```
twoIdents = ident ident
```

```
def twoIdents(input: String) = ident(input) && ident(rest)
```

Beside success, track how much input was consumed:

```
def twoIdents(input: String): Option[String]
```

Parsing failed: result is None (`false` in previous slide)
else, result is `Some(x)` where `x` is the rest of the input

Combinators

```
twoIdents = ident ident
```

```
def twoIdents(input: String) = ident(input) && ident(rest)
```

Combinators

```
twoIdents = ident ident
```

```
def twoIdents(input: String) = seq(ident, ident)(input)
```

Factoring out input-management

Combinators

```
twoIdents = ident ident
```

```
def twoIdents(input: String) = seq(ident, ident)(input)
```

Factoring out input-management

```
def ident(input: String): Boolean = ...
```

Combinators

```
twoIdents = ident ident
```

```
def twoIdents(input: String) = seq(ident, ident)(input)
```

Factoring out input-management

```
ident : String => Option[String] //function derived from method
```

Combinators

```
twoIdents = ident ident
```

```
def twoIdents(input: String) = seq(ident, ident)(input)
```

Factoring out input-management

```
ident : String => Option[String] //function derived from method
```

```
def seq(first: String => Option[String],
        snd: String => Option[String]): String => Option[String]
```

Combinators

```
twoIdents = ident ident
```

```
def twoIdents(input: String) = seq(ident, ident)(input)
```

Factoring out input-management

```
ident : String => Option[String] //function derived from method
```

```
def seq(first: String => Option[String],
        snd: String => Option[String]): String => Option[String]
```

This is the essence of combinator parsing!

Exercise: seq

```
// combine two parsers into a new one
// the resulting parser succeeds if first, and then snd succeeds
def seq(first: String => Option[String],
        snd: String => Option[String]): String => Option[String]
```

[redacted]

Censored

Exercise: seq

```
// combine two parsers into a new one
// the resulting parser succeeds if first, and then snd succeeds
def seq(first: String => Option[String],
        snd: String => Option[String]): String => Option[String]
  = { in: String =>
[redacted]
Censored
    }
```

Exercise: seq

```
// combine two parsers into a new one
// the resulting parser succeeds if first, and then snd succeeds
def seq(first: String => Option[String],
        snd: String => Option[String]): String => Option[String]
  = { in: String =>
      first(in) match {
                  Censored
      }
    }
```

Exercise: seq

```
// combine two parsers into a new one
// the resulting parser succeeds if first, and then snd succeeds
def seq(first: String => Option[String],
        snd: String => Option[String]): String => Option[String]
  = { in: String =>
      first(in) match {
        case Some(rest) => snd(rest)
        case None => None
      }
    }
```

Combinator Parsers

- A parser is first-class: a function (and thus an object)
- Build complex parsers by combining simpler ones
- Combinators are just methods that take parsers and produce a new one

Parser

One more piece of information: result value

```
abstract class Parser[+T] extends (Input => Result[T])

def Parser[T](f: Input => Result[T]): Parser[T]
    = new Parser[T]{ def apply(in: Input) = f(in) }
```

```
trait Result[+T]

case class Success[+T](result: T, rest: Input) extends Result[T]
case class Failure(errMsg: String, rest: Input) extends Result[Nothing]
```

Parser with seq

```
abstract class Parser[+T] extends (Input => Result[T]) {
  def ~ [U](p: Parser[U]): Parser[Pair[T, U]]
    = Parser { in: Input =>
      this(in) match {
        case Success(x, rest) =>
         p(rest) match {
           case Success(y, rest2) => Success((x, y), rest2)
           case Failure(e, r) => Failure(e, r)
         }
        case Failure(e, r) => Failure(e, r)
      }
    }
```

```
def ident: Parser[String]
def twoIdents: Parser[Pair[String, String]]
  = ident ~ ident
```

Alternation

```
def | [U >: T](p: Parser[U]): Parser[U]
= Parser { in: Input =>
    this(in) match {
      case s@Success(_, _) => s
      case _ => p(in)
    }
  }
```

Alternation

```
def | [U >: T](p: Parser[U]): Parser[U]
= Parser { in: Input =>
    this(in) match {
      case s@Success(_, _) => s
      case _ => p(in)
    }
  }
```

```
term = '\' ident '.' term
     | ident
     | term term
```

Alternation

```
  def | [U >: T](p: Parser[U]): Parser[U]
  = Parser { in: Input =>
      this(in) match {
        case s@Success(_, _) => s
        case _ => p(in)
      }
    }
```

```
term = '\' ident '.' term
     | ident
     | term term
```

```
def term =
   ( '\\' ~ ident ~ '.' ~ term
   | ident
   | term ~ term
   )
```

Implicit accept

```
def term =
   ( '\\' ~ ident ~ '.' ~ term
   | ident
   | term ~ term
   )
```

```
implicit def accept(e: Char): Parser[Char]
  = Parser { in: String =>
      if(!in.isEmpty && in(0)==e) Success(e, in.substring(1))
      else Failure("expected "+e, in)
    }
```

Implicit accept

```
def term =
   ( '\\' ~ ident ~ '.' ~ term
   | ident
   | term ~ term
   )
```

```
implicit def accept(e: Char): Parser[Char]
  = Parser { in: String =>
      if(!in.isEmpty && in(0)==e) Success(e, in.substring(1))
      else Failure("expected "+e, in)
    }
```

```
def term = accept('\\').~(ident).~(accept('.')).~(term).|
(ident).|(term.~(term))
```

Problem 2: Ordered Choice

```
term = '\' ident '.' term
     | ident
     | term term
```

- In BNF, the order of the alternatives does not matter.
- With our implementation of |, it does!
  - Commits to first successful branch

Problem 2: Ordered Choice

```
def term =
   ( term ~ term
   | '\\' ~ ident ~ '.' ~ term
   | ident
   )
```

- In BNF, the order of the alternatives does not matter.
- With our implementation of |, it does!
  - Commits to first successful branch

Problem 3: Cycles

```
def term =
   ( term ~ term
   | '\\' ~ ident ~ '.' ~ term
   | ident
   )
```

Problem 3: Cycles

```
def term =
   ( term ~ term
   | '\\' ~ ident ~ '.' ~ term
   | ident
   )
```

- `term` is recursive

Problem 3: Cycles

```
def term =
   ( term ~ term
   | '\\' ~ ident ~ '.' ~ term
   | ident
   )
```

- `term` is recursive
  - harmless

Problem 3: Cycles

```
def term =
   ( term ~ term
   | '\\' ~ ident ~ '.' ~ term
   | ident
   )
```

- `term` is recursive
  - harmless
  - left-recursion

Harmless Cycles

```
def term =
   ( term ~ term
   | '\\' ~ ident ~ '.' ~ term
   | ident
   )
```

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

- combinators `~` and `|` take call-by-name arguments

Left-recursion

```
def term =
    ( term ~ term
    | '\\' ~ ident ~ '.' ~ term
    | ident
    )
```

- Resulting parser immediately tries itself on the *same input*
    - (ordering due to semantics of |)
- CBN would not solve anything

Rewriting left-recursion

```
def term0 =
   ( '\\' ~ ident ~ '.' ~ term
   | ident
   )

def term = rep1(term0)
def rep1[T](p: Parser[T]) = p ~ rep1(p) | p
```

Lifting

```
def rep1[T](p: Parser[T]): Parser[List[T]]
  = p ~ rep1(p) ^^ {case (x, xs) => x :: xs}
  | p ^^ {x => List(x)}
```

```
  def ^^ [U](f: T => U): Parser[U]
    = Parser { in: Input =>
      this(in) match {
        case Success(x, r) => Success(f(x), r)
        case Failure(e, r) => Failure(e, r)
      }
    }

  def ~> [U](p: => Parser[U]): Parser[U]
    = this ~ p ^^ {case (x, y) => y}

  def <~ [U](p: => Parser[U]): Parser[T]
    = this ~ p ^^ {case (x, y) => x}
```

Producing an AST

```
def term0 : Parser[Term] =
  ( ('\\' ~> ident) ~ ('.' ~> term) ^^ {case (i, b) => Abs(i, b)}
  | ident ^^ {n => Var(n)}
  )

def term = chainl1(term0, ' ' ^^ {x => App(_: Term, _: Term)})

def ident: Parser[String] = ...

class Term
case class Var(name: String) extends Term
case class Abs(name: String, body: Term) extends Term
case class App(fun: Term, arg: Term) extends Term
```

Lexical Parsing

- Typically, parsing happens in phases:
  - lexical parsing: take stream of characters and produce stream of tokens
  - syntactical parsing: take stream of tokens, produce AST, ...
- Lexical parsing
  - Take care of low-level issues
  - Does not need full power of parsing

Integrated lex'ing

```
def term0 : Parser[Term] =
   ( ('\\' ~> wss(ident)) ~ ('.' ~> wss(term)) ^^ {
                                       case (i, b) => Abs(i, b)}
   | ident ^^ {n => Var(n)}
   )

def term = chainl1(term0, ws ^^ {x => App(_: Term, _: Term)})

def ident = rep1(letter, letter | digit) ^^ {_.mkString("")}

def letter = acceptIf(_.isLetter)
def digit = acceptIf(_.isDigit)
def ws = rep1(accept(' '))
def wss[T](p: Parser[T]): Parser[T] = opt(ws) ~> p <~ opt(ws)
```

Homework

- Get to know the combinators
  - Report&code available (will keep improving it until project assignment is released)
  - Experiment!
    - Suggestion: make your own language for screen scraping:
      - `rep(row containing {bold.class("title") ~ div.class("descr")})`