Foundations of Software

Martin Odersky, EPFL

Slides in part adapted from:
University of Pennsylvania CIS 500: Software Foundations - Fall 2006
by Benjamin Pierce

Course Overview

What is "software foundations"?

Software foundations (or "theory of programming languages") is the mathematical study of the meaning of programs.

The goal is finding ways to describe program behaviors that are both precise and abstract.

- precise so that we can use mathematical tools to formalize and check interesting properties
- abstract so that properties of interest can be discussed clearly, without getting bogged down in low-level details

Why study software foundations?

- To prove specific properties of particular programs (i.e., program verification)
  - Important in some domains (safety-critical systems, hardware design, security protocols, inner loops of key algorithms, ...), but still quite difficult and expensive
- To develop intuitions for *informal* reasoning about programs
- To prove general facts about all the programs in a given programming language (e.g., safety or isolation properties)
- To understand language features (and their interactions) deeply and develop principles for better language design
  (PL is the "materials science" of computer science...)

What you can expect to get out of the course

- A more sophisticated perspective on programs, programming languages, and the activity of programming
  - See programs and whole languages as formal, mathematical objects
  - Make and prove rigorous claims about them
  - Detailed knowledge of a variety of core language features
- Deep intuitions about key language properties such as type safety
- Powerful tools for language design, description, and analysis

Most software designers are language designers!

Greenspun's Tenth Rule Of Programming

> Any sufficiently complicated C or Fortran program contains an ad-hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp.
> – Philip Greenspun

What this course is not

- An introduction to programming
- A course on functional programming (though we'll be doing some functional programming along the way)
- A course on compilers (you should already have basic concepts such as lexical analysis, parsing, abstract syntax, and scope under your belt)
- A comparative survey of many different programming languages and styles

Approaches to Program Meaning

- Denotational semantics and domain theory view programs as simple mathematical objects, abstracting away their flow of control and concentrating on their input-output behavior.
- Program logics such as Hoare logic and dependent type theories focus on logical rules for reasoning about programs.
- Operational semantics describes program behaviors by means of abstract machines. This approach is somewhat lower-level than the others, but is extremely flexible.
- Process calculi focus on the communication and synchronization behaviors of complex concurrent systems.
- Type systems describe approximations of program behaviors, concentrating on the shapes of the values passed between different parts of the program.

Overview

This course will concentrate on operational techniques and type systems.

- Part I: Modeling programming languages
  - Syntax and parsing
  - Operational semantics
  - Inductive proof techniques
  - The lambda-calculus
  - Syntactic sugar; fully abstract translations
- Part II: Type systems
  - Simple types
  - Type safety
  - References
  - Subtyping

Overview

- Part III: Object-oriented features (case study)
  - A simple imperative object model
  - An analysis of core Java
  - An analysis of core Scala

Organization of the Course

Staff

Instructor:	Martin Odersky `martin.odersky@epfl.ch`
Teaching Assistants:	Aleksander Boruch-Gruszecki Contact him about the mailing list! `aleksander.boruch-gruszecki@epfl.ch`
	Lucas Ramirez `lucas.ramirez@epfl.ch`

Information

Textbook: Types and Programming Languages,
Benjamin C. Pierce, MIT Press, 2002

Webpage: `https://fos2020.github.io`

Elements of the Course

- The Foundations of Software course consists of
  - lectures (Tue 14:15-16:00, INM 10)
  - exercises and project work (Wed 11:15-13:00, CO 020)
- The lecture will follow in large parts the textbook.
- For lack of time, we cannot treat all essential parts of the book in the lectures, that's why the textbook is required reading for participants of the course.

Homework and Projects

You will be asked to

- solve and hand in some written exercise sheets,
- do a number of programming assignments, including
  - parsers,
  - interpreters and reduction engines,
  - type checkers

  for a variety of small languages.
- The recommended implementation language for these assignments is Scala.

Scala

- Scala is a functional and object-oriented language that is closely interoperable with Java.
- It is very well suited as an implementation language for type-checkers, in particular because it supports:
  - pattern matching,
  - higher-order functions,
  - inheritance and mixins.

Learning Scala

If you don't know Scala yet, there's help:

- The Scala web site:

  www.scala-lang.org
- On this site, the documents:
  - *A Brief Scala Tutorial - an introduction to Scala for Java programmers.* (short and basic).
  - *An Introduction to Scala* (longer and more comprehensive).
  - *An Overview of the Scala Programming Language* (high-level).
  - *Scala By Example* (long, comprehensive, tutorial style).
- The assistants.

Grading and Exams

Final course grades will be computed as follows:

- Homework and project: 30%
- Mid-term exam: 30%
- Final exam: 40%

Exams:

1. Mid-term: first week of November (exact date TBA)
2. Final exam: to be announced

Collaboration

- Collaboration on homework is strongly encouraged.
- Studying with other people is the best way to internalize the material
- Form pair programming and study groups!
  2-3 people is a good size. 4 is too many for all to have equal input.

"You never really misunderstand something
until you try to teach it...
" – Anon.

Plagiarism

- A single group will of course share code.
- But plagiarizing code by other groups as part of a project is unethical and will not be tolerated, whatever the source.

Part I

Modelling programming languages

Syntax and Parsing

- The first-level of modeling a programming language concerns its context-free syntax.
- Context free syntax determines a set of legal phrases and determines the (tree-)structure of each of them.
- It is often given on two levels:
  - concrete: determines the exact (character-by-character) set of legal phrases
  - abstract: concentrates on the tree-structure of legal phrases.
- We will be mostly concerned with abstract syntax in this course.
- But to be able to write complete programming tools, we need a convenient way to map character sequences to trees.

Approaches to Parsing

There are two ways to construct a parser:

- By hand Derive a parser program from a grammar.
- Automatic Submit a grammar to a tool which generates the parser program.

In the second approach, one uses a special grammar description language to describe the input grammar.

Domain-Specific Languages

- The grammar description language is an example of a domain-specific language (DSL).
- The parser generator acts as a processor ("compiler") for this language — that's why it's sometimes called grandly a "compiler-compiler".
- Example of a "program" in the grammar description DSL:

```
Expr    ::= Term {'+' Term | '−' Term}.
Term    ::= Factor {'∗' Factor | '/' Factor}.
Factor  ::= Number | '(' Expr ')'.
```

Embedded Domain Specific Languages

- An alternative to a stand-alone DSL is an Embedded DSL.
- Here, the DSL does not exist as a separate language but as an API in a host language.
- The host language is usually a general purpose programming language.

We will now develop this approach for grammar description languages.

An EDSL for Parsing in Scala

```
def expr  : Parser[Any] = term ~ rep("+" ~ term | "−" ~ term)
def term  : Parser[Any] = factor ~ rep("∗" ~ factor | "/" ~ factor)
def factor : Parser[Any] = "(" ~ expr ~ ")" | numericLit
```

A small Perl hack could derive the above code from the informal grammar definition above:

- Insert a def at the beginning of each production.
- The "::=" becomes ": Parser[Any] =".
- Sequential composition is now expressed by a ~.
- Repetition {...} is now expressed by rep(...).
- Option [...] is now expressed by opt(...).

Parser Combinators

- The differences between Grammar A and Grammar B are fairly minor.

  (Note in particular that existing DSLs for grammar descriptions also tend to add syntactic complications to the idealized Grammar A we have seen).
- The important difference is that Grammar B is a valid Scala program, when combined with an API that defines the necessary primitives.
- These primitives are called parser combinators.

The Basic Idea

For each language (identified by grammar symbol S), define a function f_S that, given an input stream i,

- if a prefix of i is in S, return Success(Pair(x, i')) where x is a result for S and i' is the rest of the input.
- otherwise, return Failure(msg, i) where msg is an error message string.

The first behavior is called success, the second failure.

The Basic Idea in Code

Assume:

```
class StandardTokenParsers {
    type Parser = Input ⇒ ParseResult
```

where

```
type Input = Reader[Token]   // a stream of Tokens with positions.
```

and we assume a class Token with subclasses

- case class Keyword(chars : String) for keywords,
- case class NumericLit(chars : String) for numbers,
- case class StringLit(chars : String) for strings,
- case class Identifier(chars : String) for identifiers.

Also assume a class ParseResult[T] with subclasses

```
case class Success[T](result : T, in : Input)
extends ParseResult[T]

case class Failure(msg : String, in : Input)
extends ParseResult[Nothing]
```


Object-Oriented Parser Combinators

- In fact, we will also need to express | and ~ as methods of parsers.
- That's why we extend the function type of parsers as follows:

```
abstract class Parser[T] extends (Input ⇒ ParseResult[T]) {
    // An unspecified method that defines the parser function.
    def apply(in : Input): ParseResult

    // A parser combinator for sequential composition
    def ~ ...

    // A parser combinator for alternative composition
    def | ...
}
```

It remains to define concrete combinators that implement this class (see below).

A Generic Single-Token Parser

- The following parser succeeds if the first token in the input satisfies a given predicate p.
- If it succeeds, it reads the token string and returns it as a result.

```
def token(kind : String)(p : Token ⇒ boolean) = new Parser[String] {
    def apply(in : Input) =
        if (p(in.head)) Success(in.head.chars, in.tail)
        else Failure(kind+" expected", in)
}
```

Specific Single-Token Parsers

- The following parser succeeds if the first token in the input is the given keyword chars:
- If it succeeds, it returns a keyword token as a result.

```
implicit def keyword(chars: String) = token("'"+chars+"'") {
  case Keyword(chars1) ⇒ chars == chars1
  case _ ⇒ false
}
```

- Note that keyword is marked implicit, so that you can usually write "if" instead of keyword("if").

- The following parsers succeed if, respectively, the first token in the input is a numeric or string literal, or an identifier.

```
def numericLit = token("number")(_.isInstanceOf[NumericLit])
def stringLit = token("string literal")(_.isInstanceOf[StringLit])
def ident = token("identifier")(_.isInstanceOf[Identifier])
```


The Sequence Combinator

- The sequence combinator $P \sim Q$ succeeds if P and Q succeed sequentially, and returns the concatenation of their results.
- ~ is implemented as a method of class Parser.

```
abstract class Parser[T] {
  def ~ [U](q: Parser[U]) = new Parser[T ~ U] {
    def apply(in: Input) = Parser.this(in) match {
      case Success(x, in1) ⇒
        q(in1) match {
          case Success(y, in2) ⇒ Success(new ~(x, y), in2)
          case failure ⇒ failure
        }
      case failure ⇒ failure
    }
  }
```


Concatenating Results

Normally, the ~ combinator returns a the results of the two parsers that are run sequentially wrapped in a ~-object.

The case class ~ is similar to Pair (but ~ turns out to be nicer – see below).

```
case class ~[T, U](_1: T, _2: U) {
  override def toString = "(" + _1 +" ~ " + _2 +")"
}
```

There are also two variants of ~ which return only the left or only the right operand:

```
P <~ Q     // returns only P's result
P ~> Q     // returns only Q's result
```

The Alternative Combinator

- The alternative combinator $P \mid Q$ succeeds if either P or Q succeeds.
- It returns the result of P if P succeeds, or the result of Q, if Q succeeds.
- The alternative combinator is implemented as a method of class Parser.

```
def | (q : => Parser[T]) = new Parser[T] {
    def apply(in : Input) = Parser.this(in) match {
        case s1 @ Success(_, _) => s1
        case failure => q(in)
    }
}
```


Failure And Success Parsers

- The parser failure(msg) always fails with the given error message. It is implemented as follows:

```
def failure(msg : String) = new Parser[Nothing] {
    def apply(in : Input) = Failure(msg, in)
}
```

- The parser success(result) always succeeds with the given result. It does not consume any input. It is implemented as follows:

```
def success[T](result : T) = new Parser[T] {
    def apply(in : Input) = Success(result, in)
}
```


Result Conversion

The parser P ^^ f succeeds iff P succeeds. In that case it returns the result of applying f to the result of P.

```
def ^^ [U](f : T => U) = new Parser[U] {
    def apply(in : Input) = Parser.this(in) match {
        case Success(x, in1) => Success(f(x), in1)
        case f => f
    }
}
```

A variant ^^^ takes a value V as right hand side argument.

It returns V if the left hand parser succeeds:

```
def ^^^ [U](r : U): Parser[U] = ^^ (x => r)
```


Option and Repetition Combinators

- The opt(P) combinator always succeeds and returns an Option result.

  It returns Some(R) iff P succeeds with R. In case of failure, the result is None.
- The rep(P) combinator applies P zero or more times until it fails, the result is a list of P's results.

The two combinators are implemented as follows:

```
def opt[T](p : Parser[T]): Parser[Option[T]] =
    p ^^ Some | success(None)
def rep[T](p : Parser[T]): Parser[List[T]] =
    p ~ rep(p) ^^ { case x ~ xs => x :: xs } | success(List())
```

Note that neither of these combinators can fail!

The Interleaved Repetition Combinator

The repsep(P, Q) parser parses a (possibly empty) sequence

$P\ Q\ P\ ...\ Q\ P$

It returns a list of all results returned by P.

Example: repsep(ident, ",") parses a list of identifiers separated by commas.

The repsep combinator is implemented as follows:

```
def repsep[T, U](p: Parser[T], q: Parser[U]): Parser[List[T]] =
  p ~ rep(q ~> p) ^^ { case r ~ rs ⇒ r :: rs } | success(List())
```


Other Combinators

More combinators can be defined if necessary.

Exercise: Implement the rep1(P) parser combinator, which applies P one or more times.

Exercise: Define opt and rep directly, without making use of ~, |, and empty.

An Example: JSON

JSON, or JavaScript Object Notation, is a popular data interchange format.

JSON data essentially consists of objects {...}, arrays [...], numbers, and strings.

Here is an example of a JSON value:

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


A JSON parser

```
package examples.parsing

// import the standard parser class
import scala.util.parsing.combinator.syntactical.StandardTokenParsers

object JSON extends StandardTokenParsers {

  // fix some delimiter symbols ...
  lexical.delimiters += ("{", "}", "[", "]", ":")
  // ... and some reserved words
  lexical.reserved += ("null", "true", "false")

  // here are the four productions making up the JSON grammar
  def obj    : Parser[Any] = "{" ~ repsep(member, ",") ~ "}"
  def arr    : Parser[Any] = "[" ~ repsep(value, ",") ~ "]"
  def member : Parser[Any] = stringLit ~ ":" ~ value
  def value  : Parser[Any] = stringLit | numericLit | obj | arr |
                             "null" | "true" | "false"
```

Testing the JSON Parser

Add a method main that can be used to test the parser.

```
def main(args: Array[String]) {
    val tokens = new lexical.Scanner(args(0))
    println(args(0))
    println(phrase(value)(tokens))
}
```

Here are two test runs:

```
>java examples.parsing.JSON "{ "x": true, "y": [1, 2, 3] }"
{ x: true, y: [1, 2, 3] }
[1.26] parsed: (({ ~ List(((x ~ :) ~ true), ((y ~ :) ~ (([ ~ List(1,
2, 3)) ~ ])))) ~ })
```



```
>java examples.parsing.JSON "{ "x": true "y": [1, 2] }"
{ x: true y: [1, 2] }
[1.13] failure: unexpected token string literal y
{ "x": true "y": [1, 2] }
            ^
```


Getting Better Output

- The result of the previous JSON parser was a tree containing all input tokens (slightly chaotically).
- We transform the productions using the ^^ combinator to get more structured output:

```
def obj   : Parser[Any] =  // return a Map
    "{" ~> rep(member) <~ "}" ^^ (ms => Map() ++ ms)
def arr   : Parser[Any] =  // return a List
    "[" ~> rep{value} <~ "]"
def member: Parser[Any] =  // return a name/value pair
    stringLit ~ ":" ~ value ^^
        { case name ~ ":" ~ value => (name, value) }
```

Question: What are more precise types for obj, arr, and member?


```
def value : Parser[Any] = (
    obj
  | arr
  | stringLit
  | numericLit ^^ (_.toInt)  // return an Int
  | "null" ^^^ null          // return 'null'
  | "true" ^^^ true          // return 'true'
  | "false" ^^^ false        // return 'false'
)
```

If we run the test now, we get:

```
>java examples.parsing.JSON1 "{ "x": true, "y": [1, 2, 3] }"
{ x: true, y: [1, 2, 3] }
[1.30] parsed: Map(x → true, y → List(1, 2, 3))
```

Table of Parser Combinators

ident	identifier
keyword(...)	keyword or special symbol (implicit)
numericLit	integer number
stringLit	string literal
P ~ Q	sequential composition
P <~ Q, P ~> Q	sequential composition; keep left/right only
P | Q	alternative
opt(P)	option
rep(P)	repetition
repsep(P, Q)	interleaved repetition
P ^^ f	result conversion
P ^^^ v	constant result

Arithmetic Expressions Again

Here is the full parser for arithmetic expressions:

```
object Arithmetic extends StandardTokenParsers {
  lexical.delimiters ++= List("(", ")", "+", "-", "*", "/")

  def expr : Parser[Any] = term ~ rep("+" ~ term | "-" ~ term)
  def term = factor ~ rep("*" ~ factor | "/" ~ factor)
  def factor : Parser[Any] = "(" ~ expr ~ ")" | numericLit
```

Question: How can we make it evaluate the parsed expression?

A Problem with Top-Down Parsing

As parser combinators work top-down, they don't allow left-recursion.

A production like

```
def expr = expr ~ "-" ~ term
```

would go into an infinite recursion when executed.

The alternative:

```
def expr = term ~ rep("-" ~ term)
```

produces a "right-leaning" tree: X − Y − Z parses

```
X ~ List("-" ~ Y, "-" ~ Z)
```

But the correct reduction/evaluation of $+,-,*,/$ is left-leaning!

Evaluate by FoldLeft

We can solve this problem by delaying reduction until all elements of a repetition have been parsed and then performing a fold left on the list:

```
def expr  : Parser[Int] =
  term ~ rep ("+" ~ term | "-" ~ term) ^^ reduceList
def term  : Parser[Int] =
  factor ~ rep ("*" ~ factor | "/" ~ factor) ^^ reduceList
def factor : Parser[Int] =
  "(" ~> expr <~ ")" | numericLit ^^ (_.toInt)
```

Here, reduceList is defined in terms of the fold-left operation /: ...

```
val reduceList : Expr ~ List[String ~ Expr] => Expr = {
  case i ~ ps => (i /: ps)(reduce)
}
```

... and reduce is defined as follows:

```
def reduce(x: Int, r: String ~ Int) = r match {
    case "+" ~ y => x + y
    case "-" ~ y => x - y
    case "*" ~ y => x * y
    case "/" ~ y => x / y
    case _ => throw new MatchError("illegal case: "+r)
}
```

With this, we get:

```
java examples.parsing.ArithmeticParsers1 "2 * (3 + 7)"
2 * (3 + 7)
[1.12] parsed: 20
```

(Note: the chainl combinator in the standard library does this for you.)

Conclusion

- Combinator parsers give a provide a concise, flexible, and high-level way to construct parsers.
- The token classes of a context free grammar are modeled as primitive parsers.
- The combination forms are modeled as higher-order parsers.
- Combinator parsers are an example of an embedded DSL.
- By contrast, classical parser generators are stand-alone DSLs.
- Advantage of an embedded DSL over a parser generator: It's easier to connect the results of combinator parsers with the environment.
- Disadvantage: lower efficiency – but this can be overcome (using packrat parsing)