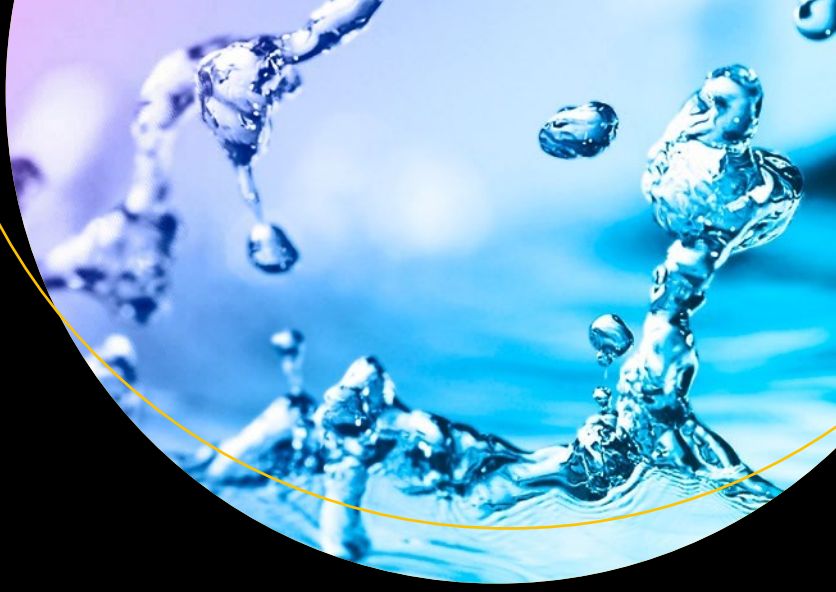

Parsing with Perl 6 Regexes and Grammars

A Recursive Descent into Parsing

Moritz Lenz

Apress®

Parsing with Perl 6 Regexes and Grammars

A Recursive Descent into Parsing

Moritz Lenz

Apress®

Parsing with Perl 6 Regexes and Grammars: A Recursive Descent into Parsing

Moritz Lenz
Fürth, Bayern, Germany

ISBN-13 (pbk): 978-1-4842-3227-9 ISBN-13 (electronic): 978-1-4842-3228-6
https://doi.org/10.1007/978-1-4842-3228-6

Library of Congress Control Number: 2017960890

Cover image by Freepik (www.freepik.com)

Managing Director: Welmoed Spahr
Editorial Director: Todd Green
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Technical Reviewer: Massimo Nardone
Coordinating Editor: Mark Powers
Copy Editor: Brendan Frost

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484232279. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Table of Contents

About the Author

Moritz Lenz is a contributor to the Rakudo Perl 6 compiler, initiator of the official Perl 6 documentation project, former maintainer of the official test suite, and a prolific blogger and author.

He works as software architect and principal software engineer for a mid-sized IT outsourcing company.

About the Technical Reviewer

Massimo Nardone has more than 22 years of experience in Security, Web/Mobile development, Cloud, and IT Architecture. His true IT passions are Security and Android.

He has been programming and teaching how to program with Android, Perl, PHP, Java, VB, Python, C/C++, and MySQL for more than 20 years.

He holds a Master of Science degree in Computing Science from the University of Salerno, Italy.

He has worked as a Project Manager, Software Engineer, Research Engineer, Chief Security Architect, Information Security Manager, PCI/SCADA Auditor, and Senior Lead IT Security/Cloud/SCADA Architect for many years.

Massimo's technical skills include Security, Android, Cloud, Java, MySQL, Drupal, Cobol, Perl, Web and Mobile development, MongoDB, D3, Joomla, Couchbase, C/C++, WebGL, Python, Pro Rails, Django CMS, Jekyll, and Scratch.

He currently works as Chief Information Security Officer (CISO) for Cargotec Oyj.

He worked as visiting lecturer and supervisor for exercises at the Networking Laboratory of the Helsinki University of Technology (Aalto University). He holds four international patents (PKI, SIP, SAML, and Proxy areas).

Massimo has reviewed more than 40 IT books for different publishing companies, and he is the coauthor of *Pro Android Games* (Apress, 2015).

Acknowledgments

I am grateful for fruitful and in-depth discussions with Barry Keeling, which helped me get a fresh look on some of the topics discussed.

I'd also like to thank the following beta readers for early feedback and corrections:

- Andrew Shitov
- Brian Duggan
- Cassandra T.
- Daniel Green
- Douglas E. Miles
- Fernando Santagata
- Jonathan Scott Duff
- Lanlan Pan
- Laurent Rosenfeld
- Leon Timmermans
- Mark Devine
- Mohammad S. Anwar
- Paul Cochrane
- Rúbio R. C. Terra
- Theodore Katseres
- Vitali Peil

- Wolfgang Banaston
- Zengargoyle
- Zoffix Znet

The number of names on this list should give you an idea of how awesome the Perl 6 community is. I asked for proofreaders for a manuscript on a fairly specialized topic, and more than 20 people volunteered, each putting multiple hours, sometimes even dozens of hours, into the task. This community also played a big part in teaching me the knowledge I relay here; I learned a lot about regexes and parsing from Patrick R. Michaud, Jonathan Worthington, Carl Mäsak, and Larry Wall, and I remain grateful for the freedom with which they taught everybody who wanted to learn.

Brian Duggan gets credit for coming up with the subtitle for this book.

Special thanks also go to the Apress team, who were both professional and very kind through the process of writing and publishing this book: Steve Anglin, Mark Powers, Massimo Nardone, and Matthew Moodie.

And last but not least, I'd like to thank my family for supporting me throughout the process of writing this book. Thank you Signe, Ronja, and Ida!

CHAPTER 1

What are Regexes and Grammars?

We come into contact with all sorts of structured data: telephone numbers, e-mail addresses, postal addresses, credit card numbers, and so on.

A regex is a declarative programming construct that describes such data formats. Regexes allow you to search for data, ensure that input is indeed in the described format, and even extract relevant components, such as the ZIP code of a postal address, or the timestamp from a log file entry.

When you need to read and validate more complex structures, such as a programming language, or a markup language like XML, you can combine many regexes into *grammars*. Grammars do more than simply combine regexes. They also offer infrastructure for generating good error messages and keeping track of state while analyzing the input text.

1.1 Use Cases

Searching

A common use of regexes is to search for patterns of interest in large volumes of data, such as looking for certain messages in log files, for URLs, or for phone numbers in text. At the time of writing, about 6% of the entries in my `.bash_history` involve searching with regexes.

M. Lenz, *Parsing with Perl 6 Regexes and Grammars*,
https://doi.org/10.1007/978-1-4842-3228-6_1

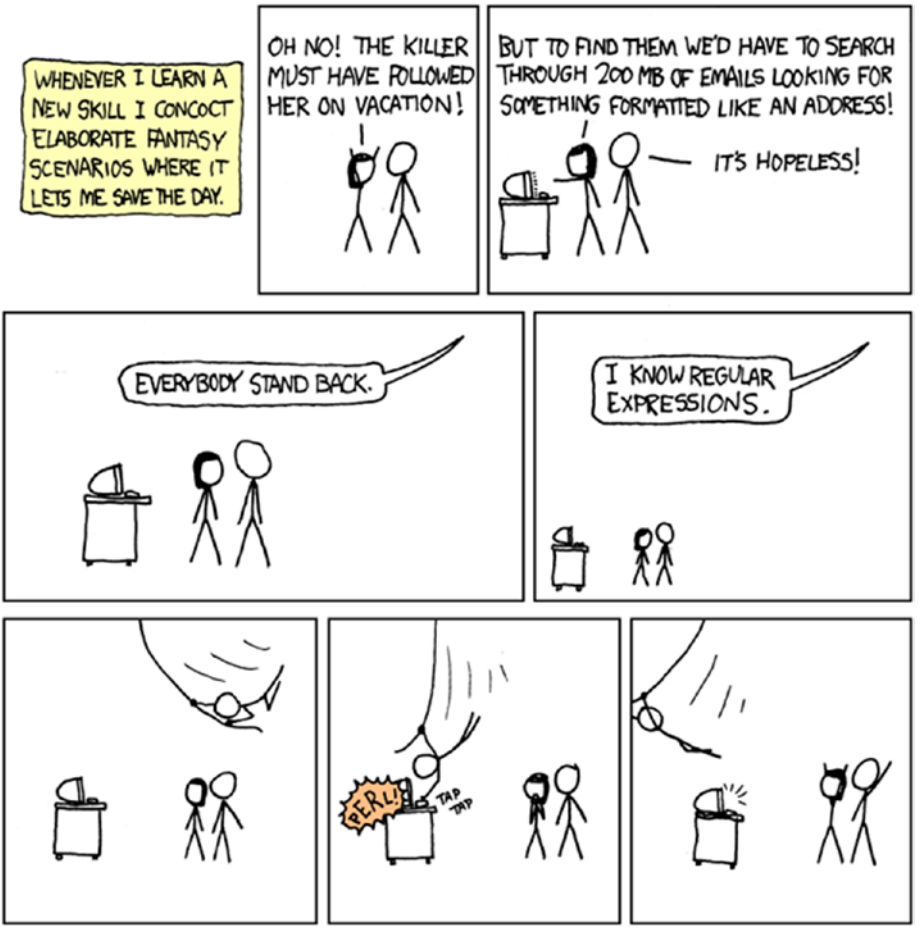

Figure 1-1. *"Regular Expressions" by Randall Munroe on XKCD, https://xkcd.com/208/*

Many command-line tools offer support for some dialect of regular expressions and allow you to search file names, file contents, logs, captured network traffic, and nearly everything else you can think of. Regexes are also easily accessible from most modern programming languages, making them a ubiquitous and indispensable search tool.

Validation

Most applications face untrusted user input. Web applications in particular are confronted with a lot of untrusted input. This input must be validated before applying any further logic to it or storing it in, for example, a database.

Regexes are a common first step toward validation. They make it trivial to check for simple things such as digits, and verifying the minimal and maximal length of input. At the same time, they allow the programmer to do much more precise and sophisticated checks.

All the web application programmer needs to provide is a regex and associate it with an input field. The web framework can then validate form input against all configured regexes and automatically generate error messages for the end user, so that the web application programmer does not need to deal with the workflow of rejecting the input and re-generating the form.

Parsing

Regexes alone are not very suitable for parsing complex input data. Perl 6, however, adds some features that make it well suited for this task. These extensions include easy-to-use backtracking control and composability through named regexes.

The result of a successful regex match is a *match object*, which contains all the necessary metadata to extract the interesting bits from the parsed text. There are also some features that make it easy to turn a match object into an *abstract syntax tree* or *AST*, a data structure suitable for use outside the parser.

1.2 Regexes or Regular Expressions?

The theoretical foundation for regular expressions comes from computer science, which describes a hierarchy of formal languages and automatons, or formal machines, that can recognize these languages. The most restricted of these languages is called a regular language.[1] Deciding whether or not a particular string is in a regular language requires a fixed amount of memory and a constant number of computing steps per character.

Regular expressions are a formalism for writing regular languages. As such concepts from theoretical computer science go, they are minimalistic, only allowing literals, alternations (`|`), parentheses for grouping, and the Kleene star[2] (*) for zero or more repetitions.

Early text processing tools such as `grep`, `sed`, and `awk` picked up the concept of regular expressions and added many convenience features, such as the ability to write `[a-z]` instead of `a|b|c|d|e...`. They provided predefined *character classes*, sets of characters like letters, digits, whitespace characters, and so on. They also added *captures*, which help with extracting strings that a particular part of a regular expression matches.

Later implementations added features that went beyond what regular languages allow, thus the need for a separate word. These implementations also optimize for ease of use instead of the minimalism of the theoretical construct that makes it easy to reason about. We now tend to use *regex* when talking about practical (and more powerful) implementations in programming languages and libraries.

[1] https://en.wikipedia.org/wiki/Regular_language
[2] https://en.wikipedia.org/wiki/Kleene_star

1.3 What's So Special about Perl 6 Regexes?

To continue the history course from the previous section, Perl was one of the first general-purpose programming languages to bake regexes into its core functionality. It borrowed syntax from earlier regex implementations and extended it in ways that made regexes more powerful and more useful. Soon, Perl's particular version of regex was the de facto standard. So much so, that a library called *Perl-Compatible Regular Expressions* (*PCRE*) was created so that other software could utilize "Perl regexes" in their implementations.

Unfortunately, in making regexes so useful, Perl had assigned special meaning to almost every ASCII character (except those that match literally). And, as newer and more powerful regex features were created, this led to using obscure character sequences for the new features while continuing to maintain backward compatibility with existing regex syntax. A good example of such a character sequence is `(?<=pattern)` for look-behind assertions.

Perl 6 regexes clean up this historical syntactic baggage. They improve readability by allowing whitespace everywhere, introducing clean rules about which characters are special and which aren't, and maybe most importantly, having a simple and extensible syntax for calling other regexes by name.

While most languages treat regexes either as strings or as special objects, Perl 6 regexes are code; and when grouped together within a grammar, they are like methods. This gives you the freedom to apply to regexes all the techniques for managing and reusing code that you are used to from programming languages: namespaces, classes, roles,[3] inheritance, etcetera.

[3]Other programming languages use the word "Traits" for the concept behind Perl 6 roles.

The ability to compose regexes makes it possible to do more than parse simple string formats. You can write grammars that use many small regexes to parse complex file formats. In fact, the Rakudo Perl 6 compiler itself uses a Perl 6 grammar to parse Perl 6 source code.

CHAPTER 2

Getting Started with Perl 6

You will likely pick up some things and understand the basic concepts from reading this book; but if your goal is fluency and a deeper understanding, you should run the examples yourself, modify them, and experiment with them.

In order to do that, you first need to install the Rakudo Perl 6 compiler, version 2017.05 or newer. Afterward, we'll discuss how to use it for regex experimentation.

If you are loath to install software on your computer, you could also use an online service that evaluates code for you. At the time of writing, `https://glot.io/new/perl6` and `https://tio.run/#perl6` support running Perl 6 code in the browser. You can also check `https://perl6.org/resources/` for an up-to-date list of similar services.

2.1 Installing Rakudo Perl 6

The Rakudo Perl 6 compiler comes in two varieties: the compiler itself, and Rakudo Star. The latter is a distribution containing the compiler, the `zef` module installer, documentation, and some modules.

M. Lenz, *Parsing with Perl 6 Regexes and Grammars*,
https://doi.org/10.1007/978-1-4842-3228-6_2

For our purposes, you need the compiler and `zef`. Installing Rakudo Star gives you both, but if the Rakudo Star installer doesn't work for you, or you prefer a leaner installation, you can install just the compiler and bootstrap `zef` according to its documentation.[1]

The following are some options for installing Rakudo Perl 6.

Rakudo Star from Native Installers

The Rakudo Star download page at `http://rakudo.org/downloads/star/` offers binary installers for Windows and Mac OS. You can install them by simply opening the downloaded file.

Binary Linux Packages

The Rakudo OS Packages[2] repository contains instructions on how to obtain and use Rakudo Perl 6 packages for CentOS, Debian, Fedora, and Ubuntu. They come with the compiler and a script to install `zef`; quite enough for our purposes.

Docker-Based Installation

On platforms with Docker support, you can obtain a prebuilt, lightweight image containing the Rakudo Perl 6 compiler, as well as `zef`, with just one command:

```
$ docker pull moritzlenz/perl6-regex-alpine
```

This Docker image contains Rakudo Perl 6 as well as a few modules that make it easier to work with regexes and grammars.

[1] `https://github.com/ugexe/zef#manual`
[2] `https://github.com/nxadm/rakudo-pkg/releases`

Once you have pulled the image, you can use it as follows to execute a one-liner:

```
$ docker run -it moritzlenz/perl6-regex-alpine -e 'say "hi"'
```

Since Docker containers run in their own isolated world, you need to take extra steps to make script files available to the container. For instance, if you wish to execute a script `search.p6`, you could run it like this:

```
$ docker run -it -v $PWD:/perl6 -w /perl6 \
     moritzlenz/perl6-regex-alpine search.p6
```

This is unwieldy, so a bash alias (or shell script) can help:

```
$ alias p6d="docker run -it -v $PWD:/perl6 -w /perl6
     moritzlenz/perl6-regex-alpine"
```

After that, executing a script becomes much easier:

```
$ p6d search.p6
```

In general, this book assumes the presence of a `perl6` executable. If you use the docker image, replace `perl6 with p6d` in all commands.

2.2 Using Rakudo Perl 6

You can verify that your Rakudo Perl 6 installation works by running `perl6 --version`, which should print something like this:

```
This is Rakudo version 2017.05-315-g160de7e built on MoarVM
version 2017.05-25-g62bc54e
implementing Perl 6.c.
```

If you can't get it to work yourself, you can ask the Perl 6 Community[3] for help.

[3] https://perl6.org/community/