Masterarbeit

Combinator Parsing in Scala

zur Erlangung des akademischen Grades Master of Science
vorgelegt dem Fachbereich Mathematik, Naturwissenschaften und Informatik
der Technischen Hochschule Mittelhessen

Eugen Labun

im April 2012

Referent:	Prof. Dr. Burkhardt Renz
Korreferent:	Prof. Dr. Thomas Karl Letschert

Ars longa, vita brevis!

– Hippocrates

Abstract

Parsing was for decades considered as a rather difficult task from both theoretical and practical points of view. We believe that complexity is not an inherent property of this area of computer science. Two fundamental language-independent approaches can significantly simplify dealing with parsers and eliminate entire classes of problems:

- using *recognition-based* Parsing Expression Grammars (PEGs) instead of *generative* context-free grammars (CFG) as the formalism for definition of languages;
- using *combinator parsers* instead of *parser generators* as the design pattern for implementing parsers.

Additional advantage comes from the use of *Scala* as the implementing language, whose syntactical flexibility and a number of special language features make possible to write readable and concise definitions of grammars directly as Scala code.

We analyse the relationship of PEGs and combinators to the traditional parsing approaches, show advantages of Scala as the host language, go stepwise through basic usage patterns of parsing techniques (validator, interpreter, compiler) including working with intermediate representations (implementation and evaluating of abstract syntax trees, AST), investigate common problems that can arise during parsing and usage of Scala, and describe in details implementations of two parsing projects: propositional logic and lambda calculus.

Contents

Listings

Preface

Origins of this thesis

Parsing has always been one of the most interesting areas of computer science for me. Not only to implement programming languages, but to perform general data analysis, too. In the last months I was also learning Scala, this exciting language that extended my thinking about languages and programming, and showed that many problems can be solved in an elegant and concise way. I learned about combinator parsing in Scala and found this technique very promising. I decided to explore this topic in depth.

Goals

The main goal of my investigations was to learn how to use combinator parsing techniques to solve practical tasks. I hope that this thesis will also be useful for those who studies combinator parsing in Scala. For this purpose the explanations contain a lot of thoroughly tested[1] code examples. And last but not least, this was a great opportunity to practice my written English.

Overview of the content

Chapter 1 first introduces basic notions: parser, combinator parsing, primitive parser, combinator. Then Scala combinators are compared with competitors from different points of view:

- as a notation system: with other familiar grammar notation systems (EBNF and regex),
- as a design pattern: with specialized parsing systems (parser generators),
- by host language (Scala): with other programming languages.

Chapter 2 investigates relationships of combinator parsers to other notions and approaches widely known in the area of *parsing methods* (recursive-descent, scannerless, top-down, bottom-up, LL, LR), *grammars* (generative / CFG, recognition-based / PEG), and definitions of *languages*.

Chapter 3 shows basic usage patterns of parsing techniques along a stepwise development of an arithmetical parser. At the beginning, the parser acts as a *validator* of the user input. Then it is exended to a *interpreter*, that computes the result of an expression. Introducing an intermediate representation (IR), which can be evaluated independently, turns the parser into a *compiler*.

Chapter 4 describes some problems that may arise during development of parsing programs with Scala, and shows how they can be solved.

[1] Java 6, Scala 2.9

Chapter 5 presents two projects developed using combinator parsers in Scala: propositional logic, and lambda calculus.

Acknowledgments

I would like to thank my teachers prof. dr. Burkhardt Renz and prof. dr. Thomas Karl Letschert for supervision, for many interesting discussions, and for supporting and helping me in study and behind it.

I am grateful to all members of Scala community for sharing their knowledge and for insightful discussions.

I thank to my family for patience during my work on this thesis.

Chapter 1

Introduction

This chapter introduces basic notions related to combinator parsing and compares it to some other parsing techniques and approaches. Combinators in Scala are also compared with combinators in other languages.

1.1 Basic notions

Parsers are computer programs that transform flat text streams into some data structures, which are (hopefully) more suitable for further processing. Sometimes, programs that traverse and transform *structured* data are also called parsers. Nevertheless, the common meaning of parsers is to work with *flat* streams as their input.

There are many different approaches how to build parsers. One of them are combinators.

A *combinator parsing system* consists of primitive parsers and combinators.

Primitive parsers recognize some flat subsequence of the input (such as a given string literal), or have a special meaning (such as parsers that always succeed or always fail regardless of the input).

Combinators are functions that combine parsers into bigger building blocks to enrich, change or create a new behavior of the combined parsers as a whole. The result is again a parser, which can be used for further compositions. I.e. class of parsers is closed under combinator operations. The most important combinators are:

- sequencing (several parsers in the given order must succeed on the input),
- alternation (one of the given parsers must succeed), and
- repetitions (repeated applying of a parser to the input):
  - zero or more,
  - one or more,
  - zero or one (option).

Making a parser for some concrete purpose means to *compose* that parser, using parser combinators, from primitive and already combined parsers.

Thus combinators are *a concept of the programming interface, a design pattern.* They are closely related to the composite[1] pattern.

[1] http://en.wikipedia.org/wiki/Composite_pattern

1.2 Combinators vs EBNF and regular expressions

The common grammar notations, EBNF and regex[2] (as well as their numerous variants), can be considered as combinator-like systems: they also use small primitive building blocks that can be composed by means of sequencing, alternation, repetition, and so on. For instance, consider the following grammar that recognize a whole number (assuming, "`digit`" and "`\d`" are already defined):

EBNF	`number = ["-"], digit, {digit};`
regex	`number = -? \d+`

In both cases, the common meaning is that a whole number is comprised by a sequence of the following elements: 1) an optional sign, and 2) one or more digits. What differs is only the notation for abstractions "sequence", "option" (zero or one repetition), "one or more" repetition.

The set of primitive building blocks ("terminals") can be considered as fixed in all three systems: combinators, EBNF, and regex.

The adavantage of combinators and EBNF over the most regex implementations is the possibility to *name* the newly composed building blocks, and use them for further compositions, especially for nested recursive constructions. The `digit` and `number` elements in the EBNF example above are such composed building blocks. On the other hand, regex implementations usually come with a rich set of powerful predefined building blocks, such as `\d` in the regex example above. Nevertheless the most regex implementations do not support referencing already created building blocks (such as `number` in the regex example above) *by name* in another or the same regex expressions. Some regex implementations allow them to be referenced *by value* (thereby treating them as macros); an attempt to define a recursive macro would lead to an infinite macro expansion though.

EBNF and regex have fixed sets of available compositions to build expressions: an (often implicit) sequencing, alternation, option, some kinds of repetitions.

The crucial advantage of combinator parsing over (E)BNF and regex is the possibility to introduce *new composition abstractions*, that is building blocks with new behavior that are *parameterized* with other building blocks. Such parameterized combinators can be used for:

- recognizing elements separated or surrounded by other elements (e.g. comma-separated lists, parenthesed expressions),
- dynamic creating of parsers based on already parsed information (can be used, for example, to match closing XML tags).
- transformation ("rewriting") of recognized parts of the input into some data structures (e.g. creating of AST nodes, left- or right-folding of lists with recognized elements),
- associating any actions with parsing rules.

These new abstractions can be used in turn to create further building blocks or new abstractions. This ability is extensively used in combinator parsing systems.

EBNF is a *notation*, not an implementation. They cannot use anything but a few predefined constructions. Nevertheless, EBNF grammars are independent from the language, in which the parser will be finally implemented. They have to be ported to that language though.

Combinators are *libraries* in some host languages. One one side, they can use the full power of that languages for parsing purposes. On the other side, they have concrete notations bound to their host languages. The code of parsers implemented as combinators in different host languages is not directly interchangable.

[2] We use the term "regex" to refer to advanced implementations of regular expressions (as found in the most modern programming languages) that go beyond Type-3 regular grammars of the Chomsky hierarchy, mainly thanks to backreferences and lookarounds.

1.3 Combinators vs specialized parsing systems

Specialized parsing systems, such as ANTLR or JavaCC, imply at least:

1) an own *grammar notation system*, which is usually based on EBNF / regex, and
2) a *parser generator*: a program that generates the parser (in the *target* language, e.g. Java or C++) that in turn can parse the input as defined in the grammar.

Auxiliary structures and routines (such as actions) have to be programmed in the target language. The parser writer has therefore to deal with two languages (including their associated tools and workflow): with the grammar language[3] of the parser generator, and with the target language.

In contrast, *combinator parsers* are implemented in the host language as a library. The parsing rules as well as auxiliary routines are both written in the host language. This eliminates an indirection level and the corresponding production step between the grammar and the parser. The parser writer has to deal with only one language[4] (and the corresponding infrastructure).

Specialized parsing systems have full control on parser code generation. They can apply arbitrary optimizations to the code. Capabilities of *combinators* are rather limited in this aspect. They are instructions in some host language, that get compiled by compiler of that language. Generally[5], they cannot affect the logic of compilation.

The parser construction phase occurs only once in a *specialized parsing system*. *Combinators* have to buid the parser structure on each program start.

1.4 Combinators in Scala vs other languages

The basic idea of the combinator parsing dates back to 1970s and became popular since 1980s in a variety of functional programming languages. In the meantime combinator parsing libraries have been written for virtually all programming languages regardless of their functional nature (in object-oriented languages, such as Java or Scala, combinators are implemented as methods).

The expressive power of languages, naturally limited by their syntax and features, differs though. That means that the parser programs also look and work differently.

Scala is a modern general purpose language that allows for concise yet type-safe syntax. Combinator parsers written in Scala are very expressive and good readable.

As an example, consider a parser that recognizes an expression consisting of 1) a sum of two numbers, or, alternatively, 2) a variable. In an EBNF notation it can be expressed as:

```
expr = number "+" number | variable
```

In Java, it could look like this:

```
class Expr {
  static Expression parse() {
    return Alt.parse(Seq.parse(Number.parse(), Lit("+").parse(), Number.parse()),
        Variable.parse())
  }
}
```

[3] *External DSL* (DSL: domain specific language) from the viewpoint of the target language, in which the parser and the main program are implemented.

[4] The parser library acts as an *internal DSL* from the viewpoint of the host language.

[5] Scala supports compiler plugins (which can be triggered e.g. with annotations in source code) that can modify the intermediate AST during compilation.

And in Scala it could be as concise as:

```
def expr = number ~ "+" ~ number | variable
```

It is only marginaly more verbose than the shortest EBNF notation, but it is a valid Scala code: compilable and typechecked by compiler. A number of language features make such concise notation possible. The most important are:

- *symbols as legal identifiers*: "~" and "|" in the example above are in fact names of methods in the `Parser` class;
- *implicit conversions* of values to values of other types: "+" is a string literal that gets automatically converted to a parser instance (via insertion of a conversion method, like `lit("+")`, in the code by compiler);
- *method calls in infix notation* (also called dotless-, or operator-notation): the example above is in fact the same as `(number.~("+").~(number)).|(variable)`
- *carefully choosed predefined precedences for methods in infix notation*: no parentheses are needed in the example above, since "~" binds more tightly than "|".
- *by-name parameters* make possible to construct recursive parsers without entering an infinite loop. For example, the recursive parser definition

  ```
  def expr: Parser[Expr] = number | "(" ~ expr ~ ")"
  ```

  is correct in Scala as is (i.e. it doesn't cause a stack overflow due to recursive calls of `expr` at the time of the parser construction) because the "~" combinator takes its argument *by name* [OSV10, 9.5 p.218] [Ode11, 4.6.1 p.47] without evaluation. (The only drawback is that recursive methods require specifying of the return type.)

  The most other languages (even the functional ones) perform eager evaluation of method parameters and need some additional tricks to work around this problem. For instance, the Java version of *parboiled*[6] parsing library reads parser definitions through reflection and modifies the code before actual parser construction[7]. The *FParsec*[8] library (written in F#) requires from the user to explicitly create a forwarding parser that at the beginning gets initialized with a dummy parser that has to be replaced with a real parser after finishing constructing the parser hierarchy[9]:

  ```
  // This is F#
  let expr, exprImpl = createParserForwardedToRef()
  // parser definitions go here
  do exprImpl := number <|> between (str_ws "(") (str_ws ")") expr
  ```

  One might wonder how the usual recursive functions work without such problems. The point is that a conditional (enclosed in an *if-then-else* construct) *reduction of parameters* takes place at each recursive invocation, eventually ending up in some elementary case without further recursion. This is not the case with the parsing examples above, where the `expr` method is parameterless and references itself uncoditionally.

[6] http://parboiled.org

[7] https://github.com/sirthias/parboiled/wiki/Rule-Construction-in-Java, https://github.com/sirthias/parboiled/blob/develop/parboiled-java/src/main/java/org/parboiled/Parboiled.java#L33-60

[8] http://www.quanttec.com/fparsec/

[9] http://www.quanttec.com/fparsec/reference/primitives.html#members.createParserForwardedToRef, http://hestia.typepad.com/flatlander/2011/07/recursive-parsers-in-fparsec.html, http://www.quanttec.com/fparsec/tutorial.html

Note that the combinator methods, such as "~" or "|", do not do any parsing. They build parser instances whose implementation methods (called `apply` in Scala std. lib.) perform actual parsing according to semantics of the corresponding combinators. E.g. the expression "`p | q`" creates an instance of the alternation parser that holds references to parsers `p` and `q` and whose `apply` method first calls `p.apply` and then `q.apply` if `p` failed.

The language features of Scala make the syntax of combinators one of the most concise, readable, and expressive across all existing programming languages.

Chapter 2

Theoretical Background

In this chapter we investigate relationships of combinator parsers to parsing methods, to grammar types, and to approaches of language definitions.

2.1 Parsing methods

As a design pattern of the programming interface, combinators are not bound to a concrete parsing method. In principle they can implement any. In praxis many combinator parsing libraries are build upon the same design decisions.

2.1.1 Recursive-descent

The most combinator parsing libraries (including the one shipped with Scala) implement a *recursive-descent* parsing method.

In a recursive-descent parser each nonterminal is represented via a procedure that recognizes that nonteminal. These procedures (methods or functions) may be mutually *recursive*. To recognize a nonterminal X the parser *descents* into the right-hand side of the rule that describes X and tries to recognize elements that comprise the X. For nonterminals the same process as described repeats again, for terminals a direct comparison with the input is performed. Thus, the grammar iteself serves as a guide to the parser's implementation: the same principle as for parser combinators. (Combinators extend this principle with the possibility to seamlessly combine those parsing routines.) So the choice to use a recursive-descent parsing for combinators implementation looks very natural.

2.1.2 Scannerless

Traditionally the parsing process is splitted into two parts:

- **lexical** analysis: building a stream of tokens from the input text. This task is performed by a scanner (also called lexer), usually powered by regular expressions.
- **syntactical** analysis: building hierarchical structures from the stream of tokens. Performed by a parser (in the narrow sense of the word), powered by context-free grammars.

Sometimes this design is less than satisfying though. For instance:

1) The lexer might depend on surrounding syntactical context to decide, which token type to create. For example, the input part "`>>`" in C++ can be:

(a) a “right_shift_operator” token (as in “`x >> 2;`”), or

(b) two “right_angle_bracket” tokens (as in “`vector<vector<float>> MyMatrix;`”).

2) The lexer might also lack parser’s possibilities of nested / hierarchical matching, e.g. to recognize nested comments. (A token is always a *flat* subsequence of the input in the traditional approach.)

In such cases the so called *scannerless parsing* can help. In this approach the both parts of analysis are specified in one unified grammar and performed by the same piece of software: the parser. Freeing lexical syntax from the restrictions of regular expressions also enables tokens to have hierarchical characteristics, or even to refer back to the hierarchical portion of the language.

Whether a certain parsing method is a scannerless or a “scannerful” one, sometimes cannot be definitely answered though. The part of a parser that performs matching of terminals corresponds in principle to a lexer. If the other part of the parser doesn’t access the input text directly and works only with tokens supplied by the “lexer”-part, then such a system can be considered as a traditional two way parsing described above. There is nothing wrong with that modular design. Important is, that either part of the parser *can* access the input text directly when it’s needed. Such implementations we will call “scannerless”.

Most combinator parser implementations, including the standard Scala one, feature scannerless parsing.

2.1.3 Top-down / bottom-up, LL / LR question

The terms *top-down, bottom-up, LL (“left-to-right, left-most derivation”), LR (“left-to-right, right-most derivation”)* are addicted to the understanding of parsing as a reconstruction of the implied *generating* of the input in accordance with a context-free grammar. Different ways and different results of such a reconstruction process are possible because context-free grammars (as well as any purely generative grammar) do not define operational semantics for recognizing.

We believe that a better approach for parsing is to use *recognition-based* grammars, where semantics of recognizing are precisely defined. In this case the questions of top-down vs bottom-up, LL vs LR parsing methods, and the whole class of the related problems (such as resolving of “shift-reduce” and “reduce-reduce” conflicts) are not more existing. The section about grammars explains the details about generative and recognition-based grammars.

2.2 Grammars

A parser is usually implemented in accordance with a formal grammar. But what for a grammar can it be?

2.2.1 Unsuitability of context-free grammars (and generative grammars in general)

Over decades, the syntax of programming languages (or data structures to parse) was described by their designers almost exclusively with *context-free* grammars. Being designed for modelling *natural* (human) languages, this grammar type (as well as any other grammar type from the Chomsky hierarchy) is not well suited for parsing *formal* languages though. The points are the following:

- The grammars of the Chomsky hierarchy are ***generative***: they only define how to *produce* strings that belong to a language (hence the term “production rules” and the right arrow “→” between the left- and the right-hand side of a rule). The parser’s task is opposite: to *recognize* whether a given string belongs to a language. Since operational semantics for recognizing are generally

not defined for these grammars, this question is often not easy to answer. The issue is handled separately as a so called *word problem*. Attempts to solve this problem led to arising of numerous parsing techniques, such as LR, LL, LALR, CYK, GLR and their subtypes, just to name some. Unfortunately these parsing techniques also often impose different requirements on the grammar specification and can treat the same specification in different incompatible ways, so that many concrete subtypes of context-free grammars exist today (usually named by the parsing method used, as in "LR(1) grammar"), and a grammar developed for a specific parser type usually cannot be used for another parser type.

  - Example of restrictions on the grammar notation that different parsing techniques may require:

    The LR parser generators `yacc` and `bison` cannot understand such convenient constructions as "zero or more repetition", "one or more repetition", "option". All these notions must be encoded using recursive rules and empty alternatives. So if you have a grammar specification in EBNF (which do allow those notions), it cannot be used "as is" with this parsing method. Needeless to say that conversions between the grammar notation types do not improve maintainability and readability of grammars, and can also introduce subtle differences in the recognized language.

  - Example of different treatment of a grammar by different parsing methods:

    Even in case of an *unambiguous* context-free grammar one might be surprised that its interpretation may vary across different parsing techniques. For instance, consider the following unambiguous context-free grammar that produces an odd number of "`x`":

    ```
    S -> x S x | x
    ```

    An LR or LL parser will not recognize this language though [For02] (for example, the sequences composed of 5, 9 or 11 "`x`"s will not be recognized), whereas a general context-free parser, such as Earley, will do.

- The grammars in the Chomsky hierarchy (intentionally) allow for ***ambiguities*** (i.e. multiple derivations for the same input). This might be useful for modelling natural languages. But it's definitely not for formal (machine-oriented) languages, such as programming languages, that are intentionally designed to avoid ambiguities. An ambiguous program has no meaning [CT12, p.2]. Examples of requirements that are difficult or impossible to express unambiguously in a context-free grammar:

  - an identifier matches the longest possible sequence of characters;
  - "`else`" belongs to the innermost "`if`" in an "`if-then-else`" construct ("dangling ELSE problem"), so that the input

    ```
    if (cond1) then  if (cond2) then expr1  else expr2
    ```

    gets parsed as

    ```
    if (cond1) then (if (cond2) then expr1  else expr2)
    ```

    and not as

    ```
    if (cond1) then (if (cond2) then expr1) else expr2
    ```

Sometimes such cases can be handled by introducing additional rules into the grammar, or by specifying additional meta-rules informally, i.e. independently of the formal grammar ("longest

match", "definition over statement", etc.), or by some parser-implementation specific solutions. Ambiguity in CFGs is difficult to avoid even when we want to, and it makes general CFG parsing (such as GLR, GLL, CYK, Earley parsing methods, see above) an inherently super-linear-time problem [For04] (for instance, GLL: $O(n^3)$ in the worst case, GLR: $O(n^4)$ in the worst case[1]).

2.2.2 PEG: a recognition-based grammar formalism

Given the problems with generative grammars, one can reasonably ask:

> "Why not to specify languages by means of *recognizing*, not by means of *generating*?"

Indeed, we can. The recognition-based systems TS/TDPL and gTS/GTDPL were developed as early as around 1970. A modern, extended reincarnation of them is *PEG (parsing expression grammars)* [For04]. Via introducing a very simple principle this formalism eliminates the whole class of problems discribed above. Moreover, the grammars specified using PEG are fully interchangable regardless of concrete PEG-parser implementations. Here are the keynotes about PEG:

- PEG solves the ambiguity problem "by not introducing ambiguity in the first place" [For04]:
  - instead of nondeterministic choice between alternatives, PEG uses *prioritized choice*:
    - it tries the alternatives in their order,
    - it unconditionally consumes the first successful match.
  - the repetitions (`e?` "zero or one", `e*` "zero or more", `e+` "one or more") unconditionally consume the longest possible sequence[2].

  For example, the above mentioned "dangling ELSE problem" does not exist in PEG. The following simple PEG grammar recognizes an arbitrary nested "`if-then-else`" construct as desired:

```
"if" condition "then" expression ("else" expression)?
```

- PEG also introduces *syntactic predicates* (syntactic conditions, which do not consume[3] any input, but influence whether the current sequence succeeds or fails as a whole):
  - `&e` is a positive look-ahead. It succeeds if `e` succeeds.
  - `!e` is a negative look-ahead. It succeeds if `e` fails.

  Syntactic predicates are not directly related to ambiguity-problem, but they can simplify grammar specification considerably and make possible to recognize more complex languages than context-free ones. For example the classical context-sensitive language $a^n b^n c^n$ can be recognized using PEG as follows [For04]:

  $$\begin{array}{lcl} A & \leftarrow & a\,A\,b \,/\, \varepsilon \\ B & \leftarrow & b\,B\,c \,/\, \varepsilon \\ D & \leftarrow & \&(A\ !b)\ a{*}\ B\ !. \end{array}$$

  where "D" denotes the root rule, and "!." stands for "not a char" thereby denoting the end of the input.

- PEG combines the possibilities of two popular notations for generative grammars – regular expressions and EBNF – in one fomalism equally suitable for lexing, parsing, or scannerless parsing. The operators for constructing parsing expressions are shown in table 2.1.

- PEG defines *well-formedness criteria* for grammars. They allow to check grammars *automatically* for common mistakes, which can lead e.g. to infinite looping. (See [For02] for details).

Operator	Type	Precedence	Description
' '	primary	5	Literal string
" "	primary	5	Literal string
[ ]	primary	5	Character class
.	primary	5	Any character
(*e*)	primary	5	Grouping
e?	unary suffix	4	Optional
*e**	unary suffix	4	Zero-or-more
e+	unary suffix	4	One-or-more
&*e*	unary prefix	3	And-predicate
!*e*	unary prefix	3	Not-predicate
$e_1\ e_2$	binary	2	Sequence
$e_1 \,/\, e_2$	binary	1	Prioritized Choice

Table 2.1: PEG operators [For04]

The most combinator parsing systems, including the one from the Scala standard library, use PEG semantics for recognizing. Table 2.2 shows how PEG operators are implemented in Scala combinator parsing library (class `RegexParsers` and its ancestors). Note that in Scala the pipe symbol "`|`" is used instead of a slash "`/`" for prioritized choice, and PEG's "←" is replaced with an "`=`" symbol.

Description	PEG notation	Scala notation
Literal string	' '	" "
Literal string	" "	" "
Character class	[ ]	"[ ]".r
Any character	.	".".r
Grouping	(*e*)	`(e)`
Optional	e?	`(e?)` or `opt(e)`
Zero-or-more	e*	`(e*)` or `rep(e)`
One-or-more	e+	`(e+)` or `rep1(e)`
And-predicate	&e	`guard(e)`
Not-predicate	!e	`not(e)`
Sequence	$e_1\ e_2$	`e1 ~ e2`
Prioritized Choice	e_1 / e_2	`e1 | e2`

Table 2.2: Implementation of PEG operators in Scala

2.3 Languages

"To the computer scientist, a *language* is a probably infinitely large set of sentences, each composed of tokens in such a way that it has structure; the tokens and the structure cooperate to describe the semantics of the sentence, its "meaning" if you will." [GJ08]

One and the same language can be defined in different ways, for instance:

- by means of a generative or a recognition-based grammar, or using a specific combinator parsing syntax;

[1] https://github.com/djspiewak/gll-combinators
[2] This is a corollary of the previous point, if expressing repetitions via recursion.
[3] I.e. the current parsing position remains the same as before applying the predicate to the input.

- when using generative grammars of the Chomsky hierarchy, it has not necessarily to be a certain grammar type. Often, the same language can be defined using different types of generative grammars.

We will illustrate that on examples of two simple languages.

2.3.1 Example: language "odd number of *x*"

2.3.1.1 Using generative grammars

1) This language (see also section 2.2.1) can be defined by the following (generative) *Type 2 context-free grammar*:

```
S -> x S x | x
```

The problem with this grammar is that many parser types (LL, LR and their subtypes, any recursive-descent) will not recognize this language (as already mentioned in 2.2.1). Those who can (general CFG parsers), cannot work with this grammar effectively.

2) But nothing prevents us to define the same language in a better way: for both to comprehend and to parse. Here is a (generative) *Type 3 regular* grammar:

```
S -> x (xx)*
```

The problem with both generative grammars is that it's not defined, how they should behave during *parsing*. For example, if the input consists of six *x*, will they report a success by recognizing the first five *x* and retaining the last *x* unconsumed, or not?

2.3.1.2 Using recognition-based grammars

For recognition-based grammars, the operational semantics for parsing are precisely defined, so that the questions like the above do not arise.

1) Here is a *PEG* grammar:

```
S <- x (xx)* !.
```

A repetition ('*') in PEG consumes uncoditionally so long as it can. The construction "`!.`" means "not a char", that is the input has to end here (after an odd number of *x* recognized before).

2) Using *combinators*, we can rewrite the above PEG grammar into the Scala syntax:

```
S = x ~ rep(x ~ x) ~ eoi
```

where `eoi` means "end of input".

Alternatively, we can use the fact that the repetion in Scala combinator parsing library returns a *list* of parsed items, so that we can simply parse a "one or more" repetition of *x*, and then directly retrieve the size of the list and check whether it is odd or even:

```
S = (x+) <~ eoi >> {xs => if (xs.size % 2 != 0) success() else failure()}
```

The validating function above plays the role of a *semantic predicate*.

2.3.2 Example: $a^n b^n c^n$ language

Yet another example shows how *semantic predicates* implemented as constructs of the host language can be used to optimize grammar implementation. The goal in this example is to recognise the classical context-sensitive language $a^n b^n c^n$ (note that no CFG grammar exists for this language):

2.3.2.1 Using a generative grammar

Here is a (generative) *monotonic*[4] *Type 1* grammar for that language [GJ08, p.22]:

$$\begin{array}{lcl} S & \rightarrow & abc \mid aSQ \\ bQc & \rightarrow & bbcc \\ cQ & \rightarrow & Qc \end{array}$$

Disadvantage of this grammar is that it's very hard to comprehend. How much time is needed to prove that it realy produces the desired language? And we have yet to find a parsing method that can deal with this grammar...

2.3.2.2 Using a PEG grammar

This *PEG* grammar (from page 22):

$$\begin{array}{lcl} A & \leftarrow & aAb / \varepsilon \\ B & \leftarrow & bBc / \varepsilon \\ D & \leftarrow & \&(A\ !b)\ a*\ B\ !. \end{array}$$

is already a big improvement regarding its comprehensibility for a human, and it can be easily parsed.

2.3.2.3 Using combinators

Can *combinators* provide a better solution? Let us think about how would a human describe a way to recognize that language. He would probably say something like "first take all *a*s, then take all *b*s, then take all *c*s, then check that the number of items in all three categories is equal". This logic can be directly expressed with combinator parsers (implementation in Scala):

```
def S = (a*) ~ (b*) ~ (c*) >> { case as~bs~cs =>
  if (as.size == bs.size && bs.size == cs.size) success() else failure()
}
```

Since this chapter should not assume that the reader is already familiar with the syntax of Scala parser combinators, here is a commented version:

```
def S = (a*) ~ (b*) ~ (c*)  // match all as, then all bs, then all cs
                            // (note: each repetition returns a list of items)

  >>                        // then feed that sequence of three lists
                            // into the following function

{ case as~bs~cs =>          // this construct mainly binds variables as, bs, cs
                            // to the individual lists of matched items

  if (as.size == bs.size    // this should be self-explanatory:
```

[4] Type 1 grammars of the Chomsky hierarchy are often loosely named as "context-sensitive". More precisely, two subtypes of Type 1 *grammars* exist: monotonic and context-sensitive. They are equivalent in their strengths though, i.e. are able to describe the same set of *languages*. For details see [GJ08, p.20].

```
  && bs.size == cs.size)    // if sizes of lists are equal,
    success()               //   then parsing succeeds,
  else                      // else
    failure()               //   fails!
}
```

Note that the combinator parser implementation can be easily extended to recognize $a^n b^n c^n d^n$, $a^n b^n c^n d^n e^n$, etc. languages as well.

Chapter 3

Usage Patterns

This chapter describes a few concrete usage patterns of combinator parsers: validating the input, direct evaluation, working with intermediate representations (including AST definition and evaluation). As a workhorse example serves a calculator-implementation. By the way various combinators ("^^", "^^^", "`chainl1`") are introduced and explained.

3.1 Pure parser, or Validating the input

The simplest usage of parsers is validating the input against a grammar. We will show how to build a parser that validates correctness of simple arithmetical expressions.

3.1.1 Objectives

The parser should be able to recognize expressions built of:

- integer and floating-point numbers;
- infix operators: $+$, $-$, $*$, $/$;
- parentheses.

The parser should treat operators according to the usual precedence rules known from math.

3.1.2 PEG grammar

A PEG grammar can be defined as follows:

```
expr   <- term ("+" term / "-" term)*
term   <- factor ("*" factor / "/" factor)*
factor <- number / "(" expr ")"
```

3.1.3 Encoding precedences in grammar rules

Defining the grammar in a *top-down decomposition* fashion allows to easily encode precedences of arithmetical operations directly in the grammar rules:

- We start with a top rule that should represent the entire input: *expression* (`expr` for short).

- An *expression* consists of parts separated with "+" or "-" operators, which bind more loosely as compared to "*" and "/" operators. We name those parts *terms*. At least one *term* is required.
- *Terms* consist of parts separated with "*" or "/" operators, which bind more tightly than "+" and "-" operators do. We name those parts *factors*. At least one *factor* has to be there.
- A *factor* is either a *number* or a parenthesed *expresssion*.

Alternatively we could also do a *bottom-up composition* of rules, starting with *factors* as the most "atomic" parts and working up to the whole *expression*. That would lead to the same grammar structure. So writing out grammar rules in a top-down or a bottom-up order is merely a matter of taste.

3.1.4 PEG to Scala conversion

Our PEG grammar can be easily translated into the syntax of Scala combinator parsers[1]:

- the concatenation of rules on the right side will be specified with tildes (~),
- the predefined `rep`-combinator is used for zero-or-more repetitions,
- the predefined parser `floatingPointNumber` (provided by the library) can be used to recognize numbers that are represented as integer or floating-point literals.

The result of translation is shown below:

```
expr   = term ~ rep("+" ~ term | "-" ~ term)
term   = factor ~ rep("*" ~ factor | "/" ~ factor)
factor = floatingPointNumber | "(" ~ expr ~ ")"
```

3.1.5 From Scala grammar to a full program

Only a few additional details turn the above Scala grammar into a full stand-alone program that can validate the user input:

```
import util.parsing.combinator.JavaTokenParsers

trait ArithParser extends JavaTokenParsers {
  def expr: Parser[Any] = term ~ rep("+" ~ term | "-" ~ term)
  def term              = factor ~ rep("*" ~ factor | "/" ~ factor)
  def factor            = floatingPointNumber | "(" ~ expr ~ ")"
}

object ArithParserCLI extends ArithParser {
  def main(args: Array[String]) {
    for (arg <- args) {
      println("input:  " + arg)
      println("output: " + parseAll(expr, arg))
    }
  }
}
```

Listing 3.1: Parser that validates arithmetic expressions

[1] See also table "Implementation of PEG operators in Scala" on page 23 showing how primitive parsers and main combinators are implemented.

The program consists of two parts:

- trait `ArithParser` that encapsulates the grammar definition,
- object `ArithParserCLI` that mixes in the trait and provides a command-line interface (CLI) to the user.

These parts are explained below.

The trait `ArithParser` extends `scala.util.parsing.combinator.JavaTokenParsers` that provides the definition of the `floatingPointNumber`[2] parser. The non-terminals become method definitions (`defs`). Since the `expr`-parser is recursive (via `term` and `factor`) its return type cannot be inferred by the actual Scala compiler, and therefore needs explicit specifying: we use `Parser[Any]`. The return type `Parser[Any]` means that (at the moment, because of a pure validating parser) we don't bother about the specific type of the return value.

The object `ArithParserCLI` contains the `main`-method that iterates over the command-line parameters printing the results of parsing. The method `parseAll(root-parser, input)` is inherited from the trait `scala.util.parsing.combinator.RegexParsers`, which is a parent trait of `JavaTokenParsers`. The result "parsed" means that the input is valid against the grammar, the result "failure" means the opposite.

3.1.6 Usage

After compiling the program file (Listing 3.1) with Scala compiler[3]

```
> scalac calc1.scala
```

we can use the program as follows:

```
> scala ArithParserCLI "10.5 - 4*2"
input:  10.5 - 4*2
output: [1.11] parsed: ((10.5~List())~List((-~(4~List((*~2))))))

> scala ArithParserCLI "5 % 2"
input:  5 % 2
output: [1.3] failure: string matching regex `\z' expected but `%' found

5 % 2
  ^
```

Notes about format of the command line Double quotes in the command line

```
> scala ArithParserCLI "10.5 - 4*2"
```

prevent that the input string gets split by the shell into multiple parameters (the space is a default separator of parameters in the most shells), and that '*' is expanded into the list of files in the current directory. To be sure, which input string is actually parsed by the the program, that string is printed out before parsing (see `main` method in object `ArithParserCLI` in listing 3.1).

[2] https://github.com/scala/scala/blob/v2.9.1/src/library/scala/util/parsing/combinator/JavaTokenParsers.scala#L21-22
[3] Symbol ">" represents the shell prompt.

Notes about the parser responce The numbers in square brackets in the parser responce, as "`[1.11]`" in

```
output: [1.11] parsed: ((10.5~List())~List((-~(4~List((*~2))))))
```

mean a *position* "`[line, column]`", where both numbers are 1-based.

In case of success this is the position of a (potential) next portion of the input. Since the `parseAll` method requires that the whole input is matched, that position is always at the end of the input. Another API method `parse` may succeed without consuming the entire input, in which case the resulting position points to the beginning of the remaining input.

In case of failure the position shows where an error is happened.

The actual error message

```
string matching regex `\z' expected but `%' found
```

is not very user friendly though. Ways of improving error messages will be discussed in section 4.1 on page 45.

3.1.7 Parsing tree

The result of a successful parsing, when no rewrite rules were applied, is represented in Scala combinator parsing library as a *(concrete) parsing tree*. Our validating parser prints out in case of success such a parsing tree in form of a parenthesed expression:

`((10.5~List())~List((-~(4~List((*~2))))))`[4]

Parsing trees are composed of the results of individual parsers:

- A concatenation of results a and b (which is produced by the `a ~ b` construction in the grammar) is displayed as $a \sim b$. Concatenations are implemented as instances of the case class that is named "~" (exactly as the corresponding parser combinator).
- Repeated results of the same type $x_1, x_2, \ldots, x_n$ (which are produced by the `rep(x)` construction in the grammar) are displayed as $List(x_1, x_2, \ldots, x_n)$. Lists are implemented as instances of Scala's immutable `List`.

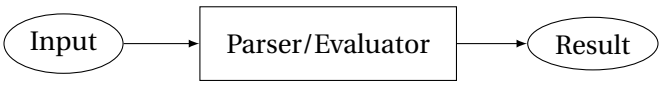
The default parsing tree (more precisely, its `toString` result) is not good readable for a human, but this doesn't matter for a validating parser, since only the type of the overall result is important: a `Success`, or a `Failure`.

The default results of individual parsers (and therefore, the parsing tree) can be easily rewritten, for example, to produce a directly evaluated result of a computation, or to create an intermediate representation of the input in form of an *abstract* syntax tree (AST). Next sections will show how to achieve that.

3.2 Interpreter, or Direct evaluation of the input

As stated before, it is possible to evaluate arithmetical expressions directly with a parser, upon the very same parsing process:

[4] Strictly speaking, the tree in this example is already to some extent *abstracted* from the original input, e.g. whitespace chars are already stripped from the input. By disabling the automatic whitespace handling around terminals, a true parsing tree can be created. But this would also require to change the grammar to handle the whitespace explicitly, and would also make the rewrite rules more complicated (since they would need to take the potential whitespace in results into account).



This effectively turns the parser into an *interpreter*.

All we need to do is to rewrite the results of specific parsers immediately after they succeed. (Recall that combinator parsers always carry the result of the own parsing process.)

3.2.1 Rewriting results with the ^^ combinator

The ^^ combinator is essential and probably the most used rewriting method.

3.2.1.1 Converting String to Double

The successful result of the `floatingPointNumber` parser is[5] a `String` that represents an integer or a floating point literal. That string can be converted to a numeric value of the type `Double` by applying the `toDouble` method. To integrate such conversion into a chain of parsers we would need a combinator that takes a conversion function as a parameter (a function and not a method because methods are not values, and therefore cannot be passed as arguments to other methods) and apply it to the previous parsing result. The predefined ^^ combinator does exactly that.

As a function we use `_.toDouble`, which is a short notation for `x => x.toDouble`. Note the abscence of the parameter type. Since the result type of the previous parser `floatingPointNumber` is `String`, the compiler can infer the type of the function: `String => Double`, so that we don't need to specify it explicitly. Here is the whole parsing and result conversion expression:

```
floatingPointNumber ^^ {_.toDouble}.
```

Note that the type of the expression (i.e. the result type of the ^^ combinator) is `Parser[Double]`. So we are free to use this expression in further compositions of parsers. And we actually do that in the `factor` rule, which is then used in the `term` rule. The complete `factor` rule will be shown in section 3.2.2, which also explains additional combinators used there.

3.2.1.2 Performing arithmetical operations

Using ^^ combinator, a multiplication of two numbers can be immediately replaced with the product of those numbers. Here is the simplified `term` rule:

```
factor ~ "*" ~ factor ^^ {case t1 ~ _ ~ t2 => t1 * t2}.
```

So far so good. Now, the real `term` rule includes not only the "*", but also the "/" operator. Moreover, the chain can include more than one operation. How to handle such chains? Let's look at the full `term` rule:

```
term   = factor ~ rep("*" ~ factor | "/" ~ factor)
```

The `term` returns a sequence of two elements: a result of the `factor` parser, and a result of the `rep` parser. The first one should be already a number (we rewrote the result of `factor` from String to Double in the previous step), the second is a List (each `rep` parser returns a List). So we can start rewriting by decomposing the sequence as follows:

```
term   = factor ~ rep("*" ~ factor | "/" ~ factor) ^^ { case number ~ list => ...
```

[5] More precisely, the parsing result is a container that *contains* that string as its "main" datum. Besides that it contains other data such as a reference to the next portion of the input. Nevertheless the result type is parameterized with the type of its main datum, and other data are mostly not important on the level of parser composition. So while speaking about parsing results we will ignore existence of that other data.

Now lets look closer at the list. Its elements are again sequences that consist of two elements: a String and a number. If the string equals "*", then the previous number (i.e. the number that accumulates the result) should be multiplied by the current one. If it is a "/", then the previous number should be divided by the current one. We can express this logic with a list folding and pattern matching on list elements:

```
term   = factor ~ rep("*" ~ factor | "/" ~ factor) ^^ { case number ~ list =>
         (number /: list) { case (acc, "*" ~ nextNum) => acc * nextNum
                            case (acc, "/" ~ nextNum) => acc / nextNum }
```

Note that we used the *left* folding of the list `(number /: list) {...}`, which can also be expressed with `list.foldLeft(number){...}`, because of the *left-associativity* of "*" and "/" operations in math.

The `expr` rule can be redefined analogously:

```
expr   = term ~ rep("+" ~ term | "-" ~ term) ^^ { case number ~ list =>
         (number /: list) { case (acc, "+" ~ nextNum) => acc + nextNum
                            case (acc, "-" ~ nextNum) => acc - nextNum }
```

These pretty verbose definitions make the grammar poorly readable though. With help of `chainl1` combinator (section 3.2.3) they can be significantly reduced.

3.2.2 ~> and <~ combinators

Due to rewriting of the result of the `floatingPointNumber` parser to Double in the `factor` rule (section 3.2.1.1), we have to ensure that each alternative in `factor` also returns a Double. Otherwise we will not be able to perform computations.

Consider the original `factor` rule:

```
factor = floatingPointNumber | "(" ~ expr ~ ")"
```

The result of the recursively defined `expr` should be a Double (after we have succeeded with the current issue). So the only remaining task is to strip parentheses from the sequence `"(" ~ expr ~ ")"`. This can be done with the already known `^^` combinator:

```
"(" ~ expr ~ ")" ^^ {case _ ~ e ~ _ => e}
```

But there is also another, more concise way: if in the paar `p ~ q` only the left, or only the right part have to be retained for the further processing, then the combinators "<~", or "~>" respectively can be used instead of "~". The arrow points to the part that should be retained. So we can simply write:

```
"(" ~> expr <~ ")"
```

instead of above. The complete `factor` rule becomes:

```
factor = floatingPointNumber ^^ {_.toDouble} | "(" ~> expr <~ ")"
```

Using ~> and <~ combinators requires understanding of precedences of combinators to avoid subtle errors (not an issue here). Section 4.2 discusses this question in details.

3.2.3 The `chainl1` combinator

Parsing a chain of elements separated with some other elements, which prescribe the functions to combine the elements of the first type, is a frequently occuring parsing task. Such functionality is required, for example, to implement operators in nearly all programming languages.

In our case (section 3.2.1.2), factors are separated with '*' and '/' that prescribe multiplication and division functions, respectively. Similarly, terms are separated with '+' and '-' that dictate addition and subtraction functions. We have seen that this can be implemented using `^^` combinator, yet the grammar definition becomes too verbose.

The predefined `chainl1` combinator allow to make such definitions more compact via specifying only the parser for (possibly repeated) main elements, and the mapping from separators to the functions belonging to that separators. Applying the functions to the parsed elements will be performed inside `chainl1` combinator, thus freeing the grammar of the boilerplate code.

The previous version of the `expr` parser:

```
expr   = term ~ rep("+" ~ term | "-" ~ term) ^^ { case number ~ list =>
         (number /: list) { case (acc, "+" ~ nextNum) => acc + nextNum
                            case (acc, "-" ~ nextNum) => acc - nextNum }
```

reduces using `chainl1` combinator to:

```
expr   = chainl1(term, "+" ^^^ Add | "-" ^^^ Sub)
```

where `term` is the parser for main elements, `Add` and `Sub` are references to arithmetic functions that we will define later, and "`^^^`" is a new combinator explained below in section 3.2.4.

Formally, the `chainl1` combinator takes two *parsers* as arguments: the first one with some parsing result of type "`T`", and the second one whose parsing result is a *function* of type "`(T,T) => T`". That function will be applied to the consecutive results returned by `p`, like a left fold. Of course, the function is applied only if `p` succeeded more than once, otherwise the result of a single `p` is returned. And `p` *must* succeed at least once, as is indicated by the '`1`' in the parser name. The '`l`'[6] means that the chain of `p`s will be concatenated from the *left* to the right, producing a left-associated grouping. And if we want a right-associated grouping, then there is the `chainr1` combinator implemented analogous to the `chainl1`.

Here is the current state of our arithmetic parser:

```
trait ArithParser extends JavaTokenParsers {
  def expr: Parser[Double] = chainl1(term,   "+" ^^^ Add | "-" ^^^ Sub)
  def term                 = chainl1(factor, "*" ^^^ Mul | "/" ^^^ Div)
  def factor               = floatingPointNumber ^^ {_.toDouble} | "(" ~> expr <~ ")"
}
```

Note that the overall result is of type `Double`.

3.2.4 The ^^^ combinator

The predefined `^^^` combinator is used to *directly replace* the previous parsing result with a given *value*, as in:

```
"+" ^^^ Add
```

```
"true" ^^^ true
```

The difference to the `^^` combinator is that the last takes a *function* that converts the previous parsing result to some other value (using the the previous parsing result as its input argument). In contrast, the `^^^` combinator simply takes a *value* that directly replaces the previous parsing result, effectively discarding it. The `^^^` combinator is useful in situations where the previous result is always the same, as in case of the "+" parsers whose result is always the string "+".

[6] You probably see that '`1`' and '`l`' are not good distinguishable when written together. With a capitalized '`L`', the name of the combinator would be better readable: "`chainL1`".

Why not to overload the `^^` combinator One might wonder, why not to overload the `^^` combinator to take either a function of type "`T => U`" (where "`T`" denotes the result type of the previous parser) or a value of type "`U`", rather than introducing a new combinator. There is no problem to define such two overloaded versions of the `^^` method.

When calling such method with a function conforming to the type "`T => U`" (i.e. when the parameter type of the function equals to the result type of the previous parser) as a parameter, the Java runtime system will dispatch the call to the overloaded version of `^^` that takes a function (since this version is more specific for the given case than another one); the combinator will then apply that function to the previous parsing result to compute a new value. When passing some value that is not a function, or is a function that is not conforming to the type "`T => U`", then the call will be properly dispatched to the other (less specific) overloaded version. So far, so good.

The only issue seems to be in the case where we would want to *replace* the previos parsing result *with a function* that conforms to the type "`T => U`". We would not be able to do that with an overloaded `^^` combinator. Hence two different names: `^^` and `^^^`.

3.2.5 Alternative notation for `chainl1`

It should be mentioned that there is also an alternative short notation for `chainl1`:

```
term * ("+" ^^^ Add | "-" ^^^ Sub).
```

This notation has in our opinion some drawbacks:

- The common meaning of "`*`" is "zero or more". In this case it works as "one or more", which can be misleading.
- There is no similar notation for `chainr1`.
- The names `chainl1` / `chainr1` express their meanings more explicitly.

Therefore we prefer here the longer notation `chainl1` / `chainr1`.

3.2.6 Definitions of arithmetic functions

The only remaining task is to define the arithemtic functions `Add`, `Sub`, `Mul`, `Div`, which will perform the actual computation.

Inline definitions The first attempt results in the following definition:

```
trait ArithParser extends JavaTokenParsers {
  def expr: Parser[Double] = chainl1(term,   "+" ^^^ {_+_} | "-" ^^^ {_-_})
  def term                 = chainl1(factor, "*" ^^^ {_*_} | "/" ^^^ {_/_})
  def factor               = floatingPointNumber ^^ {_.toDouble} | "(" ~> expr <~ ")"
}
```

where the expression "`{_+_}`" should mean a short form of the function literal "`{(a,b) => a+b}`". Other function expressions are analogous.

Unfortunately, this definition is not working. The compiler issues the following errors:

```
error: missing parameter type for expanded function ((x$5, x$6) => x$5.$times(x$6))
def term                 = chainl1(factor, "*" ^^^ {_*_} | "/" ^^^ {_/_})
                                               ^

error: missing parameter type for expanded function ((x$1, x$2) => x$1.$plus(x$2))
def expr: Parser[Double] = chainl1(term,   "+" ^^^ {_+_} | "-" ^^^ {_-_})
                                               ^
```

What is happening here? At first, the compiler expands the shortcuts of function literals into their longer forms, i.e. `{_*_}` becomes `{(a,b) => a*b}`. The next step would be to infer the type of the function: types of parameters and type of result. And here the compiler fails. This might seem strange.

Given the signature of the `chainl1`-combinator:

```
def chainl1[T](p: => Parser[T], q: => Parser[(T, T) => T]): Parser[T]
```

and the concrete expression (a simplified form from above that leads to the same error as well):

```
chainl1(factor, "*" ^^^ {_*_}),
```

where `factor` is known to be of type `Parser[Double]`, the compiler could conclude that the type of the second parameter "`"*" ^^^ {_*_}`" must be `Parser[(Double, Double) => Double]`. Since the `^^^` combinator directly replaces the previous parsing result, the type of its argument "`{_*_}`" must be `(Double, Double) => Double`. But obviously the compiler utilizes another type inferencing strategy. Perhaps also another considerations, which prevents such type inferencing, are taken into account by compiler.

Therefore we should specify the parameter types in our functions explicitly:

```
trait ArithParser extends JavaTokenParsers {
  def expr: Parser[Double] = chainl1(term,   "+" ^^^ {(a: Double, b: Double) => a + b}
                                           | "-" ^^^ {(a: Double, b: Double) => a - b})
  def term                 = chainl1(factor, "*" ^^^ {(a: Double, b: Double) => a * b}
                                           | "/" ^^^ {(a: Double, b: Double) => a / b})
  def factor               = floatingPointNumber ^^ {_.toDouble} | "(" ~> expr <~ ")"
}
```

This works and computes the result as expected:

```
> scala ArithParserCLI "10.5 - 4*2"
input:  10.5 - 4*2
output: [1.11] parsed: 2.5
```

Separate definitions The code would look more clean and better readable, if the function definitions were separated from the grammar.

Here is the revised parser definition:

```
trait ArithParser extends JavaTokenParsers {
  def expr: Parser[Double] = chainl1(term,   "+" ^^^ Add | "-" ^^^ Sub)
  def term                 = chainl1(factor, "*" ^^^ Mul | "/" ^^^ Div)
  def factor               = floatingPointNumber ^^ Number | "(" ~> expr <~ ")"

  val Add = (a: Double, b: Double) => a + b
  val Sub = (a: Double, b: Double) => a - b
  val Mul = (a: Double, b: Double) => a * b
  val Div = (a: Double, b: Double) => a / b
  val Number = (a: String) => a.toDouble
}
```

Inline vs. separate definitions: performance considerations Would separate definitions also result in a better performance as compared to inline versions? The point is that the `^^^` combinator takes its parameter *by name*. So at a first look it *might* seem that the function objects were generated anew for

each occurrence of any operator in the input. Though the by-name parameter is cached in a local `lazy val` inside the `^^^` method[7]. Therefore it will be evaluated only once in either case. So the performance should be equal in both cases.

3.2.7 The final touch

The function definitions can be a little bit shorter if using the *type ascription* notation:

```
val Add = (_:Double) + (_:Double).
```

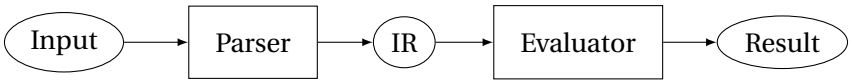
This notation also allows to abstract over names of the function parameters (note the "_"s instead of concrete names), which is good, as the parameter names have no semantic meaning here. What is really matters is only the operation itself (including associated types). This is the final touch to our arithmetic parser that directly evaluates the user input:

```
import util.parsing.combinator.JavaTokenParsers

trait ArithParser extends JavaTokenParsers {
  def expr: Parser[Double] = chainl1(term,   "+" ^^^ Add | "-" ^^^ Sub)
  def term                 = chainl1(factor, "*" ^^^ Mul | "/" ^^^ Div)
  def factor               = floatingPointNumber ^^ Number | "(" ~> expr <~ ")"

  val Add = (_:Double) + (_:Double)
  val Sub = (_:Double) - (_:Double)
  val Mul = (_:Double) * (_:Double)
  val Div = (_:Double) / (_:Double)
  val Number = (_:String).toDouble
}

object ArithParserCLI extends ArithParser {
  def main(args: Array[String]) {
    for (arg <- args) {
      println("input:  " + arg)
      println("output: " + parseAll(expr, arg))
    }
  }
}
```

Listing 3.2: Parser that directly evaluates the input

3.3 Compiler, or Working with intermediate representations

Direct evaluation is a concise and convenient method, but sometimes it is preferable to decouple the evaluation from parsing. This implies existence of some intermediate data structure *(intermediate representation, IR)*, which at the beginning represents the parsing result and then, possibly modified in between, gets passed to the evaluator:



The reasoning behind this schema can be, for instance:

[7] https://github.com/scala/scala/blob/v2.9.1/src/library/scala/util/parsing/combinator/Parsers.scala#L333-343

- a better modularity (parser and evaluator can be independently developed, compiled, tested, and replaced with another implementations),
- the need to perform analysis and rewrite of the parsing results in multiple independent steps (this is very common for compilers).

The fact of existing of an IR structure that can be evaluated independently of the original input, turns the parser into a *compiler*. The evaluator represents the *target machine*, which can execute a program denoted in the specific IR language. The IR acts as an interface between the parser and the evaluator. From now on two "times" can be distinguished: the *compile-time* (parser) and the *run-time* (evaluator).

An intermediate representation often takes the form of an *abstract syntax tree (AST)*, a hierarchical structure constituted by nodes. The word "abstract" stands for discarding of semantically irrelevant parts from a concrete representation (as found in the input).

3.3.1 Concrete vs abstract syntax

The concrete syntax of the input string "`10.5 - 4*2`" can be represented by the following abstract one: `Sub(Number(10.5),Mul(Number(4),Number(2)))`. The abstract syntax may represent multiple different concrete representations, such as:

- the original notation,
- a notation like the original with additional (or removed) whitespace chars or parentheses, e.g. "`(10.5-( (4*( 2 )) ))`",
- the same expression in the prefix ("polish") notation: "`- (* 4 2) 10.5`",
- the same expression in the postfix ("reverse polish") notation: "`10.5 4 2 * -`",
- the same expression expressed in words in some natural language,

and so on. Alternatively, we could also choose another abstract representation, for example, the postfix notation.

In fact, *the abstract representation can be viewed as standardizing of one arbitrarily choosed concrete representation.*

3.3.2 AST definition

A convenient way to define an AST in Scala is to use a hierarchy of *case classes* with a *sealed* abstract class at the top.

Let's look at the AST definition for the arithmetic parser:

```
trait ArithAST {
  sealed abstract class Expr
  case class Add(e1: Expr, e2: Expr) extends Expr
  case class Sub(e1: Expr, e2: Expr) extends Expr
  case class Mul(e1: Expr, e2: Expr) extends Expr
  case class Div(e1: Expr, e2: Expr) extends Expr
  case class Number(e: String) extends Expr
}
```

Listing 3.3: AST definition for the arithmetic parser

Case classes, powered by pattern matching, provide a convenient notation for *creation* and *decomposition* of nodes. For example, a node that represents an addition of "2" and "3":

- can be created (using the above AST definition) via construct `Add(Number("2"), Number("3"))`;
- can be decomposed with pattern matching `case Add(Number(x), Number(y)) => ...`, where the variables x and y will be bound to the actual content of the `Number`-subnodes (strings "2" and "3" in this example).

The "sealed" modifier on the base class restricts the immediate subtypes to be in the current source file only. Besides the clarity for the programmer ("which subclasses should be handled in the evaluator?") this make possible an automatic control by the Scala compiler, whether all subtypes of the sealed class are handled in the evaluator. The compiler will warn if pattern matching on subtypes of `Expr` is not exhaustive, ensuring that no case is forgotten. This is a very useful feature, especially if the AST gets changed during development.

The main node type in our AST is `Expr`, all other node types inherit from it. The subtypes are the four arithmetical operations and the type for number literals.

The `Number` node takes a `String`, which represents the number, as a parameter. Thus the result of the `floatingPointNumber` parser (a string) can be passed directly to this node. Alternatively, we could define the node to take a `Double`, and convert the parsed string to a `Double` in the parser, before the node creation. Here we decided to leave this task to the evaluator.

3.3.3 Building the AST

The only parser's task now is to build the AST for the given input. To create AST-nodes we use the same base technique as for the direct evaluation: rewriting of parsing results with `^^` and `^^^` combinators. The difference is that instead of supplying the functions that do actual arithmetic computations, we pass the functions that create AST nodes:

```
trait ArithParser extends JavaTokenParsers with ArithAST {
  def expr: Parser[Expr] = chainl1(term,   "+" ^^^ Add | "-" ^^^ Sub)
  def term               = chainl1(factor, "*" ^^^ Mul | "/" ^^^ Div)
  def factor             = floatingPointNumber ^^ Number | "(" ~> expr <~ ")"
}
```

Names of case classes as functions The parser code, as written above, is working and can build AST nodes. But where are the *functions* `Add`, `Sub`, `Mul`, `Div`, and `Number` specified? Actually, we defined those names only as *case classes*? Here is an explanation. For each case class Scala compiler creates a *companion object* with the same name as the case class. This companion object has an `apply`-method with the same parameters as the constructor parameters in the case class definition. Besides that, the companion object extends formally the appropriate `FunctionN` class (for example, `Function2` for two parameters). Thus *the companion object of a case class is a function!* This allows to use the companion object's name everywhere when a function of a corresponding arity is expected. Therefore the names `Add`, `Sub`, `Mul`, `Div`, and `Number` are references to the automatically generated companion objects of case classes from AST.

The complete program code is in listing 3.4.

```
trait ArithAST {
  sealed abstract class Expr
  case class Add(e1: Expr, e2: Expr) extends Expr
  case class Sub(e1: Expr, e2: Expr) extends Expr
```

```
  case class Mul(e1: Expr, e2: Expr) extends Expr
  case class Div(e1: Expr, e2: Expr) extends Expr
  case class Number(e: String) extends Expr
}

import util.parsing.combinator.JavaTokenParsers

trait ArithParser extends JavaTokenParsers with ArithAST {
  def expr: Parser[Expr] = chainl1(term,   "+" ^^^ Add | "-" ^^^ Sub)
  def term               = chainl1(factor, "*" ^^^ Mul | "/" ^^^ Div)
  def factor             = floatingPointNumber ^^ Number | "(" ~> expr <~ ")"
}

object ArithParserCLI extends ArithParser {
  def main(args: Array[String]) {
    for (arg <- args) {
      println("input:  " + arg)
      println("output: " + parseAll(expr, arg))
    }
  }
}
```

Listing 3.4: Parser that builds the AST representation of the input

And here is an example of usage:

```
> scala ArithParserCLI "10.5 - 4*2"
input:  10.5 - 4*2
output: [1.11] parsed: Sub(Number(10.5),Mul(Number(4),Number(2)))
```

Note how the output of result has changed. The result "`Sub(Number(10.5),Mul(Number(4),Number(2)))`" means that the overall expression is a subtraction, which is composed of the number `10.5` and of the multiplication of `4` and `2`.

3.3.4 "Mutating" parser

The actual code of the trait `ArithParser` (listing 3.4) is very similar to the direct-evaluating parser (listing 3.2). Except for the return type of the `expr` parser (`Expr` vs `Double`), the code is identical!

This opens interesting opportunities: we can define a generic parser, which can become an interpreter or a compiler ("mutating"), depending on the function definition that will be mixed in: those for direct evaluation or those for AST building. Moreover, the choice can be made dynamically, based on the user input. The full code of such "mutating" parsers is shown in listing 3.5.

An interesting point, in both cases we implement abstract methods from the `ArithParser` trait (see under "abstract semantic action" in code), and in both cases this is not a usual implementation of abstract methods:

- in trait `DirectEvaluation` *function values* implement them,
- in trait `ASTBuilding` *case classes* (more precisely, `apply` methods of their *companion objects*) implement them.

What a flexibility of Scala!

And here is an example of interaction with the user:

```
import util.parsing.combinator.JavaTokenParsers

trait ArithParser extends JavaTokenParsers{
  type T // return type of the expr-parser
  def expr: Parser[T] = chainl1(term,   "+" ^^^ Add | "-" ^^^ Sub)
  def term            = chainl1(factor, "*" ^^^ Mul | "/" ^^^ Div)
  def factor          = floatingPointNumber ^^ Number | "(" ~> expr <~ ")"

  // abstract semantic actions
  def Add: (T,T) => T
  def Sub: (T,T) => T
  def Mul: (T,T) => T
  def Div: (T,T) => T
  def Number: String => T
}

trait DirectEvaluation {
  type T = Double
  val Add = (_:Double) + (_:Double)
  val Sub = (_:Double) - (_:Double)
  val Mul = (_:Double) * (_:Double)
  val Div = (_:Double) / (_:Double)
  val Number = (_:String).toDouble
}

trait ASTBuilding {
  type T = Expr
  sealed abstract class Expr
  case class Add(e1: Expr, e2: Expr) extends Expr
  case class Sub(e1: Expr, e2: Expr) extends Expr
  case class Mul(e1: Expr, e2: Expr) extends Expr
  case class Div(e1: Expr, e2: Expr) extends Expr
  case class Number(e: String) extends Expr
}

object Interpreter extends ArithParser with DirectEvaluation
object Compiler extends ArithParser with ASTBuilding

object Arith {
  def main(args: Array[String]) {
    val arg = args.toList
    val parser: ArithParser = if (arg.head == "eval") {
      println("Now I'm interpreter!"); Interpreter
    } else {
      println("Now I'm compiler!"); Compiler
    }
    arg.tail foreach { x =>
      println("input:  " + x)
      println("result: " + parser.parseAll(parser.expr, x))
      println()
    }
  }
}
```

Listing 3.5: “Mutating” arithmetic parser

```
> scala Arith eval "10.5 - 4*2"
Now I'm interpreter!
input:  10.5 - 4*2
result: [1.11] parsed: 2.5

> scala Arith compile "10.5 - 4*2"
Now I'm compiler!
input:  10.5 - 4*2
result: [1.11] parsed: Sub(Number(10.5),Mul(Number(4),Number(2)))
```

3.3.5 Separate evaluation of the AST

A hierarchy of AST-nodes that represents an arithmetical expression can be evaluated independently of the parser. The code of the evaluator is shown in listing 3.6.

```
trait ArithEvaluator extends ArithAST {
  def eval(e: Expr): Double = e match {
    case Add(e1: Expr, e2: Expr) => eval(e1) + eval(e2)
    case Sub(e1: Expr, e2: Expr) => eval(e1) - eval(e2)
    case Mul(e1: Expr, e2: Expr) => eval(e1) * eval(e2)
    case Div(e1: Expr, e2: Expr) => eval(e1) / eval(e2)
    case Number(e: String) => e.toDouble
  }
}
```

Listing 3.6: Evaluator of arithmetic expressions

The evaluation algorithm is straightforward: we start with the root class of the AST-hierarchy and go with the `case`-construct through all subclasses accordingly to the AST definition. Each node is decomposed into its parts, the parts are evaluated, and then merged again with the operation that corresponds to the node type ('+' for `Add`, '−' for `Sub`, etc.). The `Number`-node is a special case, since this is an atomic expression that does not consist of any subexpressions.

The common source of errors in various parser generators is forgetting to handle some node type after a change in the AST definition. We make an experiment, and "forget" to handle the `Div`-case via commenting out the corresponding line. What happens? A compiler warning: "match is not exhaustive!" This is the effect of the "seal" modifier on the corresponding superclass in the AST definition.

Now we will compose the full program, which should first try to parse the input into an AST representation, and then, if the previous step was successfull, evaluate the AST. The code is shown in listing 3.7.

Example of usage:

```
input: 10.5 - 4*2
parse result: [1.11] parsed: Sub(Number(10.5),Mul(Number(4),Number(2)))
AST: Sub(Number(10.5),Mul(Number(4),Number(2)))
evaluation result: 2.5
```

Note that the AST output and the evaluation will only be performed if parsing was successful. In case of syntactical errors in the input, a message explaining the error is issued:

```
input: 10.5  - 4*2)
parse result: [1.12] failure: string matching regex `\z' expected but `)' found
```

```
trait ArithAST {
  sealed abstract class Expr
  case class Add(e1: Expr, e2: Expr) extends Expr
  case class Sub(e1: Expr, e2: Expr) extends Expr
  case class Mul(e1: Expr, e2: Expr) extends Expr
  case class Div(e1: Expr, e2: Expr) extends Expr
  case class Number(e: String) extends Expr
}

import util.parsing.combinator.JavaTokenParsers

trait ArithParser extends JavaTokenParsers with ArithAST {
  def expr: Parser[Expr] = chainl1(term,   "+" ^^^ Add | "-" ^^^ Sub)
  def term               = chainl1(factor, "*" ^^^ Mul | "/" ^^^ Div)
  def factor             = floatingPointNumber ^^ Number | "(" ~> expr <~ ")"
}

trait ArithEvaluator extends ArithAST {
  def eval(e: Expr): Double = e match {
    case Add(e1: Expr, e2: Expr) => eval(e1) + eval(e2)
    case Sub(e1: Expr, e2: Expr) => eval(e1) - eval(e2)
    case Mul(e1: Expr, e2: Expr) => eval(e1) * eval(e2)
    case Div(e1: Expr, e2: Expr) => eval(e1) / eval(e2)
    case Number(e: String) => e.toDouble
  }
}

object ArithParserCLI extends ArithParser with ArithEvaluator {
  def main(args: Array[String]) {
    for (arg <- args) {
      println("input:  " + arg)
      val parseResult = parseAll(expr, arg)
      println("parse result: " + parseResult)

      parseResult match {
        case Success(ast, _) =>
          println("AST: " + ast)
          println("evaluation result: " + eval(ast))
        case _ =>
          // no action if no success!
      }
    }
  }
}
```

Listing 3.7: Arithmetic parser combined with evaluator

```
10.5  - 4*2)
          ^
```

The actual error message is not very helpful. The ways of improving will be discussed in section 4.1.

3.3.6 AST that can evaluate itself

The structure of the actual evaluator closely resembles the structure of the AST definition. This is a sign for an optimization potential. The idea is simple: *Why not to define the `eval`-method immediately in the corresponding node types?* And this is in fact feasible. Look at the code in listing 3.8: the trait `SmartAST` contains definitions of nodes with `eval`-methods. Note that the root node type of the AST is parameterized now with the target type of evaluation (`Double` in our case). Evaluating the AST is now as easy as to call `eval` on the root node instance: "`ast.eval`". That's all, no separate evaluator required.

Example of usage:

```
> scala Arith "10.5 - 4*2"
input:  10.5 - 4*2
parse result: [1.11] parsed: Sub(Number(10.5),Mul(Number(4),Number(2)))
AST: Sub(Number(10.5),Mul(Number(4),Number(2)))
evaluation result: 2.5
```

Along `eval`, it is possible to add other actions that can be useful on AST nodes, such as various transformations or pretty printing of arithmetical expressions.

A self-evaluating-AST might seem convenient in this very simple case, but it's a good idea to keep the AST definition and the evaluation logic distinct, especially if they become more complex (consider, for example, working with context upon implementing closures, or promoting data types during type-checking).

Self type annotation The code in listing 3.8 uses yet another Scala feature: the *self type annotation*. Consider the line:

```
trait ArithParser extends JavaTokenParsers { this: ArithAST =>
```

The self type annotation "`this: ArithAST =>`" means, that the concrete class that will extend `ArithParser` have to be also an `ArithAST`. That is, the self type annotation *specifies dependencies* of the current trait on other traits. Now it is possible to use all members that are defined in the `ArithAST` in the body of the `ArithParser` as well. Those members are abstract semantic actions for building AST nodes: `Add`, `Sub`, `Mul`, `Div`, `Number`. Trait `SmartAST` provides the concrete implementations. Everything is wired together in the definition of the main object:

```
object Arith extends ArithParser with SmartAST
```

```
trait ArithAST { // Interface for AST subtypes
  type T
  def Add: (T,T) => T        // Abstract
  def Sub: (T,T) => T        // semantic
  def Mul: (T,T) => T        // actions
  def Div: (T,T) => T        // to build
  def Number: String => T    // AST nodes.
}

trait SimpleAST extends ArithAST {
  type T = Expr
  sealed abstract class Expr
  case class Add(a:T, b:T) extends T
  case class Sub(a:T, b:T) extends T
  case class Mul(a:T, b:T) extends T
  case class Div(a:T, b:T) extends T
  case class Number(e:String) extends T
}

trait SmartAST extends ArithAST {
  type T = Expr[Double]
  sealed abstract class Expr[+T] {def eval: T}
  case class Add(a:T, b:T) extends T {def eval = a.eval + b.eval}
  case class Sub(a:T, b:T) extends T {def eval = a.eval - b.eval}
  case class Mul(a:T, b:T) extends T {def eval = a.eval * b.eval}
  case class Div(a:T, b:T) extends T {def eval = a.eval / b.eval}
  case class Number(e:String) extends T {def eval = e.toDouble}
}

import util.parsing.combinator.JavaTokenParsers

trait ArithParser extends JavaTokenParsers { this: ArithAST =>
  def expr: Parser[T] = chainl1(term,   "+" ^^^ Add | "-" ^^^ Sub)
  def term            = chainl1(factor, "*" ^^^ Mul | "/" ^^^ Div)
  def factor          = floatingPointNumber ^^ Number | "(" ~> expr <~ ")"
}

object Arith extends ArithParser with SmartAST {
  def main(args: Array[String]) {
    for (arg <- args) {
      println("input:  " + arg)
      val parseResult = parseAll(expr, arg)
      println("parse result: " + parseResult)

      parseResult match {
        case Success(ast, _) => println("AST: " + ast)
                                println("evaluation result: " + ast.eval)
        case _ => // no action if no success!
      }
    }
  }
}
```

Listing 3.8: Arithmetic parser with a “smart” AST

Chapter 4

Common Problems

Parsers do not always work "out of the box" as they should (by intention of programmers). This section describes some common problems, investigates their origins, and shows the ways on improving the situation.

4.1 Generating helpful error messages

4.1.1 Issue

Error messages, automatically produced by parsers in case of an illegal input, are often inadequate.

The standard advise to improve the situation is to add an explicit `failure("error message")` to each parser as the (last) alternative. For instance, the `factor`-parser could be rewritten as follows:

```
def factor = ( floatingPointNumber ^^ Number
             | "(" ~> expr <~ ")"
             | failure("factor expected")
             )
```

Unfortunately (and surprisingly) this does not always work as desired. Consider the following input:

```
input:  10.5 - x*2
output: [1.8] failure: `(' expected but ` ' found // Scala 2.8.x branch
output: [1.8] failure: `(' expected but `x' found // Scala 2.9.x and 2.10.x branches

10.5 - x*2
       ^
```

The new explicit `failure` rule has been ignored. Why is this happening?

4.1.2 Explanation

The first thing to know is that the primitive regex- and string literal parsers (as "`(`" in the example above) do automatically skip whitespace in the input *before* trying to recognize the given pattern. Thereafter, if the pattern has not been matched, the parser reports a failure at the first non-whitespace position (*after* the skipped whitespace) in the input, but the current parsing position remains at the initial position (*before* the skipped whitespace). While the most combined parsers (such as `floatingPointNumber`) are ultimately based on these primitive parsers and therefore behave accordingly concerning the whitespace and the current position, the special `failure`-parser is different: it fails "right now", on the currrent parsing position, without skipping whitespace.

The second thing is the logic of failure recording. There is only one dedicated (global) variable that holds only one failure. If a new failure is reported during parsing at a later *or the same* position in the input, then this new failure overwrites the old one. As such, a good and simple mechanism that ensures that the final failure message has the largest position across all failures encountered during parsing.

But see how these implementation details play together in the aforementioned example (recognizing of erroneous input "`10.5 - x*2`"):

1) After a successful recognizing of the minus-operator, the current position is 7 (a whitespace-symbol):

```
pos:      123456789...
--------------------
input:    10.5 - x*2
cur.pos:        ^
```

2) Now, the `floatingPointNumber` parser is tried and fails, recording its failure at the position 8 (after the skipped whitespace). The current parsing position is still 7.

3) "`)`" parser is tried and fails at the position 8 just like the `floatingPointNumber` *overwriting the last failure value* (that was coming from the `floatingPointNumber`) with its own failure. The current position is still 7.

4) The last alternative in the `factor` rule is tried: an explicit `failure`-parser. It fails on the current position (7), without skipping whitespace. Since the last recorded failure (from the "`)`"-parser in step 3) belongs to a greater position (8), the failure-message from the `failure`-parser is completely ignored.

5) The final failure message is the last recorded one, that is the message from the failed "`)`" parser (step 3).

4.1.3 Proof

The above explanation can be proved with a simple test: delete the whitespace between "`-`" and "`x`". Now all three failures should happen on the same position, and the last one (from the explicit `failure`-parser) should be emitted. The test shows that the hypothesis is correct:

```
input:  10.5 -x*2
output: [1.7] failure: factor expected

10.5 -x*2
      ^
```

Requiring a whitespace-free input is not an option, of course. How to work around this problem?

4.1.4 Solution

A solution is to *force the `failure`-rule to skip the whitespace* too, like string and regex parsers do. This can be achieved in multiple ways:

- prepend each use of `failure` parser with an "empty string" parser, as shown here:

```
def factor = ( floatingPointNumber ^^ Number
             | "(" ~> expr <~ ")"
             | "" ~> failure("factor expected")
             )
```

Note the "" ("empty string" parser) before the `failure`. The solely purpose of the ""-parser is to skip whitespace. Now the `failure` parser starts from the same position as string- or regex parsers in other alternatives.

- override the definition of the `failure` parser using the same principle as above:

  ```
  override def failure(msg: String) = "" ~> super.failure(msg)
  ```

  and use `failure` parsers as usual.

- override `failure` and reimplement its logic to explicitly skip whitespace like regex and string parsers do.

- refactor the parsing library to skip whitespace *after* terminals, instead of doing it *before* in the current implementation of the `RegexParsers` trait.

With any of these variants the original input is parsed as expected issuing a reasonable error message:

```
input:  10.5 - x*2
output: [1.8] failure: factor expected

10.5 - x*2
       ^
```

4.2 Precedences of infix combinators

4.2.1 Operators and infix notation

There are no real operators in Scala. Everything that looks like an operator is a method used in the *infix (operator) notation*. The infix notation (sometimes also called "dotless") means that instead of "`object.method(parameter)`" we can write "`object method parameter`". Even the familiar arithmetic operations are nothing else as methods in Scala, which are defined on various numeric types.

Just like multiple method calls can be chained (provided that the return type of each method has the next chained method as its member), the infix notation can be chained as well: instead of "`object.method(parameter).method2(parameter2)...`" we can write "`object method parameter method2 parameter2...`". For example, these notations are equivalent: "`5 + 3 - 2`" and "`5.+(3).-(2)`".

Parsing combinators are methods in Scala. Many of them are almost exclusively used in the infix notation. Consider:

```
"var" ~ id ~ "=" ~ expr ^^ {case _ ~ id ~ _ ~ e => VarAssignment(id,e)}
```

The combinators "~" and "^^" are used here as infix operators. (Note that the "~" *in the case expression* is a part of the pattern, not a parser combinator.) For comparison, here is the traditional ("dotfull") method call notation:

```
"var".~(id).~("=").~(expr).^^({case _ ~ id ~ _ ~ e => VarAssignment(id,e)})
```

As with any infix operators we should think about precedences to ensure the correct usage of them.

4.2.2 Precedence rules

Precedence of operators[1] defines how operations are *grouped* in abscence of parentheses. The rules for precedences of infix operators in Scala are defined in the Scala Language Specification (SLS) [Ode11, 6.12.3 Infix Operations]. The precedence is determined by the operator's first character. The following table shows a summary from the SLS[2], extended with concrete method names used for parsing:

	Precedence according to SLS	**Related parser methods**	not in std. lib.
lowest →	*(all letters)*	`into named withFailureMessage`	
	`|`	`| |||`	
	`^`	`^^ ^^^ ^?`	
	`&`		
	`= !`		
	`< >`	`<~ >>`	`<~!`
	`:`		
	`+ -`		
	`* / %`	`*`	
highest →	*(all other special characters)*	`~ ~> ~!`	`~< ~>! ~<!`

Notes to the table:

1) Combinator `withFailureMessage` is introduced in Scala 2.10M1 (first milestone of the future version 2.10).

2) Parser method "`*`" means here the alias to the (binary) `chainl1` combinator [3], not the (unary) postfix "`*`" combinator (see also notes below).

3) The `rep`, `opt`, etc. methods from `Parsers` *trait*[4] (which encloses the `Parser` *class*[5]) are not affected, since they are "global" methods, not methods on `Parser` class, and therefore cannot be used to combine parsers in infix notation.

4) Also not affected are *unary*[6] *postfix* methods from the `Parser` class ("`*`", "`+`", "`?`"), as they inherently cannot be used in the infix notation. Specifics of their usage as unary postfix operators will be explained in section 4.3.

5) Operators on the "not in std. lib." side are not included in the official Scala distribution, but are sometimes used in custom extensions of Scala standard library.

6) Method "`named`" (as well as some other methods not mentioned in the table) is used mostly for inner purposes, not in the user code.

4.2.3 Precedence vs Order of application

Note that *the order of application* of parsers to the input always corresponds to the left-to-right order of parsers in expression, independently of their grouping order (which is based on precedences).

Precedences determine eventually the order, in which the individual parsing *results* are *combined* together.

---

[1] As well as "methods in infix notation".

[2] Note that the current version of the SLS (v2.9) has an error in the operator precedence table (https://issues.scala-lang.org/browse/SI-5209), which is already corrected here.

[3] https://github.com/scala/scala/blob/v2.9.1/src/library/scala/util/parsing/combinator/Parsers.scala#L402-407

[4] https://github.com/scala/scala/blob/v2.9.1/src/library/scala/util/parsing/combinator/Parsers.scala#L53

[5] https://github.com/scala/scala/blob/v2.9.1/src/library/scala/util/parsing/combinator/Parsers.scala#L191

[6] Sometimes referred to as "*nullary*" (i.e. methods that do not take any parameters)

4.2.4 Effect of precedences in typical usage scenarios

Generally, the symbols for combinators with their associated precedences are well choosed by the parsing library designers. In typical usage scenarios individual parsers often consist of a few alternatives ("|"), where each alternative is a sequence ("~"), that gets rewritten into some value (using "^^" or "^^^"), as in:

```
a ~ b ~ c ^^ Node1 | x ~ y ~ z ^^ Node2 | "(" ~> w <~ ")"
```

This plays very well together, exactly as the user probably expects: the "~" binds tightest, followed by "^^", and then by "|", which binds most loosely. So, no parentheses to change the grouping order are needed.

4.2.5 Effect of precedences in special cases

Care should be taken in the following cases.

4.2.5.1 "Letter-named" operators

The "letter-named" combinators (such as `into`, `named`) have the lowest precedence, that is they bind most loosely. Often this is perfectly suitable, e.g. to give a name to the newly defined parser:

```
def p = a ~ b ~ c ^^ Node1 named "node1-parser"
```

The `name` method will be called on the whole parser "`a ~ b ~ c ^^ Node1`", as probably intended by the user. In cases where such behavior is not appropriate parentheses have to be used.

4.2.5.2 Mixing ^^ and >>

The >> combinator binds more tightly than ^^.

Example 1: "`a ^^ b >> c`" means "`a ^^ (b >> c)`".

Example 2: The following parser (which recognizes a comma-separated list, enclosed in curly braces)

```
def set[T](p: => Parser[T]) = "{" ~! repsep(p, ",") ~ "}" ^^ {case _~x~_ => x} >> mkSet
```

causes the following error message (which is unfortunately not very helpful):

```
error: missing parameter type for expanded function ((x0$1) => x0$1 match {
case $tilde($tilde(_, (x @ _)), _) => x})
```

The problem will be remedied by inserting parentheses around the receiver[7] of the ^^ combinator:

```
def set[T](p: => Parser[T]) = ("{" ~! repsep(p, ",") ~ "}" ^^ {case _~x~_ => x}) >> mkSet
```

4.2.5.3 ~ / ~> and <~ in one parsing expression

This is probably the main source of precedence issues with parser combinators. The combinators ~ / ~> and <~ have different precedences. Namely, ~ / ~> bind more tightly than <~.

[7] Expression that returns an object on which a given method should be called.

Example 1: There is no problem in the expression

```
"(" ~> w <~ ")"
```

from the above example, since the operator that binds more tightly ("~>") preceedes the operator that binds more loosely ("<~"), so that the grouping order coincides with the left-to-right reading order.

Example 2: But consider the following parser:

```
def varDef = id ~ ":" ~ typeRef
```

Suppose we would want to leave only `id` and `typeRef` parts on the right-hand-site to pass them to a function to create an AST node. A naive definition

```
id <~ ":" ~ typeRef
```

wouldn't produce the desired result. According to the precedence rules, this expression will be treated as

```
id <~ (":" ~ typeRef)
```

throwing away all but `id` from the parsing result. A type mismatch (e.g. when the result of the chain should be converted into an AST node via function that takes two parameters: `id`, and `typeRef`), or other kind of compile error is the best that can happen. In worst case the code will compile, but the parser will return wrong result. So, we have to use parentheses properly to enforce the right grouping:

```
def varDef = (id <~ ":") ~ typeRef
```

Example 3: The problem with different precedences of "~>" and "<~" can become very annoying in bigger expressions. Consider the following parser:

```
def relVarDef = "var" ~ id ~ "real" ~ "relation" ~ "{" ~ schema ~ "}" ~ "key" ~ "{" ~ key ~
    "}"
```

To create the corresponding AST node (`RelVarDef(id, schema, key)`) we need only three elements (`id`, `schema`, and `key`) from the 11-elements-chain. How to express this aim? Possible solutions include:

1) Use only "~" combinators, and filter the chain elements in the pattern match expression on the right-hand-side of the ^^ combinator:

```
def relVarDef = "var" ~ id ~ "real" ~ "relation" ~ "{" ~ schema ~ "}" ~ "key" ~ "{" ~
    key ~ "}" ^^ {case _~i~_~_~_~s~_~_~_~k~_ => RelVarDef(i,s,k)}
```

2) Use "~>" / "<~" and parentheses. Many different versions are possible. After some exercise, a probably one of the best versions is:

```
def relVarDef = ("var" ~> id <~ "real" <~ "relation") ~ ("{" ~> schema) <~ "}") ~
    ("key" ~> "{" ~> key <~ "}") ^^ RelVarDef
```

3) Define an alias for "<~" with the same precedence as other two sequence combinators. The alias can be named as "~<":

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

Now the whole expression can be written without parentheses:

```
def relVarDef = "var" ~> id ~< "real" ~< "relation" ~< "{" ~ schema ~< "}" ~< "key"
    ~< "{" ~ key ~< "}" ^^ RelVarDef
```

Unfortunately, none of the proposed solutions is completely satisfying:

- it is not always obvious, how to correctly *write* the code, e.g. how to place parentheses correctly, or where to use which of the sequence combinators;
- it is not always obvious, how to correctly *read* the code, e.g. to see quickly, which parts of the parsed chain will be retained and which wiil be ignored.

4.3 Specifics of postfix combinators

4.3.1 Postfix notation

Methods that do not take any parameters can be used in the *postfix* operator notation in Scala. For example, a well-known `toString` method can also be used in the postfix notation: "`myVar toString`". Another example is the method "`r`" on the `StringLike` class that is mostly used to trigger the implicit conversion of a `String` to a `Regex`[8], as in `"[A-Za-z][0-9A-Za-z]*"r`.

4.3.2 Postfix combinators

Combinator parsing library includes three methods in `Parser` class that can be used in postfix notation: "`*`", "`+`", and "`?`". Each method has also an alias (in the parent `Parsers` trait), which can be used with the same effect. The following table shows these combinators and their usage:

Combinator	Usage as		Alias in the `Parsers` trait
	postfix operator	method call	
*	`r*`	`p.*`	`rep(p)`
+	`p+`	`p.+`	`rep1(p)`
?	`p?`	`p.?`	`opt(p)`

There are two important nuances related to postfix operator notation in Scala, which will be explained below:

1) Postfix operations have *lower precedence* (bind more loosely) than infix operations.

2) A postfix operation is *only allowed at the end* of an infix expression.

[8] A `String` will be implicitly converted to a `StringLike`, then the method "`r`" is called on the resulting object, which returns an instance of `Regex` built up from the underlying `String` instance.

4.3.3 Relative precedence of postfix, prefix, and infix operations in Scala and other languages

In math and many programming languages, including `C`, `C++`, and `Java`, postfix operations have the highest precedence (bind more tightly), followed by prefix, and then by infix operations: `postfix -> prefix -> infix`. See an example expression and its treatments in the table:

Language	Expression	Interpretation
Math / C / C++ / Java	$Z + -x!$	$Z + (-(x!))$

However these languages have a *fixed set* of symbols that denote all three kinds of operations, which is *disjunct* from the set of allowed identifiers. Each expression is also required to be explicitly terminated with a semicolon. This make possible an unambiguous interpretation of expressions.

On the contrary, Scala has a different rule: `prefix -> infix -> postfix`, i.e. postfix operations got the lowest precedence. Notice a different interpretation of the same example expression:

Language	Expression	Interpretation
Scala	$Z + -x!$	$(Z + (-x))!$

But Scala also largely *does not restrict* the set of symbols that denote operations[9]. Every operation *is* an (arbitrary) identifier, that is the name of a method, which can also be used in infix or postfix notation. The semicolons are generally not required. This makes the language very expressive, but also challenging to parse.

4.3.4 Reasoning and explanation of design decisions in Scala

Given that both the names of variables and operations come as identifier tokens from the scanner to the Scala parser, the above example "`Z + -x!`" looks for the parser like a chain of five `id` tokens: "`id id id id id`". In a context-free parsing (which is how Scala parser currently works) it is impossible to distinguish, which `id` denotes which kind of operation: prefix, infix, or postfix.

The situation is even more difficult if the next line formally looks like it could belong to the expression on the previos line. Consider

```
def m(a: Int, b: Int) = {
  val c = a + b!
  -c
}
```

What happens in the body of the method?

Whereas for a human there are probably no ambiguities (compute "`c`" value, then return "`-c`"), how to avoid that the parser recognizes this as

- `val c = a + b! - c`
  (i.e. by combining both lines into one expression)

or as

- `val c = a + b ! (-c)`
  (i.e. by treating "`!`" as an infix operation)?

[9] The only exception are prefix operations, which are restricted to the set `{'+', '-', '!', '~'}`.

To disambiguate such situations, a few rules were introduced in the Scala grammar:

1) The aforementioned **restriction of the set of symbols that denote prefix operations**. If one of the symbols `['+', '-', '!', '~']` is found at the beginning of a (sub)expression, it is unconditionally treated as a prefix operation[10][11]. In such way the problem is narrowed now to infix and postfix operations only.

   Here is the related `PrefixExpr` definition from the Scala grammar:

```
PrefixExpr  ::= ['-' | '+' | '~' | '!'] SimpleExpr
```

2) **Giving the lowest precedence to postfix operations**, thereby allowing them to appear only *at the end* of an (infix) expression. The SLS says: "Postfix operators always have lower precedence than infix operators. E.g. $e_1\, op_1\, e_2\, op_2$ is always equivalent to $(e_1\, op_1\, e_2)\, op_2$." [Ode11, § 6.12.3].

   Here is the related `PostfixExpr` definition from the Scala grammar:

```
PostfixExpr ::= InfixExpr [id [nl]]
```

   where `id` denotes a *postfix* operation.

   As an example, consider the following parser definiton: "`a ~ b*`". At a firtst look, it *might* seem, that this parser first applies "`a`", an then the repetition of "`b`". But this is not correct. According to the relative precedence of postfix and infix operations, the expression will be interpreted as "`(a ~ b)*`".

3) **An infix expression is only allowed to expand to the next line, if the infix operator was *at the end* of the current line**. Consider the following examples:

   This is *one* expression, "`a + b + (-c)`":

```
a + b +
- c
```

   These are *two* expressions, "`a + b`", and "`-c`":

```
a + b
- c
```

   Here is the related `InfixExpr` definition from the Scala grammar:

```
InfixExpr   ::= PrefixExpr
              | InfixExpr id [nl] InfixExpr
```

   where `id` denotes an *infix* operation. This definiton can also be rewritten without left recursion as:

```
InfixExpr   ::= PrefixExpr {id [nl] PrefixExpr}
```

   Note the absence of an optional `nl` (newline) token *before* `id` token.

4) Unfortunately, even with these restrictions, the following situation is still ambiguous:

```
PrefixExpr id [nl]
PrefixExpr
```

[10] https://github.com/scala/scala/blob/v2.9.1/src/compiler/scala/tools/nsc/ast/parser/Parsers.scala#L1408

[11] Excluding the case, where the `'-'` is a part of a numerical literal, https://github.com/scala/scala/blob/v2.9.1/src/compiler/scala/tools/nsc/ast/parser/Parsers.scala#L1411.

Is this a single infix expression (`PrefixExpr id PrefixExpr`) or two expressions: a postfix expression (`PrefixExpr id`), and an infix expression (`PrefixExpr`[12])? As mentioned above, it is impossible to know at the context-free phase whether an `id` represents an infix or a postfix operation.

As a pragmatic solution, ***Scala always treats the `id` in such situations as an infix operation.***

This leads sometimes to undesired behavior though.

Example 1: Consider:

```
val x = 5 toString
println("OK")

error: too many arguments for method toString: ()java.lang.String
       val x = 5 toString
                 ^
```

What is happening here, is that the `println` expression is syntactically a `PrefixExpr`, so that the parser sees:

```
val x = PrefixExpr id nl PrefixExpr
```

and treats that as an infix expression:

```
val x = 5 toString println("OK")
```

First at the typechecking phase the compiler sees that the method `toString` doesn't take parameters, and issues an error message.

Example 2: The same happens also with with postfix combinators. Consider the following parser definition:

```
("a" ~ "b")* ~ "c"
```

which leads to the error:

```
unary ~ is not a member of java.lang.String
```

What is happening here? The compiler tries to treat `*` as an infix operation with the parameter ~"c". The parameter will be typechecked first. But this fails, since the `String` class, to which the expression "c" belongs, has no unary prefix methods named "~". Neither `Parser` class, to instance of which the string could be implicitly converted, has such an unary method. This results in the shown error message.

4.3.5 Practical tips

The shown pitfalls are easy to overcome. It is enough to choose one of the following approaches:

1) If using an expression consisting of a *single postfix operation*, ensure that the next line doesn't start with an expression that can be syntactically treated as a continuation of the current one. For example, if the next line starts with a `val`, `var`, `def`, `class`, or `object` definition, there is no reason to worry.

2) Enclose usages of *postfix operators in parentheses*, e.g. `(("a" ~ "b")*) ~ "c"`, `(digit+)`.

3) Use *method-call notation* instead of postfix operator notation, as in `a ~ b.*` (notice a dot between `b` and `*`).

4) Use the *letter-named equivalents* of the postfix operators: `rep("a" ~ "b") ~ "c"`. They take the same number of characters (as compared to enclosing postfix operators in parentheses) and look less cryptic.

[12] Note that a single `PrefixExpr` is also an `InfixExpr` according to the Scala grammar.

4.4 Performance optimizations

Parser combinators in Scala are lightweight and fast enough for many applications. Still, there is a potential for some performance improvements. We show a few simple approaches.

4.4.1 Avoiding reevaluation of parsers: `lazy val`s instead of `def`s

In examples we mostly used `def`s for parser definitions. A `def` is a method. That means that every reference to the method's name leads to evaluation of the method's body. This could be costly if the method is referenced in multiple places (i.e. the defined parser is used in another parsers), and even more costly if the method's body references several other parsers, which have to be reevaluated in turn.

Clearly, it's better to evaluate the parser only once. This can be achieved with a `val` or a `lazy val` definition instead of a `def`.

In case of `val` one should keep in mind that all parsers referenced in the definition have to be initialized *before and independently* of the current definition. If the programmer arranges the parser definitions in the top-down order, or if some parsers are recursive, then the code will compile fine, but the forward references will not be properly initialized at run time[13].

A `lazy val` definition deffers the body evaluation till the time, when it's eagerly (as opposed to call-by-name passing) referenced. The order of parser definitions does not matter (as opposed to `val`). Obtaining a reference to a `lazy val` variable (e.g. when a parser definition refers to another parser, which is defined as a `lazy val`) goes through a small indirection that checks whether the referenced parser is already initalized, and triggers initialization if it was not. The footprint is very small and can be neglected. Moreover, many combinators cache the computed references to underlying parsers in their local variables, so that they don't need to be computed again.

So, the proposed approach is to use `lazy val`s for parser definitions, for example:

before:

```
def stmt = varDef | varAssign
```

after:

```
lazy val stmt = varDef | varAssign
```

With this change, the body of the parser will be evaluated only once. The overhead of parser reevaluation on its subsequent uses is eliminated. Given an arbitrary complexity of subparsers that can be involved into construction of the current parsers, this approach can significantly improve the runtime characteristics of the parsing program.

4.4.2 Reducing backtracking with `~!` combinator

Normally, if one alternative of the alternation combinator "`|`" fails, then the next alternative will be tried starting from the same (initial) input position. This is called *backtracking*. Generally, combinator parsers in Scala are capable of unlimited backtracking, that makes them also very powerful. Yet too much backtracking negatively influences the runtime characteristics of a parser. Reducing or even complete elimination of backtracking is the general approach to increase the performance of parsers regardless of used parsing techniques.

By factoring out the common parts of alternatives, the parsing can be made *predictive*, so that only one character (or, more commonly, a token) is sufficient to decide, which alternative should be taken.

[13] This problem has been discussed multiple times and is not so simple as it might seem. Here is a recent thread on scala-user http://groups.google.com/group/scala-user/browse_thread/thread/bda2a0a0017342a9/, on scala-internals http://groups.google.com/group/scala-internals/browse_thread/thread/c8c512eac68b6d26/, and a related ticket https://issues.scala-lang.org/browse/SI-4856 (the one that has been left open among a lot of duplicates).

Though by overdoing, this can easily lead to a poorly readable grammar (and parser code) with a negative impact on maintainability.

Fortunately, in Scala we are not forced to take any of extremes. We can decide *flexible*, where the backtracking is acceptable and where it is not.

Let's see how it works by a simple example. Consider the `factor` rule from the arithmetical parser (augmented with an improved error reporting from the section 4.1):

```
def factor = ( floatingPointNumber ^^ Number
             | "(" ~> expr <~ ")"
             | "" ~> failure("illegal start of expression")
             )
```

Now imagine, we have found a `"("` in the input (i.e. the first part of the second alternative succeeded). Which next input is allowed to follow? Clearly, it only may be the rest of a parenthesed expression. That is the next expected input should be an expression (`expr`), and then the closing parenthesis (`")"`) according to the parser definition `"(" ~> expr <~ ")"` . *If this sequence not succeeds, it makes no sence to check whether any other alternative could succeed on this input; such an input cannot be legal anymore; the parsing must be terminated with an error message.*

The ~! combinator[14] expresses exactly the above idiom. It can be used as follows:

```
"(" ~! expr ~! ")" ^^ {case _ ~ e ~ _ => e}
```

The ~! combinator ensures that if any of the following parsers *in the current sequence* fails, then a special (fatal) error is produced, which immediately breaks the parsing process, effectively *disabling backtracking* for the current sequence.

Unifying the non-backtracking (~!) and the keep-left / keep-right (~>, <~ / ~<) functionality Note that we had to explicitly rewrite the parsing result in the last example, because the ~! combinator returns both sides of the sequence as its parsing result. The standard Scala library does not provide combinators that unify the non-backtracking (~!) and the keep-left / keep-right (~>, <~ / ~<) functionality. But such combinators can be easily introduced:

```
def ~>! [U](q: => Parser[U]): Parser[U] = this ~! q ^^ {case a ~ b => b} named ("~>!")

def <~! [U](q: => Parser[U]): Parser[T] = this ~! q ^^ {case a ~ b => a} named ("<~!")
def ~<! [U](q: => Parser[U]): Parser[T] = this <~! q named ("~<!")
```

The previous example would then become `"(" ~>! expr <~! ")"` or `"(" ~>! expr ~<! ")"`.

Note also that we used non-backtracking combinators instead of *all* sequence combinators, notwithstanding that the first one had to work for the whole current sequence. This is due to specifics of the current implementation, which does not always automatically work up to the end of the current sequence.

[14] https://github.com/scala/scala/blob/v2.9.1/src/library/scala/util/parsing/combinator/Parsers.scala#L266-278

Chapter 5

Example Projects

This chapter presents a few projects developed using combinator parsers, with explanation of techniques used for implementation.

5.1 Propositional Logic

5.1.1 Objectives

The program should be able to understand expressions of propositional logic consisting of constants (`true`, `false`), variables, operators (`xor`, `equivalence`, `implication`, `or`, `and`, `not`), and parentheses. If the expression is syntactically valid, the program should print a truth table for that expression.

5.1.2 Preliminary considerations

At first we need to specify the input language, i.e. the syntax of expressions.

5.1.2.1 Elementary elements

The elementary elements of expressions are *constants* and *variables*; obviously, they should be distinguishable. This can be achieved in two ways:

- Define a single syntactical class of *words* that will enclose both names of variables and names of constants; define *reserved* words that will be used for constants and may not be used for variables; check for each word found in the input whether it is reserved to distinguish between variables and constants.

  or

- Define two *syntactically* different classes for variables and for constants, e.g. require that variables starts with a lower-case letter whereas constants must start with an upper-case letter.

We decide to go the second way here because of simplicity. The variables should start with a lower-case letter and continue with letters, digits, or underscore; the constants for `true` and `false` will be the upper-case letter "`T`" and "`F`".

5.1.2.2 Operators

Further, we look at the operators.

Arity The operators of propositional logic can be classified by *arity* into:

1) binary: `xor`, `equivalence`, `implication`, `or`, `and`;

   and

2) unary prefix: `not`.

Precedences We will give our operators *precedences* to make possible writing expressions without parentheses:

- the precedence of unary operators should be higher (bind more tightly) than that of binary,
- the precedences of binary operators should be as follows (from low to high):

  `xor`, `equivalence`, `implication`, `or`, `and`.

Associativity By *associativity*, all operators are left-associative, except `implication`, which is right-associative.

Symbolic notation for operators We will use the following symbolic notation for operators:

xor	`x ^ y`
equivalence	`x <-> y`
implication	`x -> y`
or	`x | y`
and	`x & y`
not	`!x`

5.1.2.3 Miscellaneous

Parentheses We specify two kinds of parentheses to better visual representation of expressions: "`()`" and "`[]`", which may be used interchangeably.

Whitespace Whitespace in expressions between syntactic elements should be ignored.

5.1.3 Implementation

The first version of the program has been written using ANTLR[1] / Java, and later rewritten in Scala. This offers an opportunity to compare these two techniques. We will show how the parser component is implemented in both variants.

5.1.3.1 Parser

The parser recognizes input expressions and builds an internal representation in form of an AST. The parser can be considered as the core component of the program.

[1] http://www.antlr.org/

ANTLR The following listing shows the parser grammar written in ANTLR. The ANTLR parser generator uses this grammar to produce the source code in Java (divided into lexer and parser parts, 48 KB in total) that performs the actual parsing.

```
// Parser rules
input : expr;
expr  : xor;
xor   : equiv ('^'^ equiv)*;
equiv : impl  ('<->'^ impl)*;
impl  : or    ('->'^ impl)?;   // right-associative via tail recursion
or    : and ('|'^ and)*;
and   : not ('&'^ not)*;
not   : '!'^? atom;
atom  : ID
      | CONST
      | '(' expr ')' -> expr
      | '[' expr ']' -> expr;

// Lexer rules
ID         : 'a'..'z' ('a'..'z'|'A'..'Z'|'0'..'9'|'_')*;
CONST      : 'T'|'F';
NEWLINE    : '\r'? '\n'   {skip();};
WHITESPACE : ('\t'|' ')+  {skip();};
```

Listing 5.1: Propositional logic: ANTLR Grammar

Note the ANTLR's operator "^" (as in `'->'^`), which is used to create AST nodes.

Another insteresting point is the way to implement right-associative operators (see the `impl` rule): the head is formed from the production with the next higher precedence (`or`), followed by the (tail) recursion of the `impl` itself. This is a common method to implement right-associativity in a variety of grammars and parser generators.

Scala The grammar written in Scala looks similar. As opposed to ANTLR, the Scala grammar is also a self-contained executable Scala code. No further source code for parsing is required.

```
  // Parser
  def expr  = xor
  def xor   = rep1sep(equiv, "^")  ^^ {_.reduceLeft(Xor)}
  def equiv = rep1sep(impl, "<->") ^^ {_.reduceLeft(Equiv)}
  def impl  = rep1sep(or, "->")    ^^ {_.reduceRight(Impl)} // right-associative
  def or    = rep1sep(and, "|")    ^^ {_.reduceLeft(Or)}    // via right folding
  def and   = rep1sep(not, "&")    ^^ {_.reduceLeft(And)}
  def not   = opt("!") ~ atom      ^^ {case Some(_)~x => Not(x)
                                       case _~x => x}
  def atom  = ( const              ^^ Const
              | id                 ^^ Id
              | "(" ~> expr <~ ")"
              | "[" ~> expr <~ "]"
              )
  def const = "T" | "F"
  def id    = """[a-z]\w*""".r
```

Listing 5.2: Propositional logic: Scala Grammar

Since Scala version uses specific node types (whereas all AST nodes created by ANTLR are of the same type), the nodes are explicitly created for each recognized input construction. The AST definition is shown below.

Chains of operators of the same precedence are stored in lists (since the `rep1sep` combinator returns its result as a list), then these lists are *folded* from the left to the right or vice versa to model the desired associativity. E.g. for the `impl`, which should be right associative, the right folding (`reduceRight`) is applied. Note that instead of the tandem `rep1sep` + left / right folding, the `chainl1` / `chainr1` combinators could be used as well.

Both grammars ANTLR and Scala utilize a top-down decomposition of the input, beginning from the operators with the lowest precedence toward the operators with the highest precedence.

5.1.3.2 AST

The parsed expression is represented in Scala nodes belonging to a hierarchy of specific types. Typed trees have the advantage of static typechecking of instructions for node creation and evaluation. Trees, where all nodes are of the same type, are easier to traverse.

The AST definition for Scala is based on the techniques from the section 3.3.

```
// AST
sealed abstract class Expr

case class Xor(l: Expr, r: Expr) extends Expr
case class Equiv(l: Expr, r: Expr) extends Expr
case class Impl(l: Expr, r: Expr) extends Expr
case class Or(l: Expr, r: Expr) extends Expr
case class And(l: Expr, r: Expr) extends Expr
case class Not(v: Expr) extends Expr
case class Const(name: String) extends Expr {
  override def toString = name}
case class Id(name: String) extends Expr {
  varNames += name; override def toString = name}

var varNames = mutable.Set[String]()
```

Listing 5.3: Propositional logic: AST

Note that the constructor of `Id` node adds itself to the variable `varNames` holding a *set* of variables found during parsing. Because of using a set, the duplicates (which could normally occur, since a variable can be used multiple times within an expression) are automatically eliminated. A better from architectural point of view, but also more expensive solution would be to discover the names of variables in a separate evaluation pass (by traversing the AST), independently of the parsing.

5.1.3.3 Evaluation

To print a truth table the expression must be evaluated for all variants of variables' allocation. A single allocation is represented via the map (`varValues`) from a name to its value. Assuming that `varValues` is already filled in, the evaluation of the AST is straightforward: first the children of a node are evaluated, then they are combined with a logical operation corresponding to the node type.

```
// Interpreter
def eval(e: Expr): Boolean = e match {
```

```
    case Xor(l, r)   => eval(l) ^ eval(r)
    case Equiv(l, r) => eval(l) == eval(r)
    case Impl(l, r)  => !eval(l) || eval(r)
    case Or(l, r)    => eval(l) || eval(r)
    case And(l, r)   => eval(l) && eval(r)
    case Not(x)      => !eval(x)
    case Const(x)    => x == "T"
    case Id(x)       => varValues(x)
  }

  var varValues = mutable.Map[String, Boolean]()
```

Listing 5.4: Propositional logic: Interpreter

The main processing logic is concentrated in the `process` method. To give the variables the alphabetical order, we first convert the set `varNames` to a collection that preserves the order of elements (a *Set* doesn't), and then sort it: `varNames.toArray.sorted`.

```
  def process(input: String) {
    println("input:  " + input)

    varNames.clear
    varValues.clear
    val res = parseAll(expr, input)
    println("result: " + res)

    if (res.successful)
      printTruthTable(res.get, varNames.toArray.sorted)
    println()
  }
```

Listing 5.5: Propositional logic: Processing

5.1.3.4 Printing truth table

The method `printTruthTable` does exactly that: prints a truth table.

```
71  def printTruthTable(tree: Expr, varNames: Array[String]) {
72    val (varCount,        resLabel, colSpace, resColSpace) =
73        (varNames.length, "Result", " ",      "   ")
74
75    // Header
76    println("TRUTH TABLE:\n------------")
77    println(varNames.mkString(colSpace) + resColSpace + resLabel)
78
79    // Body
80    val rowCount = 1 << varCount
81    for (r <- 0 until rowCount) {
82      // Evaluate the expression-tree for each state of variables and print the result:
83      // State of variables as an array of Booleans
84      val state = Array.tabulate[Boolean](varCount)(i => (r & (1<<i)) > 0)
85      // Store the current state of variables in the varVals map
86      varValues ++= varNames.zip(state)
87      print(varNames map (v => centered(varValues(v), v.length)) mkString colSpace)
```

```
88      println(resColSpace + centered(eval(tree), resLabel.length))
89    }
90
91    // helper functions
92
93    /** Returns the Boolean value <code>v</code> printed as "0" or "1"
94        in the middle of a String of length <code>w</code>. */
95    def centered(v: Boolean, w: Int) = {
96      val spaceBefore = (w-1)/2; val spaceAfter = w-spaceBefore-1
97      val buf = new StringBuilder;
98      // '1' in the prev. line is the length of a boolean if printed as "0" or "1"
99      for (i <- 0 until spaceBefore) buf += ' '
100     // We don't use (" " * Int) since it clashes with * method in Parsers
101     buf += (if (v) '1' else '0')
102     for (i <- 0 until spaceAfter) buf += ' '
103     buf.toString
104   }
105 }
```

Listing 5.6: Propositional logic: Printing Truth Table

There are a few interesting fragments in this code worth to be noted:

- parallel assignment of several values to several variables: lines 72–73 (`colSpace` is the space between columns of variables, `resColSpace` is the space between the block of variables' values and the last column that shows evaluation results);
- to iterate over all possible states of variables the following approach is used:
  - it will be iterated over integer values from 0 until $2^{number-of-variables}$ (lines 80–81);
  - for each number, the *binary representation* of that number is used *as states of boolean variables* (saved in the variable `state`), from left to the right, where 0 is interpreted as `false` and 1 as `true` (line 84);
  - then the list of variables' names `varNames` will be paired with `state` from the previos step and added to the map `varValues`, effectively overwriting previos bindings stored there (line 86);
  - then the current states of variables are printed, centered under their labels (line 87), followed by the result column, where the actual evaluation of the AST (variable `tree`) is executed (line 88);
- the helper method `centered` returns a string of a given length with a boolean value printed in the middle (lines 95–103). The code that constructs the result string:

```
97      val buf = new StringBuilder;
98      // '1' in the prev. line is the length of a boolean if printed as "0" or "1"
99      for (i <- 0 until spaceBefore) buf += ' '
100     // We don't use (" " * Int) since it clashes with * method in Parsers
101     buf += (if (v) '1' else '0')
102     for (i <- 0 until spaceAfter) buf += ' '
103     buf.toString
```

could be also expressed shorter as:

```
(" " * spaceBefore) + (if (v) '1' else '0') + (" " * spaceAfter)
```

but, as noted in the comment, there is a problem with the method `*` in the `Parser` class. What is happening here exactly? The method intended to be used in the shorter version of the code is defined on the `StringLike`[2] class, to a subtype of which, a `StringOps`, a string can be implicitly converted[3]. But the `RegexParsers` class, which our program extends (see below), also has an implicit conversion from a string to a `Parser`, and the latter also has a method `*` (two of them[4]). Despite of different signatures of all three methods, the compiler cannot disambiguate between and issues an error message:

```
... error: type mismatch;
 found   : java.lang.String
 required: ?{val *: ?}
Note that implicit conversions are not applicable because they are ambiguous:
 both method literal in trait RegexParsers of type (s: String)P.Parser[String]
 and method augmentString in object Predef of type (x: String)scala.collection.immutable.StringOps
 are possible conversion functions from java.lang.String to ?{val *: ?}
```

Fortunately, the last Scala version, where this error happens, is 2.8.0. In all later versions (checked in 2.8.1, 2.8.2, 2.9.0, 2.9.1) the compiler can pick the right implicit conversion, so that the shorter code can be used without problems.

5.1.3.5 The whole program

The whole program is composed by the parts shown above, augmented by a preamble, and a main method:

```
import util.parsing.combinator.RegexParsers
import collection._
object PropositionalLogic extends RegexParsers {

// Parser
// AST
// Interpreter
// process
// printTruthTable

  def main(args: Array[String]) {
    if (args.length > 0)
      process(args(0))
  }

}
```

Listing 5.7: Propositional logic: Preamble and the main method

5.1.3.6 Testing

Let's proof the first De Morgan's law $\overline{a \wedge b} = \overline{a} \vee \overline{b}$. For this purpose we will print truth tables for expressions "`!(a & b)`" and "`!a | !b`", and verify that they are in fact equal:

```
input:  !(a & b)
result: [1.9] parsed: Not(And(a,b))
```

[2] https://github.com/scala/scala/blob/v2.9.1/src/library/scala/collection/immutable/StringLike.scala#L68-74
[3] https://github.com/scala/scala/blob/v2.9.1/src/library/scala/Predef.scala#L312
[4] https://github.com/scala/scala/blob/v2.9.1/src/library/scala/util/parsing/combinator/Parsers.scala#L396-407

```
TRUTH TABLE:
------------
a b   Result
0 0     1
1 0     1
0 1     1
1 1     0

input:  !a | !b
result: [1.8] parsed: Or(Not(a),Not(b))
TRUTH TABLE:
------------
a b   Result
0 0     1
1 0     1
0 1     1
1 1     0
```

The next test shows that the implication will be handled correctly as a right associative operator:

```
input:  a -> b -> c
result: [1.12] parsed: Impl(a,Impl(b,c))
TRUTH TABLE:
------------
a b c   Result
0 0 0     1
1 0 0     1
0 1 0     1
1 1 0     0
0 0 1     1
1 0 1     1
0 1 1     1
1 1 1     1
```

Yet another test with more operators mixed together. Notice correctly recognized precedences according to the defined grammar:

```
input:  a & b ^ c | !a
result: [1.15] parsed: Xor(And(a,b),Or(c,Not(a)))
TRUTH TABLE:
------------
a b c   Result
0 0 0     1
1 0 0     0
0 1 0     1
1 1 0     1
0 0 1     1
1 0 1     1
0 1 1     1
1 1 1     0
```

5.2 Lambda Calculus

5.2.1 Objectives

The program should be able to understand a notation for nameless (*lambda*) functions. The body of a lambda function may use *free variables*, i. e. variables that are defined before the lambda definition. A lambda *application* should always use the values of free variables that were valid at the time of the lambda *definition*.

A variable may hold values of different types at different times.

The program should support integer and string literals, usual arithmetic operations, concatenation of strings with other types, and a `println` command to output the results of computations.

5.2.2 Preliminary considerations

5.2.2.1 Notation for lambda expressions

A common notation for lambda functions is "$\lambda\langle param\rangle.\langle body\rangle$".

"λ" symbol The Greek letter "λ" denoting the start of a lambda definition has the following specific: it is syntactically a *letter*. This has a consequence that the usual rules for parsing identifiers[5] will recognize the input "λx" as "$Identifier(\lambda x)$", and not as "$Lambda \sim Identifier(x)$". We have three reasonable alternatives to handle this situation:

- either define special rules for identifiers (where "λ" is prohibited),

  or

- always require whitespace around "λ", and handle "λ" as a *reserved identifier* (*keyword*),

  or

- use another symbol (syntactically not a *letter*) instead of the "λ".

We decide to go the last alternative and use a backslash "\" instead of the "λ": it is visually similar to a lambda glyph, and is also much simpler to type[6]. A disadvantage is that a backslash must be escaped in string literals and console inputs ("\\"), but it can still be used "as is" in Scala raw (triplequoted) strings.

Params We will allow *multiple* parameters (zero or more) separated by comma.

"." symbol Instead of the "." we will use a "=>" (in the style of Scala function literals) for better visual separation of "$\langle params\rangle$" and "$\langle body\rangle$" parts, especially in presence of several parameters and numerical literals that may start with a dot.

The final syntax of the lambda expression (using combinator parsing syntax to better express such idioms as "repetition with separator") is: `"\\" ~ repsep(param, ",") ~ "=>" ~ body`.

Example of a lambda expression: `\x,y => x + y`.

[5] E.g. the parser `ident` in `StdTokenParsers` or `ident` in `JavaTokenParsers`.

[6] Typing unicode symbols, such as "λ", is desktop specific. For example, in GTK+ based environments, one can use `Ctrl+Shift+U+3BB`. See also http://en.wikipedia.org/wiki/Unicode_input.

5.2.2.2 Types and typechecking

The language being implemented will be a *dynamically typed* language, in the sense that the typechecking will be performed at *runtime* (the time of evaluating the IR language), not at *compile time* (the time of creating and transforming the IR until the actual evaluation).

"Dynamic typed" is not an *inherent* property of this languge though. It could also be implemented as a *statically typed* language, in the sense that the typechecking would be performed before actual execution, and if it succeeded, then it's guaranteed that no type errors will occur at runtime.

It is just the overhead of a preliminary typecking (which involves the full traversing of the AST and is almost as expensive as the actual execution) that suggests to perform the only typechecking at runtime (combined with the evaluation).

If the underlying system (*target machine*) would have a more relaxed type system as compared to the input language (for example, if it would be allowed to perform, to say, an integer addition on the contents of two arbitrary memory cells or registers, as in case of the Java bytecode), this would justify the expensive static (i.e. compile-time) typechecking. Because in this case the original rich type system of the input language would be projected onto a more simple target type system, necessarily loosing information needed for the type checking (in terms of the input language) at runtime. The advantage would be a performance gain coming from absence of runtime typechecking.

Variables Objectives say "a variable may hold values of different types at different times". So we cannot statically bind a type to a variable. Variables will be untyped. The values of variables are always typed by means of the underlying system (Scala language).

Lambdas Like variables, types of lambda parameters will be not specified as well. I.e. we are going to implement an *untyped lambda calculus*. This design has some interesting implications:

1) Applying lambda to arguments of different types:

   Let's take a simple lambda expression `\x,y => x + y`. The types of parameters x and y are not specified, so the expression can be successfully applied to both numeric and string arguments. In the first case a numerical addition will be performed, in the latter a string concatenation:

   - `(\x,y => x + y) (5,3) // OK, the result is 8`
   - `(\x,y => x + y) ("Sca","la") // OK, the result is "Scala"`

   Also values of mixed types, where one of them is a string, could be allowed (the non-string value will be implicitly converted to a string, as usual in Scala and Java):

   - `(\x,y => x + y) (5, " times") // OK, the result is "5 times"`

   The following might be unexpected, but should also be legal:

   - `(\x,y => x + y) ("Hello", (\a => a*2)) // OK, the result should be a concatenation // of "Hello" and .toString called on the AST node representing the expression // (\a => a*2)`

   But applying a lambda expression to arguments that cannot be reasonably "added together" should result in a (runtime) error:

   - `(\x,y => x + y) (5, (\a => a*2)) // Error`

2) Possibility of non-termination of lambda application:

   In the untyped lambda calculus, the lambda application (which is a β-reduction) *need not terminate*. For instance, consider the term $(\lambda x.xx)(\lambda x.xx)$[7]. Here, we have:

[7] The example is taken from http://en.wikipedia.org/wiki/Lambda_calculus#Beta_reduction

- $(\lambda x.xx)(\lambda x.xx) \rightarrow (xx)[x := \lambda x.xx] = (x[x := \lambda x.xx])(x[x := \lambda x.xx]) = (\lambda x.xx)(\lambda x.xx)$.

That is, the term reduces to itself in a single beta reduction, and therefore reduction will never terminate. Expressed in our input language:

- `(\x => x x) (\x => x x)`

  or

- `f = \x => x x;  f f`

Since each recursive evaluation cicle opens a ne stack frame, the above constructions should lead to `StackOverflowError`. We will prove that during testing.

5.2.2.3 Lambda and Closure

Let's look at the requirement "a lambda *application* should always use the values of free variables that were valid at the time of the lambda *definition*". This means that each lambda definition *during evaluation* should be taken together with the current environment at the point of the lambda definion, and *saved* for the latter usage in applications. Such saved combination of a function and environment is called **closure**. An **environment** are bindings from names of variables to their values.

So, here are some important points to keep in mind upon implementing the evaluation procedure:

- Each lambda *application* will *use* one and the same closure of itself, which gets created at the point of the lambda definition.
- We need to keep an account of the current environment to be able to pass it into a closure creation when needed.
- Free variables are variables that are not defined inside lambda.

The concrete implementation will be shown in section Interpreter (page 70).

5.2.2.4 Println command

One way to implement a custom command, like `println`, is to introduce a *keyword* into the language. The AST should then define a dedicated node type for that command, and the parser would then have to create an instance of that node if it finds the command in the input. The command would be defined as a top level construct in the language grammar.

Another, and perhaps a more interesting way, is to handle the command in question as a *part of the standard library* for the language to be implemented. But what is a standard library from the technical point of view? A **standard library** is nothing else as an *initial environment* for a program, that is a predefined list of bindings from names to values. Admittedly we will be not able to implement the `println` in terms of more elementary parts of *our* language. So the implementation will have to be done in terms of the *underlying* language (which is Scala), much like native methods in `Java` or system calls in `C`.

5.2.3 Implementation

The implementation of this program is based on the excellent article series [Zei08].

5.2.3.1 Parser

The source code is shown in listing 5.8. We are not going to give a separate PEG grammar here because of better expressivity of the grammar written directly in the combinator parsers syntax.

The parser is based on the `StandardTokenParsers` to take advantage of some useful predefined parsers (see `ident`, `numericLit`, `stringLit` on lines 42–44), and automatic distinction between identifiers and keywords / delimiters (see `lexical.reserved` and `lexical.delimiters` on lines 18–19).

```
1  package lambda
2
3  import util.parsing.combinator.syntactical.StandardTokenParsers
4  import util.parsing.combinator.ImplicitConversions
5
6  object Parser extends StandardTokenParsers with ImplicitConversions {
7
8    import AST._
9
10   // API
11   def parse(input: String): Either[String, Expr] =
12     phrase(program)(new lexical.Scanner(input)) match {
13       case Success(ast, _) => Right(ast)
14       case e: NoSuccess => Left("parser error: " + e.msg)
15     }
16
17   //  Impl.
18   lexical.reserved   ++= ("if then else" split ' ')
19   lexical.delimiters ++= ("\\ => + - * / ( ) , == = ;" split ' ')
20
21   type P[+T] = Parser[T] // alias for brevity
22
23   def program       = rep1sep(expr, ";") <~ opt(";")                        ^^ Sequence
24   def expr: P[Expr] = lambda | ifExpr | assign | operations
25
26   def lambda        = ("\\" ~> repsep(ident, ",")) ~ ("=>" ~> expr)         ^^ Lambda
27   def ifExpr        = ("if" ~> expr) ~ ("then" ~> expr) ~ ("else" ~> expr) ^^ IfExpr
28   def assign        = ident ~ ("=" ~> expr)                                 ^^ Assign
29   def operations    = infixOps
30
31   def infixOps      = equality
32   def equality      = sum * ("==" ^^^ Equal)
33   def sum           = product * ("+" ^^^ Add | "-" ^^^ Sub)
34   def product       = postfixOps * ("*" ^^^ Mul | "/" ^^^ Div)
35
36   def postfixOps    = application
37
38   def application   = simpleExpr ~ rep(argList) ^^ {case e~args => (e /: args)(Application)}
39
40   def argList       = "(" ~> repsep(expr, ",") <~ ")" | simpleExpr ^^ {List(_)}
41
42   def simpleExpr    = ( ident               ^^ Var
43                       | numericLit          ^^ {x => Lit(x.toInt)}
44                       | stringLit           ^^ Lit
45                       | "(" ~> expr <~ ")"
46                       | failure ("Expression expected")
47                       )
48 }
```

Listing 5.8: λ calculus: Parser

The parser is intended to be used via a single API method `parse` (lines 11–15), which takes an input

string and returns either the root node of the AST created from the input, or a string with an error message. The return type is expressed using the Scala type `Either`, a disjoint union, where the right part means (by convention) "some successful result" and the left part stands for "failure".

The line 21 defines a short alias for the `Parser` type. It can be useful to save some horizontal space, where a return type of parsers has to (or should) be given, as in the recursive parser `expr` (line 24).

The grammar that defines the input language is given in the lines 23–47.

A **program** in the input language is a sequence of expressions *terminated or separated* by semicolons, so that both should be valid "`expr1; expr2;`" and "`expr1 ; expr2`". This idiom is achieved via `rep1sep` combinator with an optional semicolon at the end. How about an empty input, is it a legal program? No, we require at least one expression (note the "1" in "`rep1sep`").

The top-level **expressions** are lambda, if-expression, assignment, and operations. The lambda syntax follows the design worked out above. The if-expression should be self-explanatory. Assignments use the most simple form "`name = value`" without any additional keywords (like "`let`") or special assignment operators (like "`:=`"). Since we use "`=`" for assignment, an "`==`" will be used for equality comparison. The fourth element "operations" requires more detailed attention.

The **operations** parser represents the main precedence chain of supported operations. The infix operations have the lowest precedence, followed by prefix, and then by postfix operations (cf. explanations in 4.3.3 on page 52).

- The infix operations (**infixOps**) are represented by an equality chain (**equality**), which is decomposed into additive operations (**sum**), which in turn are decomposed into multiplicative ones (**product**). The "`*`" is another notation for `chainl` combinator, that is all operations are grouped in a left-associative fashion.
  Note that it is not required that each level is represented in the input via two or more elements separated with an operator, it can can be a single elment as well. For instance, a "`5`" is not only a numerical literal according to the grammar, it *is* also a "product" (which is defined as a chain of *one* or more elements separated by given operators), it *is* also a "sum", it *is* also an "equality" and so on.
  Note also that we are not creating dummy nodes that only forward to the next node; a node is only created if an operation is encountered or an elementary element is reached. So, the input "`5`" will not result in something like "InfixOps(Equality(Sum(Product(PostfixOps(...Lit(5)...))))", but simply in "Lit(5)".
- There are no prefix operations in our language, so we are going straight to
- Postfix operations (**postfixOps**), which is represented via (function) **application**: a construction like `f(z)` or `(\x => x/2) 4`. Note that chained applications will be left folded, and that single arguments need not be enclosed in parentheses.

At the bottom level is a simple expression (**simpleExpr**): an identifier, a numerical literal, a string literal, or a parenthesed expression that recurses the recognition back to the top level **expr** parser. If nothing from above is found, an error message "Expression expected" is issued.

5.2.3.2 AST

The AST definition is shown in listing 5.9. There is nothing spectacular here. The definitions of nodes correlate with the definitions of parsers from the previos section.

Note that there is no mention of a „closure“ neither in the parser nor in the AST. The „closure“ is a runtime construction, which comes into play during evaluation of the AST, not during parsing. (We will see in the next secion how lambda definitions get converted into closures.)

An expression representing a function in the `App` node (field `func`) might not be really a function that can be applied to the given arguments. The parser only ensures that it is a `simpleExpr`, nothing more. These checks will be performed in a separate semantical step, the next section is dedicated to.

```
package lambda

object AST {
  sealed abstract class Expr

  case class Sequence(l: List[Expr]) extends Expr {
    override def toString = l.mkString("\n")
  }
  case class Lambda(params: List[String], body: Expr) extends Expr {
    override def toString = "Lambda(Params("+params.mkString(", ")+"), Body("+body+"))"
  }
  case class IfExpr(e1: Expr, e2: Expr, e3: Expr) extends Expr
  case class Assign(name: String, expr: Expr) extends Expr

  case class Equal(e1: Expr, e2: Expr) extends Expr
  case class Add(e1: Expr, e2: Expr) extends Expr
  case class Sub(e1: Expr, e2: Expr) extends Expr
  case class Mul(e1: Expr, e2: Expr) extends Expr
  case class Div(e1: Expr, e2: Expr) extends Expr

  case class Application(func: Expr, args: List[Expr]) extends Expr {
    override def toString = "App(Func("+func+"), Args("+args.mkString(", ")+"))"}

  case class Var(name: String) extends Expr

  case class Lit(v: Any) extends Expr {
    override def toString = if (v.isInstanceOf[String]) "\""+v+"\"" else v.toString}
}
```

Listing 5.9: λ calculus: AST

5.2.3.3 Interpreter

This is the crucial, and probably the most interesting part of the program. The code is organized into the parts:

- API
- Type aliases
- Runtime types
- Evaluation

The first half of the Interpreter containing the parts “API”, “type aliases”, and “runtime types” is shown in Listing 5.10.

```
package lambda

object Interpreter {
  import AST._

  // API
  def eval(ast: Expr): Unit = {
```

```
  try   { (new Context) eval ast }
  catch { case e => val msg = e.getMessage
          println("eval error: " + (if (msg != null && msg != "") msg else e)) }
}

// Impl.

// Type aliases
type VarName = String
type VarValue = Any
type Environment = collection.immutable.Map[VarName, VarValue]


// Runtime types
sealed abstract class Func extends (List[Any] => Any)

case object Println extends Func {
  override def toString = "println (built-in)"
  def apply(args: List[Any]) = println(args mkString (", "))
}

case class Closure(env: Environment, lam: Lambda) extends Func {

  override def toString = "Closure(Env(" + env.mkString(",") + "), Params(" +
          lam.params.mkString(",") + "), Body(" + lam.body + "))"

  def apply(args: List[Any]) = {
    if (args.length != lam.params.length)
      error("Wrong number of arguments for function: expected " +
              lam.params.length + ", found " + args.length)
    else (new Context(env ++ lam.params.zip(args))) eval lam.body
  }
}
```

Listing 5.10: λ calculus: Interpreter (part 1 of 2)

API We define a simple API method "**eval**" that triggers the evaluation of a given AST node.

In this implementation, the `eval` method does not return anything (this is expressed via a formal return type `Unit`). The output will be done directly to the standard output (via `println`), as a side effect of the evaluation.

The `catch` clause could also be defined more simple as

```
catch { case e => println("eval error: " + e.getMessage) }  // suboptimal
```

This would work well for our own exceptions with a message (see below in listing 5.11), but is suboptimal when an exception doesn't have a message. For instance, if an evaluation enters infinite loop causing a stack overflow (see an example on page 66), then the exception being caught is a `StackOverflowError`. A simple `getMessage` may return `null` in this case, resulting in printed message "eval error: null", which is undesirable. Hence additional checks introduced to print the name of the exception in such situations, for instance: "eval error: java.lang.StackOverflowError".

Type aliases If the code itself documents its purpose, then the comments become obsolete. We define some type aliases to follow that concept. Compare "`immutable.Map[VarName, VarValue]`" vs "`Map[String, Any]`". The missing information in the latter case should have been otherwise added using comments. Yet with the type aliases it's already documented directly in the code (and the type

conformance will be enforced by compiler): e.g. an `Environment` (see below) is an immutable map from names to values of variables.

Runtime types We try to keep the IR language definition (i.e. the AST, see 5.2.3.2) free of constructs and information that can be computed, so that the abstract syntax remains abstract as much as possible. Yet during evaluation we need additional, *runtime* types to keep track of the computational process. These types are `Environment`, `Func` (with its subclasses `Println` and `Closure`), and `Context`.

- `Environment`

  Instances of this type hold bindings of variables $Name \rightarrow Value$. Important is the decision to use an *immutable* collection (an immutable `Map`) for the environment. This makes possible to simply pass the current environment into a new closure without worrying about anything. The immutability of the map "freezes" the values of variables in the passed environment instance. This works because the values of variables are themselves represented via immutable objects: instances of case classes from the AST definition.

  In such way closures can receive a freezed state of the current environment, and look up into it for free variables when needed.

- `Func / Println / Closure`

  The sealed type `Func`, which extends `Function` from `List[Any]` to `Any`, has two concrete subclasses: `Println` (singleton), and `Closure`.

  - The `Println` object implements our "library" function `println` (see p. 67). In its `apply` method it simply prints out the given list of arguments. Printed arguments are separated with comma. The list can also be empty, so that `println()` is a perfectly legal construct in the input language.
  - The `Closure` class implements the closure (see p. 67). It takes as constructor parameter an environment and a lambda definition. In its `apply` method it first checks that the number of *actual arguments* passed into is equal to the number of *formal parameters* from the lambda definition. If all is correct, then we enter the most magical moment: evaluation of a closure. Take a look at the corresponding code:

    ```
    (new Context(env ++ lam.params.zip(args))) eval lam.body
    ```

    * `lam.params.zip(args)` creates a list of pairs $parameterName \rightarrow argumentValue$. Both the parameter names (from the lambda definition) and the argument values (from the application of that lambda) are taken in their order, how they have been found in the input and returned by the parser (see listing 5.8: `repsep(ident, ",")` in `lambda` parser, and `argList` parser).
    * All those pairs are added to the current environment (which is a map $name \rightarrow value$) *replacing already existing bindings of the same names.* (Recall that environment objects are immutable maps, so that the ++ operation creates a *new* environment object here.)
    * A new Context object (see below) is created from the newly created environment, which performs the evaluation of the *body* of the original lambda definition.

    This implementation achieves an important semantical goal: names of formal parameters should shadow / overwrite names of free variables, or in other words, the names inside the body will in doubt always be treated as parameter names. In such way we perform the β-reduction[8] of the lambda: the names of parameters inside the body will now refer to the values of passed arguments, in order of parameter declaration.

[8] http://en.wikipedia.org/wiki/Lambda_calculus#Beta_reduction

Of course, this will also work when the lambda has an empty parameter list (then nothing gets changed in the current environment) or the names of parameters do not clash with the names of free variables. If the body of the lambda does not make use of free variables, such a lambda is a *pure* function.

Creating closures for pure functions is pretty useless. We could introduce a special class for pure functions, but this would also require investigation of bodies of the lambda definitions for usages of free variables, to decide which type of the function to create: a closure or a pure function. For the simplicity, in this implementation we decided to treat all functions as closures (even when they not really are) and simply overwrite the bindings in the environment (even when this doesn't really overwrite anything).

Both Println and Closure assume that arguments of their apply methods have already been evaluated by the Interpreter.

- `Context`

  Instances of this class hold the current environment and have an `eval` method to perform the evaluation of a given expression *in context* of the environment.

  The environment is held in a `var`, that allows to reassign it to a new value[9], if it should be changed in the current scope, as it happens in assignments.

  Another solution, suggested in [Zei08], could be passing the environment as a parameter into *every* `eval` call. The `eval` method would also have to return both an evaluation result and (perhaps changed) environment. This would avoid the mutable member holding the environment, thus making the code more "functional". But additional parameters in a lot of `eval` calls, the need to handle the returned environments (drop, reuse, or combine with the current one), performing every `eval` as a recursion (for instance, the evaluation of a `Sequence` would have to be recursively left-folded) would also make the code less readable. We could improve readability by declaring the environment as an implicit parameter, but this increases the overall complexity.

  So we decided to use a more simple design, where the (immutable) environment is a (mutable) member of `Context` class, which in turn defines the evaluation method. The environment is always available and can be flexibly changed in place (by assigning a new value to the member variable) by any part of the evaluation mechanism.

The second half of the Interpreter with the evaluation routine is shown in Listing 5.11.

```
// Evaluation
val initEnv: Environment = Map("println" -> Println)

class Context(private var env: Environment = initEnv) {
  override def toString = "Environment: " + env.mkString(", ")
  def eval(e: Expr): Any = e match {
    case Sequence(exprs) => exprs foreach eval
    case lam: Lambda => Closure(env, lam)
    case IfExpr(e1, e2, e3) => eval(e1) match {
      case b:Boolean => eval(if(b) e2 else e3)
      case _ => error("Not a boolean value in condition of IF expression")
    }
    case Assign(id, expr)  => env += (id -> eval(expr))

    case Equal(e1, e2) => eval(e1) == eval(e2)
    case Add(e1, e2) => (eval(e1), eval(e2)) match {
      case (v1:String, v2) => v1 + v2.toString
      case (v1, v2:String) => v1.toString + v2
```

[9] Remember that the referenced object of the type `Environment` is always immutable.

```
        case (i1:Int, i2:Int) => i1 + i2
        case _ => error("'+' requires two Int values or at least one String")
      }
      case Sub(e1, e2) => (eval(e1), eval(e2)) match {
        case (i1:Int, i2:Int) => i1 - i2
        case _ => error("'-' requires two Int values")
      }
      case Mul(e1, e2) => (eval(e1), eval(e2)) match {
        case (i1:Int, i2:Int) => i1 * i2
        case _ => error("'*' requires two Int values")
      }
      case Div(e1, e2) => (eval(e1), eval(e2)) match {
        case (i1:Int, i2:Int) => i1 / i2
        case _ => error("'/' requires two Int values")
      }

      case Application(expr, args) => eval(expr) match {
        case f:Func => f(args map eval)
        case x => error(expr + " cannot be applied as a function to argument(s) " + args +
                ".\n(Only functions can be applied)")
      }
      case Var(id) => env getOrElse(id, error("Undefined var " + id))
      case Lit(v) => v
    }
  }

}
```

Listing 5.11: λ calculus: Interpreter (part 2 of 2)

Evaluation By default, the evaluation is started with an initial environment consisting of a single binding: `"println" -> Println`. This is how our "standard library" function `println` comes into play.

The evaluation is implemented as a series of `case` statements performing pattern matching on the given AST node.

- In case of a `Sequence`, the evaluation will be performed iteratively for each expression in the list. Note how it is expressed using the infix (dotless) notation: "`exprs foreach eval`". But, wait, how can it work, if the `foreach` method takes a function object as a parameter, whereas `eval` is a *method*? This is yet another (very convenient) syntactical sugar from Scala: implicit convertion of a method to a function (**eta-expansion**) where needed. Eta-expansion is implemented in Scala in the way that the compiler generates and inserts a function object that calls the original method (e.g. our `eval` method) from the `apply` method of the newly generated function.

- In case of a `Lambda` definition, a Closure of that lambda will be returned.

- For an `If` expression is important to make sure that only one part ("then" or "else") is evaluated. It checks also that the condition evaluates to a value of type `Boolean`.

- An `Assignment` is the only case that changes the environment. It is implemented in the way that the current environment gets replaced with a new one containing an additional binding from the assignment. The old binding of the same name (if any) will be overwritten. In such way a variable can receive a new value (without type restrictions, as required in Objectives).

- The `Equality` relies on the default `equal` implementations of case classes, which is what all AST nodes are. Remember, that "==" in Scala is a semantical equality (which corresponds to `equals` in Java), not a comparison of references.

- The `Addition` is an interesting operation, because it implements not only an integer addition, but also a string concatenation, including the cases where only one part of the expression is a string.
- The `Sub` / `Mul` / `Div` are simple integer operations. They also check for the proper types of evaluated operands.
- The `Application` handles the cases where one expression follows another one in the input. The first expression must evaluate to a function, that is to an instance of `Func` (see above). Thus it could be either a `Closure` or the `Println`. Note that it cannot happen that we test a Lambda definiton to be a function: since the expression that represents a potential function is evaluated *before* pattern matching, it's guaranteed that if it was a `Lambda`, it got replaced with a `Closure` (see the implementation of the `Lambda` case above). In the next step all arguments are mapped to their evaluated values (see "`args map eval`"), and the function is simply applied to these arguments. [10]. Note yet another example of the eta-expansion in "`args map eval`".
- The `Var` case performs a lookup into the environment for the value of a given name.
- The `Lit` simply returns the own literal value, which can be a string or an integer number.

5.2.3.4 Main

The Main object is the starting point of the application. The code is shown in listing 5.12. The Main object extends the `App` trait that allows to put the processing logic directly in the body of the object. The predefined variable `args` is automatically filled in by Scala with the command line arguments, just like the `args` variable in `def main(args: Array[String])`.

All command line arguments will be iteratively processed with a dedicated method `process`. Following a defensive programming style, we also catch unforeseen exceptions here.

```
package lambda

object Main extends App {

  for (a <- args) {
    try   { process(a) }
    catch { case e => println("unexpected error: " + e.getMessage) }
  }

  def process(input: String) {
    println("\nINPUT:\n" + input)
    (Parser parse input) match {
      case Right(ast) =>
        println("\nTREE:\n" + ast)
        println("\nEVAL:"); Interpreter eval ast
      case Left(errMsg) =>
        println(errMsg)
    }
    println("====================")
  }

}
```

Listing 5.12: λ calculus: Main

[10] Remember that the function call syntax `f(x)` is a syntactic sugar for `f.apply(x)`

Each input will be first echoed to the standard output to make verifiable that the program really evaluates what the user expected (this can go wrong when entering some special symbols in the command line, for instance, a "*" can be expanded by the shell into the list of files in the current directory).

If the parsing was successful (i.e. the input was syntactically correct), the abstract syntax tree of the input is printed, followed by the result of the evaluation. A double line separates processing of different arguments.

5.2.3.5 Testing

Empty input:

```
INPUT:

parser error: Expression expected
```

Polymorphic addition:

```
Polymorphic addition:

INPUT:
f = \x,y => x + y;
println(f(5, 3));        // OK
println(f("Sca", "la")); // OK
println(f("Hello ", \a => a*2)); // OK! (String + Closure.toString)
f(5, \a => a*2); // eval error

TREE:
Assign(f,Lambda(Params(x, y), Body(Add(Var(x),Var(y)))))
App(Func(Var(println)), Args(App(Func(Var(f)), Args(5, 3))))
App(Func(Var(println)), Args(App(Func(Var(f)), Args("Sca", "la"))))
App(Func(Var(println)), Args(App(Func(Var(f)), Args("Hello ", Lambda(Params(a),
    Body(Mul(Var(a),2)))))))
App(Func(Var(f)), Args(5, Lambda(Params(a), Body(Mul(Var(a),2)))))

EVAL:
8
Scala
Hello Closure(Env(println -> println (built-in),f -> Closure(Env(println -> println (built-in)),
    Params(x,y), Body(Add(Var(x),Var(y)))), Params(a), Body(Mul(Var(a),2)))
eval error: '+' requires two Int values or at least one String
```

Aliasing `println`:

```
INPUT:
println(println);
f = println; f(5);
println(f)

TREE:
App(Func(Var(println)), Args(Var(println)))
Assign(f,Var(println))
App(Func(Var(f)), Args(5))
App(Func(Var(println)), Args(Var(f)))

EVAL:
println (built-in)
5
println (built-in)
```

Shadowing `println`:

```
INPUT:
println("Hello");    // printls Hello
println = \x => x+x; // shadows the original println binding
println("Hello");    // does not print anything
println(5);          // does not print anything

TREE:
App(Func(Var(println)), Args("Hello"))
Assign(println,Lambda(Params(x), Body(Add(Var(x),Var(x)))))
App(Func(Var(println)), Args("Hello"))
App(Func(Var(println)), Args(5))

EVAL:
Hello
```

Non-termination:

```
INPUT:
f = \x => x x ; f f;
(\x => x x) (\x => x x);

TREE:
Assign(f,Lambda(Params(x), Body(App(Func(Var(x)), Args(Var(x))))))
App(Func(Var(f)), Args(Var(f)))
App(Func(Lambda(Params(x), Body(App(Func(Var(x)), Args(Var(x)))))), Args(Lambda(Params(x),
    Body(App(Func(Var(x)), Args(Var(x)))))))

EVAL:
eval error: java.lang.StackOverflowError
eval error: java.lang.StackOverflowError
```

Illegal operations:

```
INPUT:
3 * "xyz";  // own exception
3 / 0;      // Scala/Java exception

TREE:
Mul(3,"xyz")
Div(3,0)

EVAL:
eval error: '*' requires two Int values
eval error: / by zero
```

The following three examples were adapted from [Zei08].

Reassignment of a free variable does not change the closure:

```
INPUT:
s = "foo";
println("s = " + s);
ls = \x => s + x;
println("ls = " + ls);
s = "bar";
println("s = " + s);
println("ls = " + ls);
println( ls("!") );  // prints "foo!"
```

```
TREE:
Assign(s,"foo")
App(Func(Var(println)), Args(Add("s = ",Var(s))))
Assign(ls,Lambda(Params(x), Body(Add(Var(s),Var(x)))))
App(Func(Var(println)), Args(Add("ls = ",Var(ls))))
Assign(s,"bar")
App(Func(Var(println)), Args(Add("s = ",Var(s))))
App(Func(Var(println)), Args(Add("ls = ",Var(ls))))
App(Func(Var(println)), Args(App(Func(Var(ls)), Args("!"))))

EVAL:
s = foo
ls = Closure(Env(println -> println (built-in),s -> foo), Params(x), Body(Add(Var(s),Var(x))))
s = bar
ls = Closure(Env(println -> println (built-in),s -> foo), Params(x), Body(Add(Var(s),Var(x))))
foo!
```

Equivalence of N-ary and curried functions:

```
INPUT:
add1 = \a,b => a+b;      // N-ary function
println(add1(3,4));      // 7
add2 = \a => \b => a+b;  // curried function
println(add2(3)(4));     // 7
println(add2 3 4);       // 7 (using arguments without parentheses)

TREE:
Assign(add1,Lambda(Params(a, b), Body(Add(Var(a),Var(b)))))
App(Func(Var(println)), Args(App(Func(Var(add1)), Args(3, 4))))
Assign(add2,Lambda(Params(a), Body(Lambda(Params(b), Body(Add(Var(a),Var(b)))))))
App(Func(Var(println)), Args(App(Func(App(Func(Var(add2)), Args(3))), Args(4))))
App(Func(Var(println)), Args(App(Func(App(Func(Var(add2)), Args(3))), Args(4))))

EVAL:
7
7
7
```

Y combinator[11] (allows to define recursive functions):

```
INPUT:
Y = \f =>
   (\x => f (\y => x x y))
   (\x => f (\y => x x y));
// Define a faculty function using Y
fact = Y(\fact => \n => if n == 0 then 1 else n*fact(n-1));
println("6! = " + fact(6)); //  prints "6! = 720"

TREE:
Assign(Y,Lambda(Params(f), Body(App(Func(Lambda(Params(x), Body(App(Func(Var(f)),
     Args(Lambda(Params(y), Body(App(Func(App(Func(Var(x)), Args(Var(x)))), Args(Var(y))))))))))),
     Args(Lambda(Params(x), Body(App(Func(Var(f)), Args(Lambda(Params(y),
     Body(App(Func(App(Func(Var(x)), Args(Var(x)))), Args(Var(y))))))))))))
Assign(fact,App(Func(Var(Y)), Args(Lambda(Params(fact), Body(Lambda(Params(n),
     Body(IfExpr(Equal(Var(n),0),1,Mul(Var(n),App(Func(Var(fact)), Args(Sub(Var(n),1))))))))))))
```

[11] See "call-by-value Y combinator" (a version of the Y combinator that can be used in call-by-value (applicative-order) evaluation) in http://en.wikipedia.org/wiki/Fixed-point_combinator#Other_fixed-point_combinators,.

```
App(Func(Var(println)), Args(Add("6! = ",App(Func(Var(fact)), Args(6)))))

EVAL:
6! = 720
```

Conclusion

It turned out that the subject of this thesis is large and complex. However I got an impression that many of the traditional approaches from textbooks are unnecessarily overcomplicated. For instance, formal languages will be often defined in terms of ambiguos generative grammars (mostly CFG), just to laboriously develop recognizers for that generative grammars and to fight with ambiguities in every practical parser implementation. Usage of recognition-based grammars (such as PEG), where operational semantics for recognizing are defined unambiguously, saves a lot of work by eliminating that problems. And as a bonus, one has a much better expressivity and interchangeability of the grammars. Combinator parsing, instead of parser generators, are the next step in gaining more productivity. They allow to create new parsing abstractions and are seamlessly incorporated in the host language, in which the main program is written. And Scala, across many programming languages, offers probably the best set of features that facilitate readable and concise grammar definitions. Numerous code examples in this thesis demonstrate, that even parsers that implement relatively complex languages, fit easily into one printed page.

Many interesting areas are left to explore. For example, it would be interesting to perform a *performance comparison* between combinator parsers and parser generators, based on some real-world task, such as parsing Java or Scala code. Yet another field of further investigations would be *transformation libraries*, such as Stratego/XT[12] and Kiama[13]. It would be also interesting to research how advanced Scala features, such as *compiler plugins*, *macros*[14], and *parameterized extractors*[15], can help in parser writing.

[12] http://strategoxt.org/

[13] http://code.google.com/p/kiama/

[14] New feature, expected in Scala 2.10.

[15] Not yet implemented.

Bibliography

All links were valid as of April 27, 2012.

[Aas91] Annika Aasa. "Precedences in Specifications and Implementations of Programming Languages." In: *Theoretical Computer Science* 142 (1991), pp. 183–194.

[AHL08] Roland Axelsson, Keijo Heljanko, and Martin Lange. *Analyzing Context-Free Grammars Using an Incremental SAT Solver*. 2008. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.144.7645&rep=rep1&type=pdf.

[Bas07] H.J.S. Basten. "Ambiguity Detection Methods for Context-Free Grammars." MA thesis. 2007. URL: http://homepages.cwi.nl/~paulk/thesesMasterSoftwareEngineering/2007/BasBasten.pdf.

[BS09] John Boyland and Daniel Spiewak. "TOOL PAPER: ScalaBison Recursive Ascent-Descent Parser Generator." In: *Preliminary Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA)*. 2009, pp. 143–154. URL: http://ldta.info/2009/ldta2009proceedings.pdf.

[Cha99] Manuel M.T. Chakravarty. *Lazy Lexing is Fast*. 1999. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.26.5166&rep=rep1&type=pdf.

[Con11] Jay Conrod. *A simple interpreter from scratch in Python (4 parts)*. Feb. 6, 2011. URL: http://www.jayconrod.com/posts/37/a-simple-interpreter-from-scratch-in-python-part-1.

[CT12] Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. 2nd ed. Elsevier Inc, 2012. ISBN: 978-0-12-088478-0. URL: http://store.elsevier.com/product.jsp?isbn=9780120884780.

[DD10] C. J. Date and Hugh Darwen. *Database Explorations: Essays on The Third Manifesto and related topics*. Trafford Publishing, 2010.

[Die08] Howard Dierking. "Bjarne Stroustrup on the Evolution of Languages. Interview++." In: *MSDN Magazine* (Apr. 2008). URL: http://msdn.microsoft.com/en-us/magazine/cc500572.aspx.

[DS01] Atze Dijkstra and Doaitse S. Swierstra. *Lazy Functional Parser Combinators in Java*. 2001. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.29.1452&rep=rep1&type=pdf.

[DV02] Eelco Dolstra and Eelco Visser. *Building Interpreters with Rewriting Strategies*. 2002. URL: http://archive.cs.uu.nl/pub/RUU/CS/techreps/CS-2002/2002-022.pdf.

[Ell11] Conal Elliott. *Denotational design with type class morphisms. (extended version)*. 2011. URL: http://conal.net/papers/type-class-morphisms/type-class-morphisms-long.pdf.

[Fer04] João Fernandes. *Generalized LR Parsing in Haskell*. Tech. rep. 2004. URL: http://wiki.di.uminho.pt/twiki/pub/Personal/Joao/--TwikiJoaoPublications/technicalReport.pdf.

[FL03] Manuel Fähndrich and K. Rustan M. Leino. *Declaring and Checking Non-null Types in an Object-Oriented Language*. 2003. URL: http://research.microsoft.com/en-us/um/people/leino/papers/krml109.pdf.

[Flo94] G. Florijn. *Modelling Office Processes with Functional Parsers*. 1994. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.19.1307&rep=rep1&type=pdf.

[Fok95] Jeroen Fokker. “Functional Parsers.” In: *Advanced Functional Programming, Tutorial text of the First international spring school on advanced functional programming techniques* LNCS 925 (1995), pp. 1–23. URL: http://people.cs.uu.nl/jeroen/article/parsers/parsers.ps.

[For02] Bryan Ford. “Packrat Parsing: Simple, Powerful, Lazy, Linear Time.” In: *International Conference on Functional Programming, October 4-6, 2002, Pittsburgh* (Oct. 2002). URL: http://pdos.csail.mit.edu/~baford/packrat/icfp02/packrat-icfp02.pdf.

[For04] Bryan Ford. “Parsing Expression Grammars: A Recognition-Based Syntactic Foundation.” In: *Symposium on Principles of Programming Languages, January 14-16, 2004, Venice, Italy* (Jan. 2004). URL: http://pdos.csail.mit.edu/~baford/packrat/popl04/peg-popl04.pdf.

[Fre10] Stephan Freund. *Combinator Parsing in Scala*. June 21, 2010. URL: http://users.informatik.haw-hamburg.de/~sarstedt/AKOT/scala-parser-combinators.pdf.

[Gas05] Verónica Gaspes. *Domain Specific Languages. Lecture 4: Deep and Shallow embeddings*. Feb. 2005. URL: http://www2.hh.se/staff/vero/dsl/printL4.pdf.

[GH95] Andrew D. Gordon and Kevin Hammond. *Monadic I/O in Haskell 1.3 (Section "Combinator Parsing")*. June 5, 1995. URL: http://www-fp.dcs.st-and.ac.uk/~kh/papers/io-tutorial/section3_5.html.

[Gho08] Debasish Ghosh. *External DSLs made easy with Scala Parser Combinators*. Apr. 14, 2008. URL: http://debasishg.blogspot.com/2008/04/external-dsls-made-easy-with-scala.html.

[Gij09] Bas van Gijzel. *Comparing Parser Construction Techniques*. 2009. URL: http://fmt.cs.utwente.nl/files/sprojects/76.pdf.

[GJ08] Dick Grune and Ceriel J.H. Jacobs. *Parsing Techniques. A Practical Guide*. 2nd ed. Springer, 2008. ISBN: 978-0-387-20248-8. URL: http://dickgrune.com/Books/PTAPG_2nd_Edition/.

[Gle10] Mario Gleichmann. *Functional Scala: Closures*. Nov. 15, 2010. URL: http://gleichmann.wordpress.com/2010/11/15/functional-scala-closures/.

[Gri06] Robert Grimm. *Better Extensibility through Modular Syntax*. 2006. URL: http://cs.nyu.edu/rgrimm/papers/pldi06.pdf.

[Han04] Christian Plesner Hansen. “An efficient, dynamically extensible ELL parser library.” MA thesis. 2004. URL: http://data.quenta.org/tedir.pdf.

[HM96] Graham Hutton and Erik Meijer. *Monadic Parser Combinators*. Tech. rep. NOTTCS-TR-96-4. Department of Computer Science, University of Nottingham, 1996. URL: http://www.cs.nott.ac.uk/~gmh/monparsing.pdf.

[HM98a] Graham Hutton and Erik Meijer. *Monadic parsing in Haskell*. 1998. URL: http://www.cs.tufts.edu/~nr/cs257/archive/graham-hutton/monadic-parsing-jfp.pdf.

[HM98b] Graham Hutton and Erik Meijer. “Monadic Parsing in Haskell.” In: *Journal of Functional Programming* 8.4 (July 1998), pp. 437–444. URL: http://www.cs.nott.ac.uk/~gmh/pearl.pdf.

[Hof+08] Christian Hofer et al. *Polymorphic Embedding of DSLs*. 2008. URL: http://www.daimi.au.dk/~ko/papers/gpce50_hofer.pdf.

[Hut92a] Graham Hutton. *Higher-Order Functions for Parsing*. 1992. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.63.3555&rep=rep1&type=pdf.

[Hut92b] Graham Hutton. “Higher-order Functions for Parsing.” In: *Journal of Functional Programming* 2.3 (July 1992), pp. 323–343. URL: http://www.cs.nott.ac.uk/~gmh/parsing.pdf.

[Ier08] Roberto Ierusalimschy. *A Text Pattern-Matching Tool based on Parsing Expression Grammars*. 2008. URL: http://www.inf.puc-rio.br/~roberto/docs/peg.pdf.

[Jel10] Rick Jelliffe. *What is a Packrat Parser? What are Brzozowski Derivatives?* Mar. 10, 2010. URL: http://broadcast.oreilly.com/2010/03/what-is-a-packrat-parser-what.html.

[JM09] Trevor Jim and Yitzhak Mandelbaum. “Efficient Earley Parsing with Regular Right-hand Sides.” In: *Preliminary Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA)*. 2009, pp. 127–142. URL: http://ldta.info/2009/ldta2009proceedings.pdf.

[Jon09] Manohar Jonnalagedda. *Packrat Parsing in Scala*. Tech. rep. LAMP Laboratory for Programming Methods, EPFL, Jan. 17, 2009. URL: http://scala-programming-language.1934581.n4.nabble.com/attachment/1956909/0/packrat_parsers.pdf.

[KP98] Pieter Koopman and Rinus Plasmeijer. *Efficient Combinator Parsers*. 1998. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.36.2596&rep=rep1&type=pdf.

[Leta] Thomas Karl Letschert. *PKR: Programmiersprachen, Konzepte und Realisationen. Materialien zur Veranstaltung an der FH Gießen–Friedberg im WS 2010–2011*. Folien. URL: http://homepages.fh-giessen.de/~hg51/Veranstaltungen/PKR-1011/.

[Letb] Thomas Karl Letschert. *PKR: Programmiersprachen, Konzepte und Realisationen. Materialien zur Veranstaltung an der FH Gießen–Friedberg im WS 2010–2011*. Skript. URL: http://homepages.fh-giessen.de/~hg51/Veranstaltungen/PKR-1011/pl.pdf.

[LM01] Daan Leijen and Erik Meijer. *Parsec: Direct Style Monadic Parser Combinators for the Real World*. Tech. rep. UU-CS-2001-27. Department of Computer Science, University of Utrecht, 2001. URL: http://legacy.cs.uu.nl/daan/download/papers/parsec-paper.pdf.

[Lóp97] Pablo E. Martínez López. *Generic Parsing Combinators*. 1997. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.48.7088&rep=rep1&type=pdf.

[McB08] Jim McBeath. *Scala Parser Combinators*. Sept. 26, 2008. URL: http://jim-mcbeath.blogspot.com/2008/09/scala-parser-combinators.html.

[McB09] Jim McBeath. *Scala Functions vs Methods*. May 15, 2009. URL: http://jim-mcbeath.blogspot.com/2009/05/scala-functions-vs-methods.html.

[Miz10] Kota Mizushima. *PEGEX: A PEG-based pattern matching library EXtended by back reference with regex-like notation in Scala*. Apr. 2010. URL: http://www.coins.tsukuba.ac.jp/~i021216/pegex.pdf.

[MMY10] Kota Mizushima, Atusi Maeda, and Yoshinori Yamaguchi. “Packrat Parsers Can Handle Practical Grammars in Mostly Constant Space.” In: *PASTE’10, June 5-6, 2010, Toronto, Ontario, Canada* (June 2010). URL: http://www.diku.dk/hjemmesider/ansatte/henglein/papers/mizushima2010.pdf.

[Mona] *Monade (Typkonstruktion) (Wikipedia / de)*. Aug. 24, 2010. URL: http://de.wikipedia.org/wiki/Monade_(Typkonstruktion).

[Monb] *Monads in Scala*. Oct. 11, 2006. URL: http://lamp.epfl.ch/~emir/bqbase/2005/01/20/monad.html.

[Moo07] Adriaan Moors. *Parser Combinators in Scala*. Mar. 9, 2007. URL: https://codereview.scala-lang.org/fisheye/browse/~raw,r=10301/scala-svn/combinator/trunk/doc/tutorial/pdf/sparsec.pdf.

[MPO08] Adriaan Moors, Frank Piessens, and Martin Odersky. *Parser Combinators in Scala*. Tech. rep. CW491. Department of Computer Science, Katholieke Universiteit Leuven, Feb. 2008. URL: http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW491.pdf.

[Ode11] Martin Odersky. *The Scala Language Specification*. Version 2.9. May 24, 2011. URL: http://www.scala-lang.org/docu/files/ScalaReference.pdf.

[Oka98] Chris Okasaki. “Even higher-order functions for parsing or Why would anyone ever want to use a sixth-order function?” In: *Journal of Functional Programming* 8.2 (Mar. 1998), pp. 195–199. URL: http://www.eecs.usma.edu/webs/people/okasaki/jfp98.ps.

[OSV08] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. 1st ed. Artima Inc, Nov. 2008. ISBN: 978-0981531601. URL: http://www.artima.com/pins1ed/.

[OSV10] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. 2nd ed. Artima Inc, Dec. 2010. ISBN: 978-0981531649. URL: http://www.artima.com/shop/programming_in_scala_2ed.

[Par08] Doug Pardee. *Scala function objects from a Java perspective*. Mar. 1, 2008. URL: http://creativekarma.com/ee.php/weblog/comments/scala_function_objects_from_a_java_perspective/.

[Par09a] Terence Parr. *Grammars*. Feb. 2, 2009. URL: http://www.antlr.org/wiki/display/CS652/Grammars.

[Par09b] Terence Parr. *Overview of language implementation and supporting tools*. Jan. 21, 2009. URL: http://www.antlr.org/wiki/display/CS652/Overview+of+language+implementation+and+supporting+tools.

[Pat] Richard E. Pattis. *EBNF: A Notation to Describe Syntax*. URL: http://www.cs.cmu.edu/~pattis/misc/ebnf.pdf.

[Peg] *Parsing Expression Grammar (Wikipedia / en)*. URL: http://en.wikipedia.org/wiki/Parsing_expression_grammar.

[Pol09] David Pollak. "Parsers – Because BNF Is Not Just for Academics Anymore." In: *Beginning Scala*. Apress, May 2009. Chap. 8. ISBN: 9781430219897. URL: http://amazon.de/o/ASIN/1430219890/.

[Pop09] Bernie Pope. *Parser Combinators in Scala*. 2009. URL: http://www.berniepope.id.au/docs/scala_parser_combinators.pdf.

[Pra73] Vaughan R. Pratt. *Top Down Operator Precedence*. 1973. URL: http://hall.org.ua/halls/wizzard/pdf/Vaughan.Pratt.TDOP.pdf.

[PTW98] Bernard Pope, Simon Taylor, and Mark Wielaard. *Monadic Parsing: A Case Study*. 1998. URL: http://ww2.cs.mu.oz.au/~bjpop/papers/monad_parse.ps.gz.

[Rey72] John C. Reynolds. "Definitional Interpreters for Higher-Order Programming Languages." In: *Proceedings of the ACM National Conference*. Vol. 2. ACM, Aug. 1972, pp. 717–740. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.110.5892&rep=rep1&type=pdf.

[Ror06] Ruben J. Rorije. "Input/Output in Functional Languages. Using Algebraic Union Types." 2006. URL: http://essay.utwente.nl/57287/1/scriptie_Rorije.pdf.

[SA99] S. Doaitse Swierstra and Pablo R. Azero Alcocer. *Fast, Error Correcting Parser Combinators: A Short Tutorial*. 1999. URL: http://people.cs.uu.nl/doaitse/Papers/1999/SofSem99.pdf.

[Sch06] Sylvain Schmitz. *Modular syntax demands verification*. Tech. rep. 2006. URL: http://www.i3s.unice.fr/~mh/RR/2006/RR-06.32-S.SCHMITZ.pdf.

[SD00] S. Doaitse Swierstra and Luc Duponcheel. *Deterministic, Error-Correcting Combinator Parsers*. 2000. URL: http://www.staff.science.uu.nl/~swier101/Papers/1996/LL1.pdf.

[SJ09] Elizabeth Scott and Adrian Johnstone. "GLL Parsing." In: *Preliminary Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA)*. 2009, pp. 113–126. URL: http://ldta.info/2009/ldta2009proceedings.pdf.

[Spe11] Tim Speier. "Untersuchung von API-Entwurfsrichtlinien in Scala und des TransRelationalen Modells." MA thesis. 2011.

[Spi08] Daniel Spiewak. *Formal Language Processing in Scala*. June 16, 2008. URL: http://www.codecommit.com/blog/scala/formal-language-processing-in-scala.

[Spi09] Daniel Spiewak. *The Magic Behind Parser Combinators*. Mar. 24, 2009. URL: http://www.codecommit.com/blog/scala/the-magic-behind-parser-combinators.

[Spi10a] Daniel Spiewak. *Derivative Parsing*. Dec. 2, 2010. URL: http://www.cs.uwm.edu/~dspiewak/papers/derivative-parsing.pdf.

[Spi10b] Daniel Spiewak. *Generalized Parser Combinators*. Mar. 28, 2010. URL: http://www.cs.uwm.edu/~dspiewak/papers/generalized-parser-combinators.pdf.

[Spi11] Daniel Spiewak. *CanBeConfusing. API Design in Scala*. July 2011. URL: http://dl.dropbox.com/u/1679797/Scalathon/CanBeConfusing.pdf.

[Ste09] Ruedi Steinmann. *Handling Left Recursion in Packrat Parsers*. Mar. 26, 2009. URL: http://n.ethz.ch/~ruediste/packrat.pdf.

[Swi08] S. Doaitse Swierstra. *Combinator Parsing: A Short Tutorial*. Tech. rep. Dec. 2008. URL: http://www.cs.tufts.edu/~nr/cs257/archive/doaitse-swierstra/combinator-parsing-tutorial.pdf.

[Vis] *Visitor Pattern Versus Multimethods (Visitor pattern considered useless)*. URL: http://nice.sourceforge.net/visitor.html.

[Wad85] Philip Wadler. *How to Replace Failure by a List of Successes. A method for exception handling, backtracking, and pattern matching in lazy functional languages*. 1985. URL: http://www.springerlink.com/content/y7450255v2670167/fulltext.pdf.

[WDM08] Alessandro Warth, James R. Douglass, and Todd Millstein. *Packrat Parsers Can Support Left Recursion*. Tech. rep. TR-2007-002. 2008. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.99.7466&rep=rep1&type=pdf.

[Zei08] Stefan Zeiger. *Formal Language Processing in Scala (6 parts)*. July 2008–July 2009. URL: http://szeiger.de/blog/2008/07/27/formal-language-processing-in-scala-part-1/.

Index

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Gießen, 27. April 2012

Eugen Labun