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/

Once it works, you can start a simple, interactive shell by running `perl6` without further arguments:

```
$ perl6
To exit type 'exit' or '^D'
> say "Hello, world";
Hello, world
> exit
```

In this shell, you can type lines of Perl 6 code, which are executed immediately. This is very useful for testing regexes:

```
> say "Hello, world" ~~ / \w+ /
「Hello」
```

(Don't worry if you don't understand yet what exactly is happening here; the next chapter will explain it in detail.)

If the prompt comes back like this instead:

```
$ perl6
You may want to `zef install Readline` or `zef install
Linenoise` or use rlwrap for a line editor.

To exit type 'exit' or '^D'
>
```

You should follow the advice from the greeting message and install one of the modules mentioned. This gives you the ability to access and edit the history of your commands:

```
$ zef install Linenoise
===> Searching for: Linenoise
===> Searching for missing dependencies: LibraryMake
...
===> Installing: Linenoise:ver('0.1.1'):auth('Rob Hoelz')
```

Of course, you can also use `perl6` to execute a script file, just by adding the script file to the command line:

```
$ perl6 greet.p6
Hello, world
```

assuming you have a file `greet.p6` in the current working directory containing the line `say "Hello, world";`.

2.3 Obtaining the Code Examples

The source for the code examples used in this book is available on GitHub:

```
$ git clone https://github.com/apress/perl-6-regexes-and-grammars.git
```

If you don't have `git` available, you can also download a zip archive with the source code from https://github.com/apress/perl-6-regexes-and-grammars/archive/master.zip.

I encourage you to run these example scripts, modify them, and play with them. This is the best way to learn the topics more deeply than by merely reading this book.

2.4 First Steps with Perl 6

In order to use Perl 6 regexes effectively, you need to know a bit about the language they are embedded in.

Perl 6 is a mostly free-form language, so you indent your code whatever way you like. There are however cases where the presence or absence of whitespace matters: `doit(1, 2, 3)` calls the function `doit` with three arguments, whereas `doit (1, 2, 3)` calls the same function with a single argument, which is a list of three values.

Statements are separated by the semicolon (;) character, and you can also put one in the last line of your program. Comments start with a hashmark (#) character, and extend until the end of the line:

```
say "Hello, World";        # Output: Hello, World
say( 2 * 21 );             # Output: 42
```

Variables and Values

Variables are declared with the keyword `my`, and start with a *sigil*, a symbol that tells us a bit about their general type:

```
my $x = 1;
my @capitals = 'Algiers', 'Tirana', 'Berlin', 'Tokio';
my %populations =
      Algiers => 3_500_000,
      Berlin  => 3_700_000,
      Tirana  => 353_400,
      ;
```

The `$` stands for a scalar variable—a variable that typically holds a single value. In contrast, the `@` denotes an *array*, a linear collection of values that lets you access elements through a zero-based index. In the context of the previous code block, `@capitals[0]` returns `'Algiers'`.

A *hash* (some other languages call this a *dictionary* or a *map*) associates keys (which are strings) with values. You can look up values to a key via the {} indexing operator, and use several useful methods:

```
say %populations{'Algiers'};   # Output: 3500000
say %populations.keys.sort;    # Output: (Algiers Berlin Tirana)
say %populations.values.sum;   # Output: 7553400
```

Strings

Since regexes work on strings, we should talk more about them.

A string is a sequence of Unicode characters.[4] We've already seen some examples of strings surrounded by single quotes '' and double quotes "". The main difference is that double quotes enable escape sequences like \n for a line break (newline), and *interpolation* of variables, where the name of a variable is replaced by its value:

```
my $name = 'Larry';
say "Hello, $name";          # Output: Hello, Larry
say 'Hello, $name';          # Output: Hello, $name
```

However, even inside single quotes, you can write a single quote by escaping it with a backslash. By the same token, a backslash must be doubled in such a string:

```
say 'a quote: \' a backslash: \\';
    # Output: a quote: ' a backslash: \
```

Here-documents offer a neat way to write multiline strings:

```
my $macbeth = q:to/END/;    # need to put the ; on this line!
When shall we three meet again?
In thunder, lightning, or in rain?
    When the hurlyburly's done,
    When the battle's lost and won.
END
# normal Perl 6 code resumes here
```

You can pick your own delimiter, though it is customary to write it all in uppercase, to make it easier to spot where the here-document ends.

[4]Rakudo Perl 6 expects your program file to be UTF-8 encoded.

Control Structures

Control structures such as loops and branches all follow the same basic pattern, `KEYWORD EXPRESSION { BLOCK }`, and the `if` construct also allows `elsif` and `else` statements to be attached:

```
for 1, 2, 3 {
    say $_;
}
if 1 > 2 {
    say "No Way";
}
elsif 1 == 2 {
    say "Still no chance";
}
else {
    say "This runs";
}
```

This code produces the output

```
1
2
3
This runs
```

Inside the block of a `for` loop, the current value is stored in the special variable `$_`. If you want to use a different variable, you can use the following syntax, which is called a *pointy block*:

```
for 1, 2, 3 -> $value {
    say $value;
}
```

This works in any location where a block can be used, and can be extended to multiple *parameters* as well:

```
my $callback = -> $x, $y { $x + $y };
say $callback(1, 2);                    # Output: 3
```

Functions, Classes, and Methods

A *function* or *subroutine* is a piece of code with a list of formal parameters, and optionally a return value:

```
sub double($x) {
    return 2 * $x;
}
say double(2);                      # Output: 4
```

If no `return` statement runs, the return value is the value of the last expression, so we could have written this as `sub double($x) { 2 * $x }` instead.

You can optionally declare a type for parameters (and for variables):

```
sub double(Numeric $x) { 2 * $x }
```

A *class* can have per-object storage, called an *attribute*, and code that is attached to the object called a *method*:

```
class Point {
    has $.x;
    has $.y;
    method magnitude() {
        return sqrt($.x * $.x + $.y * $.y);
    }
}
my $p = Point.new( x => 5, y => 2 );
say $p.x;                               # Output: 5
say $p.magnitude();                     # Output: 5.3851648071345
```

Learning More About Perl 6

There are many resources for diving deeper into Perl 6.

Perl 6 Fundamentals: A Primer with Examples, Projects, and Case Studies[5] by Moritz Lenz (Apress Media, 2017) offers an example-driven approach to exploring Perl 6.

Perl 6 Deep Dive[6] by Andrew Shitov (Packt Publishing, 2017) is a more feature-oriented guide to similar topics.

`http://perl6intro.com/` is a free online resource for learning Perl 6, available in several languages.

Last but not least, the official Perl 6 documentation at `https://docs.perl6.org/` provides a mixture of introductory and reference material, and lets you search for built-in types, functions, methods, and operators.

2.5 Summary

We've seen several methods for installing the Rakudo Perl 6 compiler. Once you've installed it, you can start an interactive Perl 6 shell by running `perl6` without any arguments, or `perl6 script.p6` to execute a Perl 6 program.

We have also explored our first simple Perl 6 programs. Next we'll dive right into writing our first regexes.

[5] `www.apress.com/us/book/9781484228982`

[6] `www.packtpub.com/application-development/perl-6-deep-dive`

CHAPTER 3

The Building Blocks of Regexes

After all the talk and preparation, it's time to look at some actual regexes.

3.1 Literals

The simplest elements of a regular expression are literals: strings that exactly match themselves. For instance, the regex `/perl/` matches any string that contains the characters `p`, `e`, `r`, and `l` in exactly this order. So the string `"properly"` matches this regex from the fourth character onward (pro**perl**y), but the string `"superficial"` does not, because it contains other characters (`ficia`) between `per` and `l`.

Literals composed of only letters, the underscore `_`, and digits don't require any special syntax. Simply writing the letters and digits as part of the regex makes up a literal. Other literals must be enclosed in quotes, either single or double: `/'2016-12-24'/` and `/"2016-12-24"/` are similar, and both match the string `2016-12-24`. Since the two dashes are neither letters nor digits, they need to be part of a quoted string to match literally. Single and double quote pairs differ in their behavior regarding *interpolation*: variables

M. Lenz, *Parsing with Perl 6 Regexes and Grammars*,
https://doi.org/10.1007/978-1-4842-3228-6_3

and code blocks delimited in curly braces `{...}` are interpreted in double quotes, so the following regexes all match the string `"3"`:

```
$_ = 3; # the string to be matched against
my $x = 3;
say "yes" if /"$x"/;          # Output: yes
say "yes" if /"{1 + 2}"/;     # Output: yes
```

whereas the regex `/'{1 + 2}'/` only matches the literal string `{1 + 2}`.

Whitespace is ignored by default in Perl 6 regexes, so `/perl/`, `/pe rl/`, and `/p e r l/` are all equivalent (although the latter two produce a warning), and match exactly the same strings. Inside quotes, whitespace is significant, so `/"pe rl"/` *does not* match the string `properly` because the string contains no space character, but the string inside the quoted literal does.

3.2 Meta Characters vs. Literals

As we've seen in the previous section, alphanumeric characters in regexes match themselves. All other characters can have some special meaning. We've already seen that ' and " are special: they surround quoted strings.

We call such "special" characters *metasyntactic*, or *meta* characters. The backslash \ makes a literal character metasyntactic, and vice versa. For example, `/d/` matches a literal `d`, but `/\d/` matches a single digit. On the other hand, + in a regex modifies the previous character, but `/\+/` matches a plus sign, +. The only exception is the # character, which can't be made to match literally by prepending a backslash, and must be quoted instead.

Not all meta characters have a special semantic meaning. For instance, the exclamation mark (!) does not have a metasyntactic meaning, and Perl 6 produces an appropriate error message:

```
$ perl6 -e '/!/'
===SORRY!=== Error while compiling -e
Unrecognized regex metacharacter ! (must be quoted to match
literally)
```

3.3 Anchors

Regexes search an entire string for matches. Sometimes this is not what you want. *Anchors* match only at certain positions in the string, thereby anchoring the regex match to that position.

The most common anchors are ^ and $. They match the start and end of a string, respectively. The regex /^go/ matches all strings starting with the letter g followed by o, for example gown. In contrast, /go$/ matches all strings ending with the letter o preceded directly by g, for example tango. Lastly, the regex using both anchors, /^go$/ only matches the string go.

Since whitespace is normally ignored, we could have written these regexes as / ^ go /, / go $ /, or / ^ go $ /.

The regexes / g ^ / and / $ g / can't match any string, because no string has a character before its start or after its end. Generally, Perl 6 does not warn or complain if you write such regexes, though future versions might generate a warning. If you ever need a regex that can never match, use <!> which achieves the same, but is more explicit.

A string can consist of multiple lines; the anchors ^^ and $$ match to the start and end of a line, respectively. The $$ anchor is special in that it matches before a newline character, or at the end of the string if there is no trailing newline character:

```
# Start-of-line positions that ^^ matches:
"␤ab\n␤de"
"␤ab\n␤de\n"

# End-of-line positions that $$ matches:
"ab␤\nde␤"
"ab␤\nde␤\n"
```

Anchors are *zero-width* regex elements. Hence they don't "use up" a character of the input string, that is, they do not advance the current position at which the regex engine tries to match. A good mental model is that they match between two characters of an input string (or before the first, or after the last character of an input string).

There are more anchors built into Perl 6 (Table 3-1).

Table 3-1. *Perl 6 Regex Anchors. ⏏marks positions where the anchor matches.*

Anchor	Description	Example Matches
^	Start of string	"⏏some\nlines"
^^	Start of line	"⏏some\n⏏lines"
$	End of string	"some\nlines⏏"
$$	End of line	"some⏏\nlines⏏"
<<	Left word boundary	"⏏some ⏏words"
«	Left word boundary	"⏏some ⏏words"
>>	Right word boundary	"some⏏ words⏏"
»	Right word boundary	"some⏏ words⏏"
<?wb>	Any word boundary	"⏏some⏏ ⏏words⏏!!"
<!wb>	Not a word boundary	"s⏏o⏏m⏏e w⏏o⏏r⏏d⏏s!⏏!"
<?ww>	Within word	"s⏏o⏏m⏏e w⏏o⏏r⏏d⏏s!!"
<!ww>	Not within word	"⏏some⏏ ⏏words⏏!⏏!⏏"

Word boundaries are boundaries between groups of alphanumeric characters and the underscore (_), and any other character. For example, the string "some-words_here" contains these word boundaries: "⏏some⏏-⏏words_here⏏". Note that there are no boundaries around the "_" character in this case; it's treated as if it were a letter.

3.4 Predefined Character Classes

As long as you have to spell out each character to match, regexes are very limited. *Character classes* loosen that requirement. For instance, the dot (`.`) stands for any single character.

You can use this to great effect to solve crosswords. Suppose you are searching for a five-letter word, and you know that the second letter is an `e`, and the last two are `rl`. That could be pretty much anything, right?

Once you have a list of words, you can test each word against the regex `/ ^ .e.rl $ /`, and so greatly reduce the number of words to consider.

Some Linux distributions come with a file `/usr/share/dict/words`, which contains an English word list, one word per line. Searching through it is a very short script:

```
for '/usr/share/dict/words'.IO.lines -> $word {
    say $word if lc($word) ~~ / ^ .e.rl $ /;
}
```

On my system (Ubuntu 16.04), this produces only two lines of output:

```
Pearl
pearl
```

They only differ in case, so for the sake of a crossword puzzle, they are the same.

How did that script work? The code `'/usr/share/dict/words'.IO` creates an `IO::Path` object, and calling the `.lines` method on it returns a sequence of lines from the file. The `for` loop iterates over these lines, which also happen to be words.

The command `say $word` prints the word, followed by a newline, but only if the condition of the `if` postfix is met. `lc($word)` returns `$word` in lowercase, and finally the rest of the line checks whether the lowercase word matches against the regex `/ ^ .e.rl $ /`.

Since character classes are very useful, Perl 6 has many of them. Some consist of a backslash followed by a single lowercase letter. In those cases, the uppercase version is the negation (Table 3-2).

Table 3-2. *Perl 6 Regex Predefined Character Classes*

Character Class	Description	Ex. of Matches	Ex. of Nonmatches
`.`	Any character	`a, 4, +`	
`\d`	Digit	`1, ٤`	`a, +`
`\D`	Not digit	`a, +`	`1, ٤`
`\w`	Word character	`a, 4,`	`+, /, " "`
`\W`	Not word	`+, /, " "`	`x, 4, ٤`
`\s`	Whitespace	`" ", "\t"`	`a, -, 4`
`\S`	Not whitespace	`a, -, 4`	`" ", "\t"`
`\n`	Logical newline	`"\n", "\c[LINE SEPARATOR]"`	`"\t", +`
`\N`	Not newline	`"\t", +, a`	`"\n"`
`\h`	Horizontal space	`" ", "\t"`	`"\n", a, 4`
`\H`	Not hor. space	`"\n", a, 4`	`" ", "\t"`

All of these classifications take into account the full Unicode character repertoire; so `\d` does not just match the digits from `0` to `9`, but also digits from all scripts and variants such as `٤ -- ARABIC-INDIC DIGIT FOUR`. In Rakudo 2017.05, which supports Unicode version 9, 580 different characters match the `\d` character class.

If you want to find all characters that match a character class, you can iterate over all Unicode characters. Here is an example that lists all characters matching \n:

```
for 0..0x1FFFF -> $c {
    if chr($c) ~~ /\n/ {
        printf "U+%05X - %s\n", $c,    $c.uniname
    }
}
```

This produces the output

```
U+0000A - <control-000A>
U+0000B - <control-000B>
U+0000C - <control-000C>
U+0000D - <control-000D>
U+00085 - <control-0085>
U+02028 - LINE SEPARATOR
U+02029 - PARAGRAPH SEPARATOR
```

`chr($c)`[1] translates the codepoint number $c to the actual character behind it. `printf`[2] is a routine for printing output formatted according to a template, and `uniname`[3] returns the name of the character from the Unicode character database.

User-Defined Character Classes

Instead of only relying on character classes baked into Perl 6 regexes, you can also specify your own. The simplest way is to enumerate them between <[and]>:

```
say 'perl' ~~ / <[aeiou]>/;     # Output: ｢e｣
```

[1] https://docs.perl6.org/type/Int#routine_chr
[2] https://docs.perl6.org/type/Str#sub_sprintf
[3] https://docs.perl6.org/type/Cool#method_uniname

Inside the character class, nonalphanumeric characters lose their special meaning, so `/ <[ " ']> /` matches either a single quote or a double quote. Exceptions are spaces, which are simply ignored, the closing square bracket] (which ends the character class and must be escaped with a backslash to be part of a character class), and the backslash. To match either an opening or a closing square bracket, use the regex `/ <[ [ \] ]> /`.

You can include predefined character classes inside a custom character class: `/ <[ \w $ - ]> /` matches a single character that's either a word character (`\w`), a hyphen, or the dollar sign.

Ranges of characters can simplify listing consecutive characters. For instance, `/ <[ 0..9 a..f A..F ]>` matches a hexadecimal character, in other words, any of these characters: `0123456789abcdefABCDEF`.

You can negate a character class by prepending a minus sign (`-`). Thus, to match any character except a double quote, use `/ <-[ " ]> /`. This is a special case of a more general syntax that lets you mix and match positive and negative character classes as well as predefined ones:

```
/ <[\d]-[78]+[abc]> /
```

This regex matches any digit except `7` and `8`, but it also matches the characters `a`, `b`, and `c`.

Finally, backslash escapes for single characters work inside character classes just like they do in double-quoted strings, so `<[ \c[ CHARACTER TABULATION] x ]>` and `<[ \t x ]>` both match either the letter `x` or the tabulator.

Unicode Properties

Perl 6 also offers access to character classes by referencing Unicode properties and categories.[4] To use the `Letter` property in a regex, you can write `<:Letter>` to match a single character that is a letter, or `<:!Letter>` as its negation—that is, a single character that is *not* a letter. Some of these properties have short forms, like `<:L>` for `<:Letter>`.

These properties are not limited to the Latin alphabet you are currently reading, but instead include the whole Unicode character database. So `<:Letter>` may match a Greek, Cyrillic, or Hebrew letter, or a letter from any script that has the concept of letters (Table 3-3).

Table 3-3. *Selected Unicode Properties from the General Category*

Property	Short	Examples
`Letter`	`L`	aΦЊ
`Uppercase_Letter`	`Lu`	AЊ
`Lowercase_Letter`	`Ll`	xήፊ
`Mark`	`M`	
`Number`	`N`	8¾
`Decimal_Number`	`Nd`	8
`Symbol`	`S`	$÷ɜ
`Math_Symbol`	`Sm`	+±♯
`Punctuation`	`P`	!@_

[4] http://unicode.org/reports/tr23/

3.5 Quantifiers

The regexes we've seen so far all match a fixed number of characters. This is about to change. A *quantifier* controls how often the previous regex element (an *atom*) matches, hence allowing for optional and repeated elements.

The + quantifier matches one or more repetitions of the thing that came before it. For instance, `/ ^ a+ $ /` matches the strings a, aa, aaa, aaaa, and so on.

Quantifiers bind very tightly, more so than concatenation of regex elements, thus `/ ab+ /` matches ab, abb, abbb etc. You can write that as `/a [b+]/` to remove any ambiguity.

If you want to match ab, abab, ababab instead, you can either use quotes or brackets to force the + to apply to more than one character: both `/ [ab]+ /` and `/ 'ab'+ /` match these strings. The former is more generic, since it applies not only to literals but also to other regex expressions. `/ [\d+ ',']+ /` matches a list of digits, each followed by a comma.

There are more quantifiers, as summarized in Table 3-4.

Table 3-4. *Perl 6 Regex Quantifiers*

Quantifier	Min Matches	Max Matches
?	0	1
*	0	infinite
+	1	infinite
**4	4	4
** 4..20	4	20
** 4..*	4	infinite

The most general form is the ** RANGE quantifier, where the RANGE can be of the form MIN..MAX for ranges with an upper bound, or MIN..* for repetitions without an upper limit:

```
/ ^ a ** 4 $ /;          # Matches exactly 4 a's
/ ^ a ** 2..4 $ /;       # Matches between 2 and 4 a's
/ ^ a ** 5..* $ /;       # Matches at least 5 a's
```

Greedy and Frugal Quantifiers

Quantifiers are *greedy* by default: they go for the largest number of repetitions that can possibly match.

You can observe this behavior by applying a regex to a string that has more than one possible way to match:

```
say "<a> b <c>" ~~ /"<" .+ ">"/;   # Output: 「<a> b <c>」
```

The regex here matches the full string, from the first < to the last >.

Should this not be what you want, you can rein in the quantifier's greediness by appending a question mark (?) to it:

```
say "<a> b <c>" ~~ /"<" .+? ">"/;   # Output: 「<a>」
```

The Perl 6 community calls this version a *frugal* quantifier, although this name isn't widespread in the general regex literature. Some call it a *lazy* quantifier.

In the case of the general quantifier, the ? goes before the range, like so: **?1..5.

Quantifiers with Separators

Parsing a list of things joined by a fixed separator is a common task. You can perform it with / <thing> [<separator> <thing>]* /, or if a trailing separator is allowed, / <thing> [<separator> <thing>]* <separator>?/. More modifications are necessary if you want to allow an empty list.

Perl 6 offers a shortcut: `/<thing>+ % <separator>/`, which matches a list of `<thing>`s separated by a `<separator>`. If a trailing separator is allowed, just change the `%` to a `%%`. This works for any quantifier. Thus, `<thing>* % <separator>` also matches an empty string.

For example, a list of numbers separated by commas can be parsed as

```
say '1,24,5' ~~ / [\d+]* % ',' /;   # Output: 「1,24,5」
```

When you combine a frugal quantifier with the separator feature, the frugal `?` comes first:

```
say '1,24,5' ~~ / [\d+]*? % ',' /;   # Output: 「」
```

3.6 Disjunction

"Hi," "Hello," and "Hey" might all be acceptable greetings in English. A regex matching such a greeting has to accept any of these three alternatives:

```
/ Hi | Hello | Hey /
```

The vertical bar `|` separates alternatives or branches of a *disjunction*. The branches don't have to be literals as in this example; they can be any regex.

Perl 6 being what it is, regex disjunctions come in two flavors. The single vertical bar flavor matches the branch that produces the longer match. If two or more matches are the same length, those branches starting with literals win, so for the regex `/a.|../` matching against the string `ab`, the first alternative is preferred over the second.

If you double the vertical bar, the alternatives are tried from left to right, and the first that matches is the winner:

```
say 'aab' ~~ / a+ | \w+ /;   # Output: 「aab」
say 'aab' ~~ / a+ || \w+ /;  # Output: 「aa」
```

In the preceding example, `a+` matches `aa`, while `\w+` can match the whole string, `aab`. In the case of `|` as the disjunction operator, the longer match wins, hence the regex matches the whole string. With `||`, the first part, `a+`, matches successfully, so there is no need to even try the second branch.

When you write alternatives, you are allowed to keep an empty first branch that Perl 6 ignores. This is purely for aesthetic reasons, so that you can write

```
/
    | first branch
    | second branch
    | third branch
/
```

instead of the visually somewhat less balanced

```
/
      first branch
    | second branch
    | third branch
/
```

3.7 Conjunction

You can think of `|` and `||` as a logical OR. But what about the AND operator? Most regex implementations omit it, but not so Perl 6. It's spelled `&` and, just like with disjunctions, there's a sequential variant `&&`. The difference between `&` and `&&` will become apparent when we talk about side effects in regexes in a later chapter.

The & operator is useful when you have a regex and want to constrain it further. As an example you might be searching for a phone number in a text document and already have a regex for a phone number, but you remember that there was a `17` sequence somewhere in the number you're looking for. You could search for such a number with this regex:

```
/ <phonenumber> & .* 17 .* /
```

The branches of an & conjunction must match the same part of the string, so here the `17` is padded with `.*` to stretch the match of the right branch to the match of the left branch.

You can apply the same technique to exclude certain characters from matching. For instance, a phone number that doesn't contain the digit `9` could be written as

```
/ <phonenumber> & <-[9]>* /
```

3.8 Zero-Width Assertions

How could you implement your own anchor?

The regex constructs we've talked about so far don't allow you to do that, because they all need to consume characters to decide whether or not they match.

A *zero-width assertion* turns another regex into an anchor, making them consume no characters of the input string. There are two variants: *look-ahead* and *look-behind* assertions.[5]

[5]Technically, anchors are also zero-width assertions, and they can look both ahead and behind.

The look-ahead assertion is spelled `<?before regex>`, and turns `regex` into an anchor. Thus, if you want to match a number followed by a unit (such as `kB` for kilobyte or `MB` for megabyte), but not the unit itself, you can do so with such a regex:

```
say 'up to 200 MB' ~~ / \d+ <?before \s* <[kMGT]>? B > /;
# Output: ｢200｣
```

Here, the regex `\s* <[kMGT]>? B` matches optional whitespace, followed by the unit `B`, `kB`, `MB`, `GB`, or `TB`. The `<?before ...>` around it makes the match only succeed if the regex for the unit matches successfully. Note that the assertion did not advance the regexes' position in the string, so the string matching the unit is not included in the string that the whole regex matches.[6]

You can negate the look-ahead by using a `!` instead of a `?`. So to match a number that is not followed by such a unit, you can use the regex:

```
say 'up to 200 MB' ~~ / \d+ <!before \s* <[kMGT]>? B > /;
# Output: ｢20｣
```

In this case, the regex matches just the string `20`, because that's a number not followed directly by a unit. If you don't want it to match this input string at all, you can use word-boundary assertions around the number:

```
say 'up to 200 MB' ~~ / « \d+ » <!before \s* <[kMGT]>? B > /;
# Output: Nil
```

Instead of looking forward in the string, we can also write regexes that look back from the current position. To match a number that comes after a comma, you can write

```
say '200,50' ~~ / <?after \, > \d+ /;    # Output: ｢50｣
```

[6] If this doesn't make sense to you, don't worry; it will be clearer after the discussion of match objects and Regex Mechanics in later chapters.

Again, you can negate the condition by writing a ! instead of the ?:

```
say '200,50' ~~ / <!after \, > \d+ /;    # Output: ｢200｣
```

Using positive and negative look-ahead and look-behind assertions, you have the capabilities to reimplement the built-in anchors,[7] and write your own. For instance, ^, the start of the string, could also be written as `<!after .>`; it only matches if no character comes before it. Likewise, the left word boundary, «, could be written as `<!after \w> <?before \w>`. A left number boundary would then be `<!after \d> <?before \d>`.

Note that `<!after \w>` and `<?after \W>`, though similar, are not quite the same. The difference is that `<!after \w>` matches at the start of the string, whereas `<?after \W>` does not, because it requires a nonword character to match.

3.9 Summary

The basic building blocks of regexes are literals, character classes, anchors, quantifiers, disjunctions (alternatives), and conjunctions (overlaps). They form the basis for searching, validating, and parsing with Perl 6 regexes.

[7]There should be no need for that. Still, it's good to have that power.

CHAPTER 4

Regexes and Perl 6 Code

Now that you know the basic building blocks of regexes, we'll explore different ways of using regexes in Perl 6 code.

4.1 Smart-Matching

The *smart-match* operator ~~ is a general comparison operator, and the object on its right-hand side determines the semantics of the comparison. For instance, if the right-hand side is a number, ~~ performs a numerical comparison. If the right-hand side is a type, it checks if the left-hand side is of that type (or a subtype thereof).

Of special interest to us is the case when the right-hand side is a regex. In this case, the smart-match operator interprets the left-hand side as a string and searches for a match using the regex:

```
my $str = "If I had a hammer, I'd hammer in the morning";
say $str ~~ /h.mm\w*/;      # Output: 「hammer」
say $str ~~ /hummer/;       # Output: Nil
```


M. Lenz, *Parsing with Perl 6 Regexes and Grammars*,
https://doi.org/10.1007/978-1-4842-3228-6_4

The return value of the match is either `Nil` if the match failed, or a *match object*[1] if it succeeded. If you use a match object in a Boolean context (like the conditional of an `if` statement), it evaluates to `True` (whereas `Nil` evaluates to `False`); in a string context (or when forcing it into string context by calling the `.Str` method), it evaluates to the part of the string that the regex matched. Corner brackets (`「` and `」`) in the output from `say` indicate a `Match` object.

Apart from smart-matching, you can also search for a match in a string with the `.match` method[2]:

```
my $str = "If I had a hammer, I'd hammer in the morning";
say $str.match(/h.mm\w*/);  # Output: 「hammer」
```

4.2 Quote Forms

So far we've seen regexes delimited by two slashes, `/.../`, but there are other ways to write a regex. These are all equivalent:

```
/ a.b /;
rx/ a.b /;
rx{ a.b };
rx! a.b !;
regex { a.b }
```

If you use `rx` to write a regex, you can use almost any nonword character as a delimiter (with the exception of whitespace, the colon (`:`), and the hashmark (`#`)).

The forms introduced in the preceding return an object of type `regex`, which you can store in a variable, or use in a smart-match operation.

[1] https://docs.perl6.org/type/Match.html
[2] https://docs.perl6.org/type/Str#method_match

In addition, there is a form that uses `m` (for *match*) instead of `rx`. Instead of returning a regex object, it immediately matches the regex against the special variable `$_`:

```
$_ = 'abc';
if m/b./ {
    say "match";
}
```

Since the smart match operator ~~ automatically sets `$_` to the left-hand side of the expression, you can use `m/.../` in a smart-match:

```
say "abc" ~~ m/b/;          # Output: ｢b｣
```

4.3 Modifiers

Modifiers change the way regexes work. They come in two categories. If you think of regexes as small programs that are compiled to byte code,[3] then some modifiers affect the compilation and others affect the way you call that program.

For example, the `:global` modifier (short form `:g`) belongs to the latter category; it instructs the regex to search for all nonoverlapping matches in the string:

```
my $str = "If I had a hammer, I'd hammer in the morning";
say $str.match(:global, /h.mm\w*/).join('|');
# Output: hammer|hammer
```

[3]Which, in fact, they are.

An example of the first category would be `:ignorecase` (short form `:i`), which instructs the regex engine to ignore case, so an uppercase character in the regex can match its lowercase equivalent character, and the other way around[4]:

```
say 'Hello, world'.match(/:i hello/);    # Output: ｢Hello｣
```

As shown in the preceding, modifiers affecting the compilation of the regex may be used inside the regex itself. Modifiers that affect the behavior of the compiled regex always go on the outside.

Since the `m/.../` form matches immediately, you can also add modifiers to it that affect the runtime behavior of regexes:

```
say ('abc' ~~ m:g/./).elems;    # Output: 3
```

Here `m:g/./` returns a list of three matches, so the `.elems` method returns the number 3. (See Table 4-1.)

Table 4-1. *Modifiers That Affect the Compilation of a Regex, or Parts Thereof*

Long form	Short form	Description
`:ignorecase`	`:i`	Match regardless of letter case
`:ignoremark`	`:m`	Match regardless of mark or combining character
`:sigspace`	`:s`	Treat whitespace as significant
`:ratchet`	`:r`	Disable backtracking

[4]Technically, there's also title case, where special provisions are made when uppercasing the first letter of each word, for example in a headline. The modifier `:ignorecase` also works with title case.

The `:ignoremark` or `:m` modifier allows you to match by the base character only, and ignore diacritics and other marks that decorate a character. With `:m` in effect, `/a/` matches a, à, á, â, ã, ä, å, ā, and 22 other characters.

The `:sigspace` modifier makes whitespace in the grammar match whitespace in the string, though not necessarily with a one-to-one mapping. The `:ratchet` modifier disables backtracking, so if the regex engine succeeds in matching a string in one way, it won't try to do so again in a different way. We will discuss both modifiers later in more detail.

These modifiers can also go into the middle of a regex, or be limited to a part of a regex by a square bracket `[...]` group:

```
/ ab :i cd /;          # match only the cd case-insensitively
/ [:i ab] cd /;        # match only the ab case-insensitively
```

You can also disable a modifier by adding an exclamation mark after the colon. So the last regex could have been written as `/:i ab :!i cd/`. (See Table 4-2.)

Table 4-2. *Modifiers That Affect the Runtime Behavior of a Compiled Regex*

Long form	Short form	Description
`:global`	`:g`	Find all matches that do not overlap
`:overlap`	`:ov`	Find all matches with different starting positions
`:exhaustive`	`:ex`	Find all possible matches
`:continue(5)`	`:c(5)`	Start searching from position 5
`:pos(5)`	`:p(5)`	Start from, and anchor at, position 5
	`:x(5)`	Attempt 5 matches, and fail if there are fewer
`:nth(5)`		Return the 5th match

All the forms that take an argument, here 5, can also accept a variable as their argument. String positions start counting at zero from the start of the string, just like string indexing with `substr`.[5]

If you search for multiple matches with `:global`, the search for the second match starts where the first match ended. With `:overlap`, the search for the second match starts one character after the first match started, so the matched string from both matches can overlap. The `:exhaustive` modifier finds all possible matches, even if several matches start from the same position.

If you use the `:continue` and `:pos` modifiers without an argument, they default to the position where the previous match left off.

4.4 Comb and Split

If you want to find all occurrences of a regex in a string, but you're only interested in the strings of the result (not in match objects), you can use the `comb`[6] method to find them:

```
my @numbers = "1308 5th Avenue".comb(/\d+/);
say @numbers;       # Output: [1308 5]
```

If instead you are interested in the parts of a string that don't match a regex, the `split`[7] method will work for you:

```
my ($city, $area, $popul) = 'Berlin;891.8;3671000'.split(/';'/);
say $area;          # Output: 891.8
```

[5] https://docs.perl6.org/type/Str#routine_substr
[6] https://docs.perl6.org/type/Str#routine_comb
[7] https://docs.perl6.org/type/Str#routine_split

Here, the regex contains only a single literal; in this case, you can pass the literal string directly to `split`, such as in `'Berlin;891.8;3671000'.split(';');`.

Both the `comb` and `split` methods accept a limit as a second, optional argument:

```
my ($city, $rest) = 'Berlin;891.8;3671000'.split(';', 2);
say $city;          # Output: Berlin
say $rest;          # Output: 891.8;3671000
```

The `split` method is useful for parsing simple file formats, like comma-separated files without quotes.

4.5 Substitution

You can use regexes not only to match text, but also to transform it.

With the `subst`[8] method you can replace the part of a string that a regex matched with another string. In the simplest case, the replacement is a string constant:

```
say '42 eur'.subst( rx:i/ « eur » /, '€');        # Output: 42 €
```

If the replacement is the empty string, the substitution deletes the matched string:

```
say '42 €'.subst(/\s+/, '');                      # Output: 42€
```

As with regex matches, you can use modifiers together with substitution. For example, `:global` replaces all occurrences of the regex match (instead of just the first one, as happens without the modifier):

```
say '1 2 3'.subst(/\s+/, '');                     # Output: 12 3
say '1 2 3'.subst(:g, /\s+/, '');                 # Output: 123
```

[8] https://docs.perl6.org/type/Str#method_subst

If you want the replacement string to depend on the matched value, you can pass a subroutine or a block as the replacement part, and it receives the match object as its argument:

```
say "9 of spades".subst(/\d+/, -> $m { $m + 1 });
    # Output: 10 of spades
```

In the block of such a substitution, match variables like `$0`, `$1`, etcetera (more on those in the next chapter) do not work, unless you explicitly declare the match variable `$/` as a parameter of this block:

```
say "9 of spades".subst(/(\d+)/, -> $/ { $0 + 1 });
    # Output: 10 of spades
```

There is also a syntactic variant for substitutions that modify a variable in-place:

```
my $var = '1 2 3';
$var ~~ s:g/ \s+ //;
say $var;                                    # Output: 123
```

Note that in this case the part between the first and the second slash (`/`) is a regex, but the part between the second and the third slash is a string. This implies that whitespace is ignored in the first part, but relevant in the second. Inside the replacement string, you can use `$0`, `$1`, `$2`, etcetera to refer to captures (see the next chapter for more on captures):

```
my $var = '"fantastic", she said';
$var ~~ s:g/ \" (.*?) \" /«$0»/;
say $var;           # Output: «fantastic», she said
```

The replacement part can also be a Perl 6 expression if you use `[...]` or `{...}` to delimit the regex:

```
my $ad = 'Buy now! USD 10 per book. Prices double soon to 20.';
$ad ~~ s:g[ \d+ ] = 2 * $/;
say $ad;
```

where $/ refers to the match object. Note that in this case, the assignment operator is used before the right-hand side.

This produces the output

```
Buy now! USD 20 per book. Prices double soon to 40.
```

When you substitute a zero-width match, the substitution becomes an insertion operation:

```
my $input = "It's just a jump to the left.
And then a step to the right.";

$input ~~ s:g/ <?before jump> /⇑/;
$input ~~ s:g/ <?before left> /←/;
$input ~~ s:g/ <?after right> /→/;

say $input;
```

Here <?before ...> and <?after ...> turn the regexes inside them into zero-width regexes, so the substitution commands do not replace the matched text (like "jump"), but rather they insert the replacement part before or after the match:

```
It's just a ⇑jump to the ←left.
And then a step to the right→.
```

You can combine normal and zero-width matches. For instance, this statement substitutes all numbers that are followed by the unit MB or GB by 500:

```
my $input = '2 links with 75MB each';
say $input.subst(:g, / \d+ <?before <[MG]> B>/, 500);
    # Output: 2 links with 500MB each
```

4.6 Crossing the Code and Regex Boundary

Regexes and regular Perl 6 code compile to the same byte code, and you can mix Perl 6 code and regexes.

The most obvious interaction is storing regex objects in variables and using them in regular code:

```
my $word-regex = /\w+/;
say "Hello, world" ~~ $word-regex;    # Output: 「Hello」
```

This way you can give regexes a name, but also do all other things that you can do with variables: put them into data structures, return then from functions, and so on.

It works the other way around too. Variables can be part of a regex:

```
my $audience = 'world';
my $greeting = 'Hello';

if "Hello, world" ~~ / $greeting ', ' $audience / {
    # this branch is executed
}
```

If a variable contains a string, it is always matched literally; if it contains a regex, it matches as a regex.

If you want to interpret the contents of a variable as a regex, you have to include it in angle brackets:

```
my $audience = "\\w+";
my $greeting = 'Hello';

if "Hello, world" ~~ / $greeting ', ' <$audience> / {
    # this branch is executed
}
```

This example also demonstrates that backslashes must be doubled within quoted strings—in this case assigned to the variable `$audience`—but not in regexes. In a double-quoted string, a backslash introduces an escape sequence (such as `\n` for a newline, or `\t` for a tabulator). Using the syntax `<$audience>` interprets the contents of variable `$audience` as a regex.

Using an array variable inside a regex is equivalent to matching each element of the array as an alternative; thus

```
my @numbers = 'one', 'two', 'three';
my $regex = / @numbers /;
```

is equivalent to writing

```
my $regex = / [ 'one' | 'two' | 'three' ] /
```

except that when using an array variable, it's easier to supply the values programmatically.

Finally, you can include Perl 6 code blocks in regexes, simply by embedding them in curly braces. This can be useful for building data structures such as symbol tables during the execution of a regex.

For instance, we could count how many numbers appear in a string:

```
my $count = 0;

my $str = "between 23 and 42 numbers";

if $str ~~ / [ \d+ { $count++ } \D* ]+ / {
    say $count;     # Output: 2
}
```

Here the block { $count++ }, which increments the variable $count by one, is executed after the \d+ part of the regex. While this is a somewhat constructed use case, we will see very practical applications of code blocks in regexes in later chapters.

Another good use case for blocks in regexes is adding print statements for debugging, to see how a regex matched.

Here we attempt to match a floating-point number with a regex:

```
say '1.0e42' ~~ / ^ \d+ ['.' \d+]?  [e|E \d+]? $ /;
```

This regex match fails, and to see how far the regex gets, you can remove the question marks that make pieces optional and add some code blocks to show the incremental progress:

```
'1.0e42' ~~ / ^
    \d+        { say "integer: '$/'" }
    ['.' \d+]  { say "decimal place: '$/'" }
    [e|E \d+]  { say "exponent: '$/'" }
    $ /;
```

which produces the output

```
integer: '1'
decimal place: '1.0'
exponent: '1.0e'
```

We can see that the last part of the regex matches only the e, not e42. Closer scrutiny reveals that the alternative in [e|E \d+] extends to the \d+, which wasn't intended. Adding a level of grouping, [[e|E] \d+], fixes the regex.

The code blocks make use of the special variable `$/`, which contains the match (or at least the partial match, as far as it has progressed at that point).

These code blocks are executed purely for their side effects. You can use the form `<?{ ... }>` instead to influence the regex match. If the code inside that block returns a false value, the match fails[9]:

```
my $one-byte = / ^ \d ** 0..3 $ <?{ $/.Int <= 255 }> /;
for 0, 100, 255, 256, 1000 -> $num {
    if $num  ~~  $one-byte  {
        say $num;
    }
}
```

This example features a regex that matches a number between 0 and 255 (which might be useful for validating IPv4 addresses, for example), and only prints the numbers 0, 100, and 255.

The `$/.Int` call converts the matched string to an integer, and the comparison `<= 255` returns `True` if and only if that number is at most 255.

[9]Or at least it tries to match in a different way; see the discussion on backtracking in Chapter 6 for more details.

Table 4-3. *Ways to embed Perl 6 code into regexesx*

Syntax	Description
`{ CODE }`	Runs Perl 6 code; no effect on regex match.
`<?{ CODE }>`	Code needs to return a true value for the match to succeed.
`<!{ CODE }>`	Code needs to return a false value for the match to succeed.
`<{ CODE }>`	Result of code is interpreted as a regex.
`<$STRING>`	Interprets `$STRING` as regex source code.

4.7 Summary

In this chapter, we've seen how to use regexes in Perl 6 code and the other way around. You can store regexes in variables and also use variables inside regexes.

In the next chapter, we will explore how to extract information from regex matches or parts thereof.

CHAPTER 5

Extracting Data from Regex Matches

When you use a regex to validate input, it is enough to know whether a given string matches the regex or not. In many other cases, we want to extract more data from the string and the regex match.

For instance, if you want to parse an INI file (the configuration file format common on Microsoft Windows), you might be interested in the section headers and the key/value pairs inside a section; however, things like the square brackets around the section headers aren't so exciting.

5.1 Positional Captures

You can extract relevant data by instructing a regex to *capture* some data, by surrounding a part of a regex with a pair of parentheses. The string that this part of the regex matched is then available separately:

```
if "Hello, world" ~~ / (\w+) ', ' (\w+) / {
    say "Greeting: $0";     # Output: Greeting: Hello
    say "Audience: $1";     # Output: Audience: world
}
```


M. Lenz, *Parsing with Perl 6 Regexes and Grammars*,
https://doi.org/10.1007/978-1-4842-3228-6_5

The captures are numbered from left to right, starting with zero. The string that the regex within the first pair of parentheses matched is available in the special variable `$0`, the string from the second pair of parentheses in `$1`, and so on.

Since the numbering of these captures relies on their relative position within the regex, we call them *positional captures*.

5.2 The Match Object

The result of each successful regex match is an object of type `Match`.[1] When you match a regex with the smart-match operator `~~`, the regex object is also available in the special variable `$/`.

The match object offers a wealth of information about the regex match: `$/.orig` contains the string that the regex matched against, `$/.from` and `$/.to` the start and end positions of the match, and using it in a string context (or by explicitly calling the `.Str` method on it) returns the matched string.

But there is more: when you use the match object like an array, it gives you access to the matches from the captures. `$0` is actually an alias for `$/[0]`, `$1` for `$/[1]`, etcetera.

The captures themselves are `Match` objects again:

```
my $input = "There are 9 million bicycles in Beijing";
if $input ~~ /(\d+) \s+ (\w+)/ {
    say $0.^name;       # Output: Match
    say $1;             # Output: ｢million｣
    say $1.from;        # Output: 12
    say $1.to;          # Output: 19
}
```

[1] https://docs.perl6.org/type/Match.html

Nesting of Captures

If you nest captures inside other captures, the structure of the match objects corresponds to the nesting of the captures:

```
'abcdef' ~~ /(.) (b(c)(d(e))) /;
say $0.Str;          # Output: a
say $1.Str;          # Output: bcde
say $1[0].Str;       # Output: c
say $1[1].Str;       # Output: de
say $1[1][0].Str;    # Output: e
```

The match object is in fact a tree of match objects.

As you can see in the preceding example, the numbering of captures is per nesting level; there are five pairs of parentheses in the regex, but only two of them are top-level, and so there are only `$0` and `$1` after a successful match, no `$2` or higher. Instead, each level of nesting also introduces match numbers starting from `0` again. This is a departure from Perl 5 and PCRE semantics, where the matches are continuously numbered.

Quantified Captures

When a capture is subject to a quantifier, the corresponding capture in the match object becomes an array[2] of `Match` objects:

```
if "127.0.0.1" ~~ /(\d+)**4 % "."/ {
    say $0.elems;       # Output: 4
    say $0[3].Str;      # Output: 1
}
```

[2] https://docs.perl6.org/type/Array

The only exception is the ? quantifier, which does not produce an array for the capture.[3] Instead its capture is a `Match` object when it matches once, and `Nil` if it matches zero times.

5.3 Named Captures

When you have more than two or three captures, you can lose track of which number refers to which capturing group. To relieve that burden, and to make the code more robust to refactoring, you can use *named* captures instead:

```
my $str = 'Hello, World';
if $str ~~ / $<greeting>=[\w+] ', ' $<audience>=[\w+] / {
    say $<greeting>.Str;        # Output: Hello
    say $<audience>.Str;        # Output: World
}
```

`$<thename>` refers to the named capture both inside and outside the regex. Inside the regex, you can assign to it, outside you can access the corresponding match object. Outside the regex, `$<thename>` is actually a shortcut for a name-based access of the match object, `$/<thename>` or `$/{ 'thename' }`. This scheme for accessing named captures follows the same syntax as accesses to hash elements.

If you use the same name twice (or more often) within the same regex, the capture becomes an array again.

[3]Early versions of Perl 6 created an array capture for all quantified captures, even with the ? quantifier. It confused most users.

In a later chapter, we'll talk about named regexes in greater detail. For now it suffices to say that an easier way to create a named capture is to simply call a regex by name:

```
my regex byte {
    \d ** 1..3
    <?{  $/.Int <= 255 }>
}

my $str = '127.0.0.1';
if $str ~~ / ^ <byte> ** 4 % '.' $ / {
    for $<byte>.list -> $byte {
        say $byte.Str;
    }
}
```

Here `<byte>` calls the named regex `byte` and automatically captures the string matched by this subregex under the same name, `byte`. You can rename the capture with the syntax `<alias=originalname>`:

```
my $str = 'Hello, World';
my regex word { \w+ };

if $str ~~ /<greeting=word> ', ' <audience=word>/ {
    say $<greeting>.Str;        # Output: Hello
    say $<audience>.Str;        # Output: World
}
```

5.4 Backreferences

Backreferences allow you to access a capture inside the regex, matching the string that a previous part of a regex captured. There are several cases where this can be useful, such as when searching for accidentally duplicated words in a text. The first attempt would be a regex like this: `/ (\w+) \s+ $0 /` This regex matches a word (`\w+`) and captures it into `$0`, followed by whitespace (`\s+`), followed by the original word (`$0`). However this doesn't work as intended. Used against the string `the next thing`, it matches `t t`, because `\w+` is happy to match just a single character.

To make it work, we can restrict matches to whole words by including word boundary assertions: `/ « (\w+) \s+ $0 » /` (remember that « matches a left word boundary and » matches a right word boundary).

Now if you test it against the string `"the quick brown fox jumps over the the lazy black dog"`, it correctly matches the doubled `"the"`.

Backreferences also work with named captures; the previous example could have been written as `/« $<word>=\w+ \s+ $<word> »/`.

Another common use case for backreferences is finding something like a quoted string, where you don't care much about the actual quoting character as long as it's the same on both ends.

Note that backreferences are firmly outside the world of regular languages as defined in computer science; you need more powerful formalisms for that.

Excursion: Primality Test with Backreferences

This section is not relevant to any practical application, but gives insight into the crazy things that people have done with regexes. You can safely skip it if you're reading this merely for practical reasons.

A prime number is an integer that is only divisible by number 1 and by the number itself.

You can use regexes to test if a number is a prime number.[4] First we have to encode the number by creating a string of identical characters (for example a), where the number of characters in the string is identical to the number. So 2 would be encoded as aa, 5 as aaaaa, and so on. In Perl 6, we can do this with the string repetition operator, x:

```
my $encoded = 'a' x 5;
```

We then try to find a factor larger than one that evenly divides the number. In our encoded string, this is a substring of at least two characters which, when repeated often enough (but at least once), reproduces the original encoded string:

```
for 2..15 -> $number {
        my $encoded = 'a' x $number;
        if $encoded ~~ / ^ (a ** 2..*) $0+ $ / {
            say "$number is not a prime, a factor is ", $0.chars;
        }
        else {
            say "$number is a prime";
        }
}
```

This produces the following output:

```
2 is a prime
3 is a prime
4 is not a prime, a factor is 2
5 is a prime
6 is not a prime, a factor is 3
7 is a prime
```

[4]This primality test is commonly attributed to Abigail, a prolific Perl hacker. See for example http://neilk.net/blog/2000/06/01/abigails-regex-to-test-for-prime-numbers/.

```
8 is not a prime, a factor is 4
9 is not a prime, a factor is 3
10 is not a prime, a factor is 5
11 is a prime
12 is not a prime, a factor is 6
13 is a prime
14 is not a prime, a factor is 7
15 is not a prime, a factor is 5
```

For instance, if `$number` is 6, `$encoded` becomes `"aaaaaa"`. The regex engine then finds a match when the first group, `(a ** 2..*)`, matches three a's, aaa. Accordingly, `$0+` produces another three a's, totaling six a's. In the case of a prime number, the regex engine is unable to find such a partition, causing the match to fail.

Since the ** quantifier is greedy, the regex engine finds the largest prime factor first, so it identifies 5 as a prime factor of 10. If you change it to the **? quantifier (so the regex becomes `/ ^ (a **? 2..*) $0+ $ /`), it finds the smallest prime factor.

Note that this is a very inefficient way to test for primality, mostly because the regex engine is not optimized for this kind of task, uses a very inefficient number representation, and has to try all possible factors, even ones that would be easy to rule out by a smarter algorithm.

Nonetheless, the regex engine manages to test numbers up to fifty thousand in a few seconds. Perl 6 has a built-in way to test for primality that is much faster and less memory hungry:

```
say 50101.is-prime;                    # Output: True
```

5.5 Match Objects Revisited

As mentioned before, successful regex matches return match objects. In addition to the position and length of the matched substring, the match object also stores all positional and named captures. To get a list of all positional captures, you can call the `.list` method. Likewise, the `.hash` method returns the named captures as a *hash*. A hash is a data structure with mappings from names or keys to values; Python calls it a *dict* or *dictionary*, JavaScript an *object*.

When a match object is printed with the `say` function, the output is the matched string delimited by ⌜ ⌟:

```
say "abc" ~~ /../;      # Output: ⌜ab⌟
```

If the regex includes captures, they are printed on separate lines, indented by a single space. Thus

```
say "abc" ~~ /.(.)(.)/
```

produces the output

```
⌜abc⌟
 0 => ⌜b⌟
 1 => ⌜c⌟
```

where `0` and `1` are indices of the captures. Nested captures produce more indentation, so `"abc" ~~ /.(.(.))/` prints

```
⌜abc⌟
 0 => ⌜bc⌟
  0 => ⌜c⌟
```

Named captures are distinguished simply by their name, hence `say "abc" ~~ /.(.$<char>=[.])/` prints

```
「abc」
 0 => 「bc」
  char => 「c」
```

5.6 Summary

Captures allow you to extract data from successful regex matches. You can instruct the regex engine to capture data by adding a pair of round parentheses around a part of a regex, by using the `$<name>=[...]` syntax, or by calling a named regex: `<name>`.

Both regex matches and captures produce match objects; the topmost match object thus becomes a tree of submatches.

Backreferences allow you to match exactly what a previous part of a regex matched, and you can abuse that feature to build a simple primality test.

CHAPTER 6

Regex Mechanics

Regexes can seem magical at times. They often have many different ways of matching a string. How are matches found? Which one is found first?

This chapter demystifies the magic by explaining the mechanics that a regex engine uses. Understanding these mechanics is key to writing robust regexes that match what you want, no more and no less.

Our discussion is slightly complicated by the two different paradigms implemented in Perl 6 regexes: declarative matching with finite state machines, and the more powerful backtracking engine.

6.1 Matching with State Machines

When computer scientists talk about regular languages, they also describe a mechanism to efficiently recognize whether a particular string is part of the language. In Perl 6 this is simply if the string matches the regular expression.

Deterministic State Machines

The mechanism used to recognize matching strings is a *finite state machine*. It is a set of states with arrows between them. Each arrow is labeled with a single character. The machine reads each character from the input string separately and follows the arrow with the same label as the current character. If there is no arrow labeled with the current character, the match fails immediately.

M. Lenz, *Parsing with Perl 6 Regexes and Grammars*,
https://doi.org/10.1007/978-1-4842-3228-6_6

Some states are labeled as *accepting* states. After the last character from the input string is read, the match succeeds if the final state is an accepting state. Otherwise, it fails (Figure 6-1).

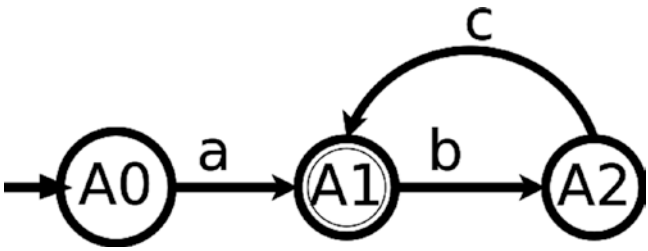

Figure 6-1. *A simple deterministic finite automaton. The accepting state A1 is marked by a doubled circle.*

Consider the example automaton. The starting state is `A0`, and from there on it needs to read the character `a` to get to the accepting state `A1`. It can then read a `b` followed by a `c` to get back to the accepting state, so the strings it accepts are the same as the regex `/ ^ a [bc]* $ /.`[1] Let's call this automaton A.

For each state, there is at most one arrow leaving that state with a certain character. Hence, the machine is in only one state at a time, and we call it a *deterministic* finite automaton, or DFA for short.

A DFA can be implemented very efficiently in code. For each state, you need a lookup table that maps incoming characters to the next state, which means you only have to spend a fixed number of steps on each character of the input string.

As demonstrated by the previous automaton, you can use it to match literals and you can implement the * quantifier with arrows going backward. It also supports alternatives, although they can be a bit trickier.

[1]The finite state machine formalism assumes that you always want to match the whole string, hence the anchors. If you want to just search in a string, you have to append and prepend .* to your regex.

First, let's modify the original automaton to match `/ ^ a [bca]* $ /` instead (Figure 6-2).

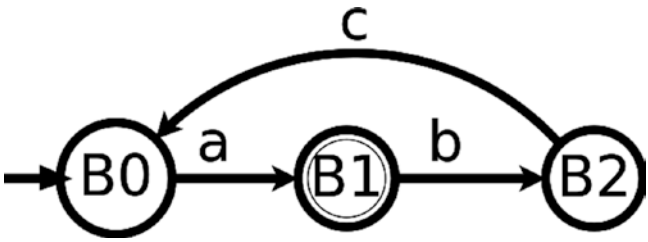

Figure 6-2. *A simple automaton for matching the regex `/ ^ a [bca]* $ /`*

We call this automaton B.

Now we want to construct an automaton for matching the strings that either A or B matches. We can do this by creating an automaton that runs both automata at the same time. The starting states from the two automata are `A0` and `B0`, so let's call our new starting state `A0B0`. From this starting state, reading an `a` moves automaton A from `A0` to `A1` and automaton B from `B0` to `B1`, so the next state is `A1B1`. Since at least one of the original states is accepting, so too is `A1B1` (Figure 6-3).

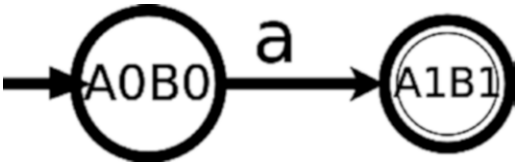

Figure 6-3. *Partially constructed automaton for matching A or B, first step*

The third state is pretty obvious as well: reading a `b` from either `A1` or `B1` moves us to `A2` and `B2`, respectively (Figure 6-4).

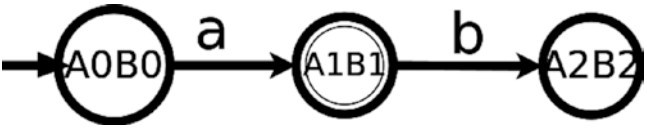

Figure 6-4. *Partially constructed automaton for matching A or B, second step*

Now it starts to get interesting: on reading character `c`, A transitions to state `A1`, but B to `B0`. We need a state for that, and we call it `A1B0`. Because `A1` is an accepting state, `A1B0` is also accepting (Figure 6-5).

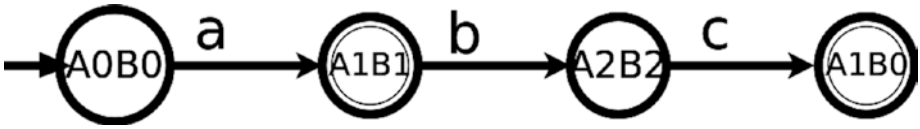

Figure 6-5. *Partially constructed automaton for matching A or B, third step*

What happens if the next character is an `a`? Automaton A rejects the input, and automaton B moves to state `B1`. So we can duplicate automaton B at this point (Figure 6-6).

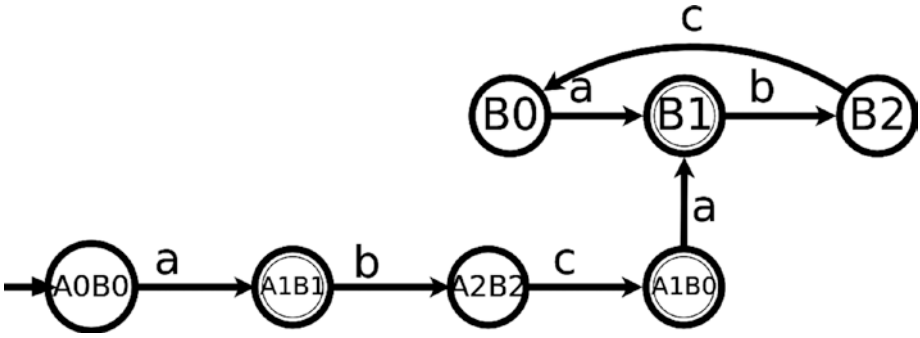

Figure 6-6. *Partially constructed automaton for matching A or B, fourth step*

Likewise, reading character `b` from state `A1B0` makes automaton B fail and moves automaton A into state `A2`. Thus we can copy states `A1` and `A2` into the new automaton (`A0` is not reachable from there), and we get our final result (Figure 6-7).

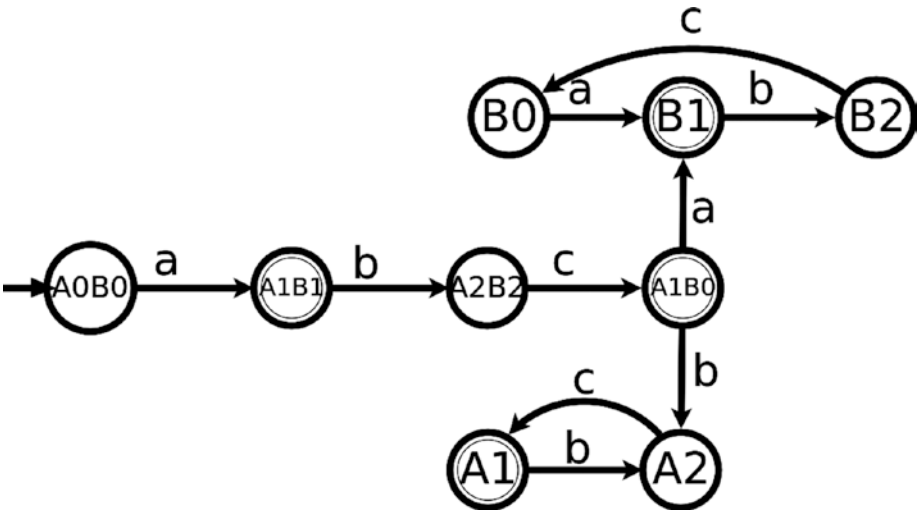

Figure 6-7. *Fully constructed automaton for matching A or B*

This mechanism is called the Powerset construction,[2] because the set of new states is the powerset of the source automata.

In our example, the resulting automaton has nine states, whereas A and B only had three states each. If we had to build the disjunction of several automata, in the worst case the number of states in the resulting automaton is proportional to the product of the number of states of each input automaton. Or phrased differently, the number of states can grow exponentially with the number of characters of the regex we want to model.

[2] https://en.wikipedia.org/wiki/Powerset_construction

Nondeterministic State Machines

This state explosion limits the practicability of DFAs in regex matching code. Instead, regex implementations often use *nondeterministic* finite automata, or NFAs for short. NFAs allow multiple arrows labeled with the same character to depart from the same state. The nondeterministic part is that the regex engine has to guess which arrow to follow, or it simply follows all of them, leading to a model where the machine can be in more than one state at a time.

If you think of the states and labeled arrows as a game board, starting a match places a chip on the initial state. When you read an input character, you follow all relevant arrows and place a chip on each target state. Finally, you remove the chips from the previous states.

To make it even easier to construct NFAs, people often allow so-called *epsilon* (ε) transitions. The NFA follows these arrows without consuming an input character. Or phrased differently: if you have an ε transition from state `C0` to `C1`, you place a second chip on `C1` as soon as you place one on `C0`.

With these ε transitions, creating a disjunction of two automata becomes easy. You just add a new starting point, `C0`, and add ε transitions from it to the starting points of the automata you combine (Figure 6-8). Voilà:

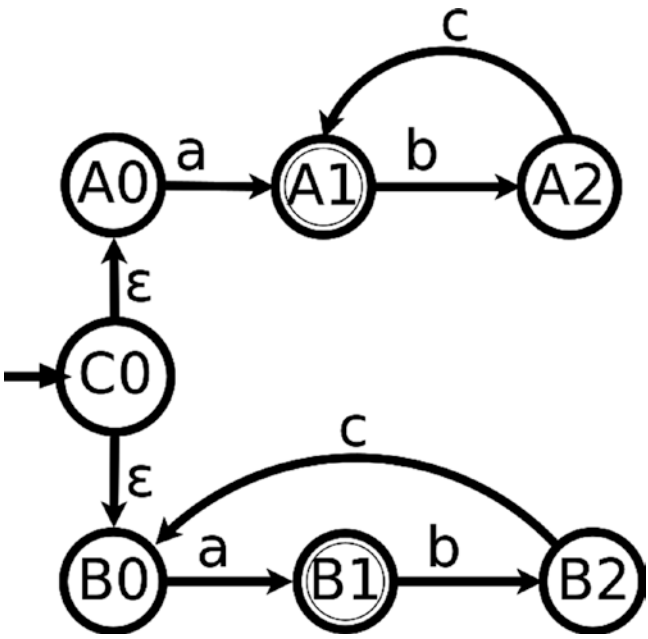

Figure 6-8. *Nondeterministic finite automaton for matching A|B*

Deterministic and nondeterministic finite automata are equally powerful. For each NFA, you can construct a DFA that matches exactly the same strings. In reverse, every DFA is also an NFA, because the rules for NFAs are looser than for DFAs.

Building an NFA from a regex is typically much faster (and can use significantly less memory) than building the corresponding DFA. At runtime, the NFA might have to advance multiple states (think chips in the previous analogy), so in the worst case, the number of steps it needs to take is proportional to the number of NFA states per input character.

Perl 6 uses NFAs to match most of the regex features we've discussed so far, including literals, quantifiers, conjunctions, and disjunctions, as well as character classes (which are just a convenient syntax for disjunctions). Regex features that an NFA can't handle are the sequential disjunction (`||`) and conjunction (`&&`), anchors, backreferences, code blocks, and code assertions, as well as some features that we will discuss later: regexes with explicit backtracking control, recursion, look-ahead, and look-behind.

6.2 Regex Control Flow

In cases where the regex engine cannot use an NFA, or where understanding the NFA doesn't provide much insight into how a regex will match, it is useful to understand the control flow of the general regex engine.

The first rule is that the regex engine always starts at the start of a string and prefers the leftmost match that is possible.

The second rule is the rule of greediness: the regex is evaluated from left to right and each part that can match a variable number of characters tries to match the longest substring it can.[3]

Thus, if you have the regex `/\w+ .*/` and run it against the string `abcd`, the `\w+` matches the whole string, leaving the `.*` to successfully match the empty string. A match where `\w+` matched the first character only and `.*` the rest would be equally allowed, but doesn't happen, because it violates the rule that leftmost parts of the regex match as much as they can.

[3]With lazy or frugal quantifiers, it tries to match the shortest substring it can.

If you try to match a quoted string with a simple regex `/ \" .* \" /`, it is likely that it matches more than you might have expected:

```
my $str = 'Amanda sighed. "It was madness", she said. '
    ~ '"Sheer madness"';
if $str ~~ / \" .* \" / {
   say $/.Str;
}
```

This produces the output

```
"It was madness", she said. "Sheer madness"
```

because the `.*` doesn't care that you didn't intend it to match past the boundaries of the first quote; it simply matches as much as it can, while still making the whole match succeed.

6.3 Backtracking

The rule of greediness doesn't always produce a match on the first attempt. In the previous example, the `.*` first matched all it could, including the final quote in the input string. The final quote in the regex had nothing to match, leading to a failure.

Instead of giving up, the regex engine started to *backtrack*: it went back to the previous quantifier (the * in `.*`), and made it match one less repetition. Only then could the quote in the regex find the quote in the input string, making the whole regex match successfully.

If there are multiple regex elements that could match in different ways, backtracking tries all options from the last element, and if they all fail, tries one more option from the second-to-last element. It then explores all variations of the last element in that configuration and continues attempting from previous elements in its search for a match. If you think of it as a search tree, it's a depth-first search.

Let's consider a regex with two variable parts, combined with a search:

```
'mabracadabra' ~~ / (.+a) (.*) $0 /;
```

This starts with the first group `(.+a)` matching the full string; the second group, `(.*)`, matches the empty string; the final part, `$0`, fails to match. Consequently, backtracking kicks in and asks the second group to match fewer characters, which it can't; so it goes back to the first group and asks it to match fewer characters. Now the first group matches `mabracada` and the `.*` matches the three remaining characters. Still, the final group fails to match, so `.*` gives up another character and the last group continues to fail.

This cycle repeats until the first group matches only two characters, `ma`. Even in this configuration, the rest of the regex can't match successfully. The regex engine is out of options, so it goes to the mechanism of last resort: it tries to start the match one position to the left, at the second character.

Starting from the second character, the first group tries again to match the whole remaining string, `abracadabra`. We can visualize this by adding a code block after the second part of the regex:

```
'mabracadabra' ~~ /(.+a) (.*) { say "0: '$0'; 1: '$1'" } $0 /;
```

This produces the following output:

```
0: 'mabracadabra'; 1: ''
0: 'mabracada'; 1: 'bra'
0: 'mabracada'; 1: 'br'
0: 'mabracada'; 1: 'b'
0: 'mabracada'; 1: ''
0: 'mabraca'; 1: 'dabra'
0: 'mabraca'; 1: 'dabr'
0: 'mabraca'; 1: 'dab'
0: 'mabraca'; 1: 'da'
```

```
0: 'mabraca'; 1: 'd'
0: 'mabraca'; 1: ''
0: 'mabra'; 1: 'cadabra'
0: 'mabra'; 1: 'cadabr'
0: 'mabra'; 1: 'cadab'
0: 'mabra'; 1: 'cada'
0: 'mabra'; 1: 'cad'
0: 'mabra'; 1: 'ca'
0: 'mabra'; 1: 'c'
0: 'mabra'; 1: ''
0: 'ma'; 1: 'bracadabra'
0: 'ma'; 1: 'bracadabr'
0: 'ma'; 1: 'bracadab'
0: 'ma'; 1: 'bracada'
0: 'ma'; 1: 'bracad'
0: 'ma'; 1: 'braca'
0: 'ma'; 1: 'brac'
0: 'ma'; 1: 'bra'
0: 'ma'; 1: 'br'
0: 'ma'; 1: 'b'
0: 'ma'; 1: ''
0: 'abracadabra'; 1: ''
0: 'abracada'; 1: 'bra'
0: 'abracada'; 1: 'br'
0: 'abracada'; 1: 'b'
0: 'abracada'; 1: ''
0: 'abraca'; 1: 'dabra'
0: 'abraca'; 1: 'dabr'
0: 'abraca'; 1: 'dab'
0: 'abraca'; 1: 'da'
0: 'abraca'; 1: 'd'
0: 'abraca'; 1: ''
```

```
0: 'abra'; 1: 'cadabra'
0: 'abra'; 1: 'cadabr'
0: 'abra'; 1: 'cadab'
0: 'abra'; 1: 'cada'
0: 'abra'; 1: 'cad'
```

After trying 46 different configurations, it finds a match: starting at the second character, the `(.+a)` matches `abra`, the `(.*)` matches `cad`, and the `$0` again matches `abra`.

You can clearly see that the later parts of the regex vary more rapidly than the earlier parts of a regex, judging by what they match. The advancement of the starting position of the regex is even slower, and is in fact the slowest variation there is.

The regex engine never backtracks into a zero-width assertion (look-ahead and look-behind assertions). For those, it is only relevant whether they match at all, not how they match, since a successful match never consumes any characters of the input string (it matches the empty string).

6.4 Why Would You Want to Avoid Backtracking?

Backtracking is the Secret Magic Sauce™ that makes regexes work. It makes the computer work hard on your behalf to find a match in the input string.

But there are cases where backtracking does more harm than good; thus, Perl 6 (and some other regex implementations) offer ways to disable backtracking, or reduce how much work it does.

Performance

One reason is performance: backtracking can be computationally intensive, and if you already know that a certain match (or part of a match) will fail, telling the computer to do less work speeds things up.

A simple example is this regex match:

```
"aaaaaa" ~~ /^ (a+) <[bcd]> /;
```

You can see that this match fails, because there is no `b`, `c`, or `d` in the input string. However, the regex engine isn't smart enough to know that it must search this way for optimal performance. Instead it first tries to match all `a`'s in the input, then one fewer, and so on, before declaring defeat. Again, we can track its progress through an embedded code block. `say "aaaaaa" ~~ /^(a+) { say $0 } b/` produces this output:

```
「aaaaaa」
「aaaaa」
「aaaa」
「aaa」
「aa」
「a」
Nil
```

Being smarter in some ways than the computer, we can help it by telling it to never give up any a's it matched. `say "aaaaaa" ~~ /^(a+:) { say $0 } b/` produces a much shorter output:

```
「aaaaaa」
Nil
```

The colon (`:`) behind the quantifier tells it not to backtrack for this quantifier. It works not only for all quantifiers, but also for disjunctions. The quantifier together with the backtracking control colon is called a *possessive quantifier*, because the quantifier never gives up what it once possessed.

A second way to disable backtracking is to use the `:ratchet` (or `:r`) modifier, which you can also use inside a group `[...]` or a capture `(...)`. In that case, the effect of the modifier stretches from the point of use to the closing bracket of the group.

Correctness

Backtracking can sometimes lead to unexpected match results. This happens because part of a regex matched the way you intended it to, and then a later part failed to match. The first part then starts to backtrack and match in an unexpected way.

Jan Goyvaerts shares an example on the regex guru blog,[4] paraphrased here and adapted to Perl 6 regex syntax.

When you try to match a pair of HTML tags, like the string `<a  href="...">some text</a>`, you could use a regex like this:

```
my $html-re = rx:ignorecase{
    '<' $<tag>=[ <[a..z]>+ ] # opening tag
         <-[ > ]>*           # attributes within opening tag
    '>'
        ( .* )               # content between opening and
                               closing tags
    '</' $<tag> '>'          # closing tag
};

say 'more text <a href="..:">link text</a> bla' ~~ $html-re;
```

This seems to work, but erroneously also matches mismatched tags such as `<abc></a>`. Here the regex engine first tries to match with `$<tag>` being `abc`, fails to find the match, and then backtracks. In the second iteration, it tries with `ab`, fails, and succeeds in the third iteration with `$<tag>` being `a`.

[4] http://www.regexguru.com/2008/04/unintended-backtracking-can-bite-you/

One could argue that the regex is flawed and that it should have a word boundary assertion after matching the opening HTML tag. But still, our intuition about regex matches often coincides with the greedy nature of quantifiers, not with the more nuanced matches that backtracking can produce.

When writing a regex or a parser, ask yourself about the individual elements: if this part of the regex finds a match, and then fails later, do I want it to try a different match? Imagine you're writing a parser for a programming language and write a regex that matches an identifier. Should the match ever give up a character or two? Probably not. In a case like this, it's a good idea to add a colon (`:`) to prevent it from backtracking:

```
/ $<tag>=[ <[a..z]>+: ] /
```

6.5 Frugal Quantifiers and Backtracking

As mentioned in an earlier chapter, there are *frugal* or *lazy* quantifiers that try to match as little as possible. These always backtrack, even with the `:ratchet` modifier enabled.

When a greedy quantifier tries to match as much as possible, even without backtracking, there is a search going on, by applying the quantified regex element and seeing if it matches. On the other hand, a frugal quantifier always starts with the minimal number of matches, and only backtracking can bring it to match more. So without backtracking, `/a??/` or `/a*?/` would match just the empty string, `/a+?/` just a single `a`, and so forth.

6.6 Longest Token Matching

Disjunctions or alternatives with | prefer the branch that matches the longest string. We call this *longest token matching*.

This is important when parsing, because it matches the way we intuitively recognize text ourselves. For instance, the Perl 6 expression `++$var` could be parsed as the prefix operator ++ (which increments the following variable by one), followed by the variable `$var`. Or it could be parsed as `+(+$var)`, applying the prefix + operator (which converts its argument to a number) to `$var`, and then applies the same operator again to the result. The Perl 6 grammar is based on regexes, and it knows to prefer the first variant (a single ++ operator), because it produces the longest match when trying to parse a prefix operator.[5]

To be able to match the longest alternative efficiently, Perl 6 determines which parts of a regex form a *declarative prefix*. It then constructs an NFA to match this declarative prefix. Once that's done, the rest of the general regex engine kicks in and carries on, until the next declarative section is found.

The declarative prefix is limited to elements that an NFA can model: literals, character classes, disjunctions, conjunctions, and greedy quantifiers. In the case of named regex calls, the declarative prefix from the named regex is merged into the one from the calling regex, except when it leads to recursion.

We can illustrate this by inserting a code block {} (which is not declarative) to limit the length of the alternative (Figure 6-9).

[5]This is another case where you don't want backtracking to kick in: once the parser has recognized a ++ operator, you don't want it to change its mind and interpret the two characters as separate operators.

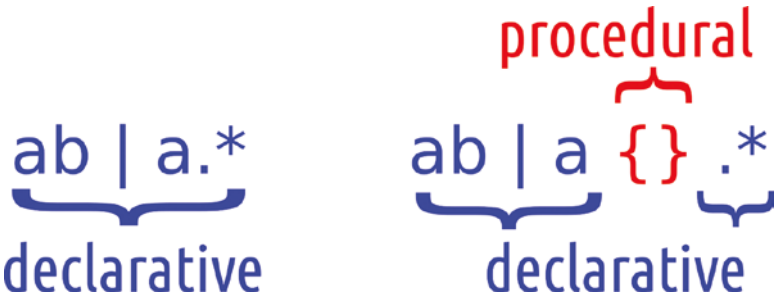

Figure 6-9. *The code block {} is considered procedural, thereby limiting the declarative prefix of the regex*

```
say "abc" ~~ /ab | a.* /;       # Output: ｢abc｣
say "abc" ~~ /ab | a {} .* /;   # Output: ｢ab｣
```

Here in the regex `ab | a.*`, the second branch matches the whole string, which is longer than what the first branch matches.

By inserting the empty block `{}`, the longest token matching is effectively limited to the regex `ab | a`. Now the first branch, `ab`, produces the longer match (because the declarative part of the second branch matches only `"a"`), and the regex engine commits to this decision, matching just `"ab"`. Only if a later regex element fails to match (and backtracking is enabled) will the regex engine reconsider this decision.

6.7 Summary

Computer science gives us tools for matching regular languages efficiently through finite automata. Even so, regular languages are only a subset of what regexes can describe. The rest is processed by using backtracking, which prefers leftmost matches over matches to the right, and longer matches over shorter matches.

CHAPTER 7

Regex Techniques

The previous chapters have introduced many regex constructs that you can use to build your regexes, and have explained how they work. But that alone doesn't tell you how to structure your regex for the best results. This chapter provides such guidance.

7.1 Know Your Data Format

This might be obvious, but is still too often overlooked. To be able to write a regex for some data format, you need to know its rules. That is, if there are rules. If not, you might need enough input data to come up with rules yourself and to test them against the data. Then, you need to know about the context of the validation.

Well-Defined Data Formats

Let's talk about the case of defined data formats first. Even if you think you know the data format pretty well, it's worth reading the specification. Often there are little-used edge cases that might be worth considering.

For instance, a string like `johndoe` (without any `@` sign) is a valid e-mail address for local delivery. If you're writing an e-mail address regex for input validation, should you allow it? The answer depends on the context. If you are developing an application that is used purely internally in a corporation, it might make sense to allow it. In a public web application, much less so.

M. Lenz, *Parsing with Perl 6 Regexes and Grammars*,
https://doi.org/10.1007/978-1-4842-3228-6_7

Staying with e-mail addresses, did you know that the local part can be quoted? Or that the domain part can be an IP address, optionally in square brackets? Yes, `"somebody"@[93.184.216.34]` and `some+body@[IPv6:2001:db8::1]` are both valid e-mail addresses.

Some data formats come with specifications for their format. For example JSON, the JavaScript Object Notation, has parsing rules on its homepage.[1] In this case, it is often a pragmatic choice to reuse the work that the authors have put into creating those rules and simply translate them into Perl 6 syntax.

Exploring Data Formats

If you don't have the luxury of a ready-made specification, you must come up with your own rules. The best way to come up with such rules mirrors the scientific method: first you come up with a hypothesis, and then you test it. You iterate until you are satisfied with the result.

Testing a hypothesis about a data format can take several forms. If you know of another program that deals with the same data format, you can modify existing files in that format, and then load them with the other program. If that is not an option, you should try to collect as much real-world data as you can in the same format, and then search through that corpus to test your hypothesis.

Let's suppose you want to write a regex that parses an INI file, the configuration format commonly used on the Microsoft Windows platform.

In its basic form, it looks like this (example taken from the Wikipedia page on INI files[2]):

```
; last modified 1 April 2001 by John Doe
[owner]
name=John Doe
organization=Acme Widgets Inc.
```

[1] http://json.org/

[2] https://en.wikipedia.org/wiki/INI_file#Example

```
[database]
; use IP address in case network name resolution
; is not working
server=192.0.2.62
port=143
file="payroll.dat"
```

The rules look pretty simple: the file consists of a list of sections. Each section starts with a section name in square brackets and contains key/value pairs separated by an equals sign (`=`). Empty lines, and lines starting with a semicolon (`;`), are ignored.

Now it's time to ask some more questions:

- Can the list of sections be empty (that is, is an empty file a valid INI file)?
- Can a section be empty (not contain any key/value pairs)?
- Are comments allowed after a key/value pair? For instance, if you write `port=443; only one not blocked by firewall`, is the comment part of the value or not?
- Where is whitespace allowed? Are `[ database ]` or `port = 443` valid?
- What's allowed in a section name? For example whitespace, or an opening bracket? What about line breaks?
- Same for keys. Is a dash (`-`) allowed in a key?

- What's the rule for values? Do they extend to the end of the line? Do they stop at a comment marker? Are they multiline if the next line isn't a comment, a key/value pair, or a section marker? Is an empty value allowed?
- Are any escape sequences (like `\t` for a tabulator) permitted? If yes, where? Just in the value, or also in the key or the section headings?
- Is quoting supported, and if yes, where?
- Are non-ASCII characters allowed in section names, keys, and values?

These are typical questions when analyzing a plain text data format. When you investigate the answers to these questions, you can turn them into test cases for your regex.

Data formats without a fixed specification often have several variations, and implementations differ in what they accept. In this case, you have to decide whether you target just one variant, or a common subset, or maybe a superset of them all.

7.2 Think About Invalid Inputs

When you write your regexes, don't just think about the strings that the regexes should match. Pay equal attention to the strings they are not supposed to match. In general, there are more strings that a regex should reject than strings that a regex should accept, which makes it both important and hard to think about them.

One thing you can do is write negative tests; that is, a test with input that is not supposed to match. Another is to stop and think harder when you see a term like `.*` or `.+`. These are so broad that they are wrong most

of the time. Most data formats don't include a clause "and then, whatever." For instance, if you parse comments that run to the end of the line, `/'#' .*/` is wrong, because it can match past a line break; `/'#' \N* \n? /` would be the better regex for that. And if you parse C-like comments, `/* ... */`, the regex `/ '/*' .* '*/'` also matches the whole string `/* abc */ de */`, because `.*` greedily matches the closing comment marker `*/` in the middle of the string.

7.3 Use Anchors

If you write a regex intended to match a string in its entirety, remember the anchors `^` and `$` to ensure that it does match the whole string. If not, think about the boundaries of the regex match. If you try to match a word not followed by a dot (`.`), the regex `/\w+ <!before '.'> /` can match a partial word:

```
say "supercalifragilisticexpialidocious."
        ~~ / \w+ <!before '.'> /;
    # Output: 「supercalifragilisticexpialidociou」
```

The `\w+` here actually matches one character less than the actual word, so that `<!before '.'>` can match successfully.

If such behavior is not what you have in mind, you can add an assertion such as a word boundary, `/ \w+ » <!before '.'> /`. Since the `\w+` implies that part of a word has already been matched, you could also write `/ \w+ <!before \w> <!before '.'> /`.

The contents of an assertion is a regex, so here we have to quote the dot (or escape it as `<!before \.>`), otherwise the dot would unfold its special meaning and match any character.

7.4 Matching Quoted Strings

Many data formats include quoted strings. Their defining feature is that between two (or possibly more) delimiting characters—the quotes—more (typically nearly all) characters are allowed.

We've seen quoted strings in Perl 6 regexes themselves, where nonword characters may appear that are otherwise reserved in regexes. Common data exchange formats such as CSV, JSON, and YAML also contain quotes. CSV—the comma separated values format—has a separator character (by default the comma (`,`); hence the name) that separates columns in a table-like structure. If a column value itself contains the comma, you have to be able to distinguish that comma from the separator character, and that is typically done by quoting.

A CSV file containing the columns `a`, then `b,c`, and finally `d` (i.e., three columns) could be written as

```
a,"b,c",d
```

where `"b,c"` is a quoted string.

As mentioned several times already, `/ \" .* \" /` is not a valid way to parse quoted strings, since the `.*` can match past closing quotes. If the input were

```
a,"b,c",d,"e,f"
```

then the regex for the quoted string would actually match the two quoted strings in the input as well as the columns in between.

A less naïve approach is to write `/ \" .*? \" /`, which limits the quoted string to the shortest possible match. This works, but only if nothing forces the regex to backtrack and match in a different way:

```
say '"a,b","' ~~ / ^ \" .*? \" $ /;       # Output: 「"a,b","」
```

Here the input is an *unbalanced* quoted string, with a third quote character in the middle. The first attempt only matches the part `"a,b"` of the string, the `$` anchor fails to match, and therefore backtracking kicks in and the next attempt matches the whole string.

This is typically not the desired behavior. A more robust approach is to forego the dot in the regex, and think harder about what's allowed inside the quoted string: everything except a quote character. That's easy to realize with a negated character class:

```
say '"a,b","' ~~ / \" <-["]>* \" /;        # Output: 「"a,b"」
say '"a,b","' ~~ / ^ \" <-["]>* \" $ /;    # Output: Nil
```

This now matches just a balanced, quoted string.

Quoted Strings with Escaping Sequences

Our challenges don't end here. In the preceding CSV example, we can now deal with columns containing the separation character, but since the quote character gained a special meaning, you can't easily have a column containing the quote character.

The most common solution to this problem is to introduce an escape character, often the backlash (`\`). If you put a backlash before a quote character or a backslash, that second character loses its special meaning. Thus, if you want to include the string `she said "hey, ho"` in a CSV column, you have to write it as `"she said \"hey, ho\""`.

Working with the backslash as an escape character becomes more complicated because the backslash also has a special meaning in Perl 6 (and most other programming languages, for that matter). Consequently, in ordinary string literals, you have to double the backslash to produce a single one:

```
say "a\\b";           # Output: a\b
```

You can switch off this behavior in Perl 6 by adding a capital `Q` in front of the string, which disables all escape sequences:

```
say Q"a\b";          # Output: a\b
say Q"a\\b";         # Output: a\\b
```

Regexes have no such mode, so you have to double backslashes in regexes to match a literal backslash.

Coming back to quotes with escape characters, there are now two cases to consider inside the quoted strings: regular characters without any meaning, and escape sequences. Regular characters are all characters except the quote or the backslash, which the character class `<-[ " \\ ]>` describes. An escape sequence is a backslash, followed by a quote or a backslash. In regex terms, that's `\\ <[ " \\ ]>`. Many data formats allow other characters behind a backslash too, in which case the sequence simplifies to `\\ .` (remember the dot matches any character).

Combining these two cases leads to this regex:

```
my regex quoted {
    \" # opening  quote
    [
          <-[ " \\ ]>  # regular character
        | \\ .         # escape sequence
    ]*
    \" # closing quote
}
```

Don't worry if it takes you a minute or more to read the regex. It's a bit to unpack and understand, but it'll be the most involved regex in this book.

The `my regex quoted { ... }` construct declares a named regex as a lexical variable. Later, we'll learn about grammars where we can omit the `my`.

7.5 Testing Regexes

Regexes are code. They are declarative rather than procedural, which means when you write a regex, you specify what text it matches, and typically not how. But still, they are code, and you should write tests for code to gain confidence they work as intended, and to be able to change them without fear of breaking other code which uses them.

Perl 6 ships with a `Test` module that makes it easy to write such tests. Here is a small test suite for the regex from the previous section:

```
my regex quoted {
    \" # opening quote
    [
             <-[ " \\ ]>  # regular character
          | \\ .          # escape sequence
    ]*
    \" # closing quote
}

my @should-match =
    Q<"abc">,
    Q<"abc\\">,
    Q<"ac\\def\"ef">,
    ;

my @should-not-match =
    Q<abc>,
    Q<"abc"def">,
    Q<"ab\\"cdef">,
    ;

use Test;
plan 6;
```

```
for @should-match -> $s {
     ok $s ~~ / ^ <quoted> $ /,
         "Successful match of string $s";
}
for @should-not-match -> $s {
      nok $s ~~ / ^ <quoted> $ /,
          "Successful rejection of string $s";
}
```

Let's explore this in more detail. The start is the familiar declaration of the regex. The code then declares and initializes two array variables, `@should-match` and `@should-not-match`.

The first contains strings that we expect the regex to match; the second has examples of strings that the regex is not supposed to match.

Then, the testing begins. `use Test;` imports the testing module,[3] which provides some functions used further down in the script. `plan` is one of them and `plan 6;` tells the testing module that six test cases are upcoming.

Two `for` loops follow, which check the example strings. `ok` is a testing function that, when called with a true value as its first argument, makes a test succeed; a false value fails it. The first argument here is the result of an anchored regex match, `$s ~~ / ^ <quoted> $ /`. The second argument is a label for the test. In the second loop, the testing function `nok` is used instead, which has the inverse logic from `ok`: it expects a false value (like a failed regex match) as the first argument.

When you run the script, it produces output like this:

```
$ perl6 quotes-with-escapes.p6
1..6
ok 1 - Successful match of string "abc"
ok 2 - Successful match of string "abc\\"
```

[3] https://docs.perl6.org/language/testing

```
ok 3 - Successful match of string "ac\\def\"ef"
ok 4 - Successful rejection of string abc
ok 5 - Successful rejection of string "abc"def"
ok 6 - Successful rejection of string "ab\\"cdef"
```

If you change one of first three example strings to make the regex match fail, you get a failing test output like this:

```
$ perl6 quotes-with-escapes.p6
1..6
ok 1 - Successful match of string "abc"
not ok 2 - Successful match of string "abc\\"a
# Failed test 'Successful match of string "abc\\"a'
# at quotes-with-escapes.p6 line 26
ok 3 - Successful match of string "ac\\def\"ef"
ok 4 - Successful rejection of string abc
ok 5 - Successful rejection of string "abc"def"
ok 6 - Successful rejection of string "ab\\"cdef"
# Looks like you failed 1 test of 6
```

With many tests in a file, or even many test files, it can be helpful to have a summary of all test outputs. The `prove` program commonly bundled with Perl 5 understands the output from the tests (which is in the Test Anything Protocol format[4]) and prints out a short summary:

```
$ prove -e perl6 quotes-with-escapes.p6
quotes-with-escapes.p6 .. ok
All tests successful.
Files=1, Tests=6, 1 wallclock secs ( 0.02 usr  0.00 sys +  0.24
cusr 0.01 csys = 0.27 CPU)
Result: PASS
```

[4] https://testanything.org/

If tests fail, it focuses on the failed tests to make it easier to fix them:

```
$ prove -e perl6 quotes-with-escapes.p6
quotes-with-escapes.p6 .. 1/6
# Failed test 'Successful match of string "abc\\"a'
# at quotes-with-escapes.p6 line 26
# Looks like you failed 1 test of 6
quotes-with-escapes.p6 .. Dubious, test returned 1 (wstat 256,
0x100) Failed 1/6 subtests

Test Summary Report
-------------------
quotes-with-escapes.p6 (Wstat: 256 Tests: 6 Failed: 1)
  Failed test: 2
  Non-zero exit status: 1
Files=1, Tests=6, 0 wallclock secs ( 0.01 usr 0.01 sys
+ 0.23 cusr 0.01 c\sys = 0.26 CPU)
Result: FAIL
```

`prove` even adds color to the terminal output, with green for "All tests successful.", and red for test failures, making the status obvious.

After initial development of your regexes, it is likely that you want to use them for something other than running tests. In this case, you can extract the tests into a subroutine and run them only when you call your script with a special command-line argument, such as `--test`:

```
my regex quoted {
    \"  # opening quote
    [
         <-[ " \\ ]>  # regular character
        | \\ .        # escape sequence
    ]*
    \" # closing quote
}
```

```
multi sub MAIN(Bool :$test!) {
    my @should-match =
        Q<"abc">,
        Q<"abc\\">,
        Q<"ac\\def\"ef">,
        ;

    my @should-not-match =
        Q<abc>,
        Q<"abc"def">,
        Q<"ab\\"cdef">,
        ;

    use Test;
    plan 6;

    for @should-match -> $s {
        ok  $s ~~ / ^ <quoted> $ /,
            "Successful match of string $s";
    }
    for @should-not-match -> $s {
        nok $s ~~ / ^ <quoted> $ /,
            "Successful rejection of string $s";
    }
}

multi sub MAIN($input) {
    if $input ~~ / ^ <quoted> $ / {
        say "$input is a quoted string";
    }
    else {
        say "invalid input: $input";
        exit 1;
    }
}
```

In this example, `multi sub MAIN` introduces a subroutine called `MAIN`. Perl 6 automatically calls a subroutine `MAIN` for you and translates command-line arguments into arguments for this function. The `multi` implies that there can be more than one subroutine with this name. Perl 6 calls the candidate that has the best fit of arguments. `:$test!` is a *named argument*, by virtue of the leading colon (`:`), and the trailing exclamation mark (`!`) makes it required. You can only call this candidate by passing in a named argument `test`.

This is what it looks like on the command line:

```
$ perl6 quote-checker --test
1..6
ok 1 - Successful match of string "abc"
ok 2 - Successful match of string "abc\\"
ok 3 - Successful match of string "ac\\def\"ef"
ok 4 - Successful rejection of string abc
ok 5 - Successful rejection of string "abc"def"
ok 6 - Successful rejection of string "ab\\"cdef"
```

Or if you want to use the `prove` test harness:

```
$ prove -e "" "perl6 quote-checker.p6 --test"
perl6 examples/quote-checker.p6 --test .. ok
All tests successful.
Files=1, Tests=6,  0 wallclock secs ( 0.01 usr  0.00 sys
+  0.26 cusr  0.02 c\sys = 0.29 CPU)
Result: PASS
```

The second candidate, declared with `multi sub MAIN($input)`, is the "normal" execution path of the program. It takes a positional argument, which in terms of the command line is a string that is not an option (so doesn't start with a minus (`-`) character). It checks whether the argument is a quoted string and prints a message accordingly:

```
$ perl6 quote-checker '"yes"'
"yes" is a quoted string
```

```
$ perl6 quote-checker no
invalid input: no
```

The shell uses up one level of quoting, which is why in the first invocation I used single quotes around the double-quoted string. This works in the standard POSIX shell and in bash; other shells might require you to do different quoting to get this example running.

7.6 Summary

When writing a regex for a data format, we often discover that we know less about that data format than we originally thought. We can try to find a formal specification that answers our questions, or we can use an experimental approach, combined with a catalog of questions we have about the data format.

We also discussed the use of assertions at the boundaries of what regexes should match, as well as clever use of negated character classes to reliably parse quoted strings, even in the presence of escape characters.

Testing gives us confidence that the regexes we write match no more and no less than what we expected, and we explored a simple way to write automated tests.

CHAPTER 8

Reusing and Composing Regexes

Perl 6 offers great tools for composing regexes, thus making them reusable. This inspires programmers to carefully organize and test their regexes, just like regular code.

8.1 Named Regexes

Giving things a name, and being able to refer to those things by name, is the first step toward composability and reusing regexes.

As we have seen before in some examples, you can give regexes names, just like you can with variables, subroutines, and so on. This raises regexes to the same level as the other constructs in a language that enable abstraction:

```
my regex byte {
    \d ** 1..3
    <?{ $/.Int <= 255 }>
}

my $str = '127.0.0.1';
say $str ~~ / ^ <byte> ** 4 % '.' $ /;
```


M. Lenz, *Parsing with Perl 6 Regexes and Grammars*,
https://doi.org/10.1007/978-1-4842-3228-6_8

Here `my regex byte { ... }` declares a regex called `byte`. The word before the `regex` keyword determines the scope: `my` is for lexical scoping. Leaving it out declares a regex that is attached to a grammar. More on that later.

Perl 6 regexes are code objects, like subroutines or methods. Just like routines, you can refer to regexes by adding an ampersand (`&`) before their name[1]:

```
say &byte.^name;              # Output: Regex
say &byte ~~ Code;            # Output: True
```

Unlike a normal subroutine, you can't directly call a regex; it needs a *cursor object* to track its progress. This is what you get when you try anyway:

```
$ perl6 -e 'my regex a { . }; a("x")'
No such method '!cursor_start' for invocant of type 'Str'
```

Instead, you can use smart-matching to invoke a regex, just as you do with an anonymous regex:

```
my regex a { . };
say "x" ~~ &a;                # Output: ｢x｣
```

Or of course one could invoke it from inside an anonymous regex:

```
my regex a { . };
"x" ~~ / <a> /;
```

Calling a named regex through the angle bracket syntax `<a>` creates a named capture with the name of the regex being called, here `a`. If you want to avoid the capture, you can write it as `<&a>` instead.

[1]Perl 6 has a built-in type called `byte`, so if you leave out the ampersand, you'll accidentally reference this type instead.

If you want the capture to be of a different name, you can write `<b=&a>`. This produces a named capture `b`, but invokes the regex named `a`. If you instead write `<b=a>`, the same capture is available under the names `a` and `b`. There are even more ways to invoke named regexes (Table 8-1).

Table 8-1. *Syntactic Forms of Invoking Named Regexes*

Example	Description	Captures
`<a>`	Named regex call	`a`
`<&a>`	Named regex call without capture	(none)
`<b=a>`	Named regex call with alias	`a`, `b`
`<b=&a>`	Named regex call, renamed	`b`
`<?a>`	Named regex call as look-ahead	(none)
`<!a>`	Named regex call as negated look-ahead	(none)

If the first character after the opening `<` is a word character, a named capture is produced with that name. Any form of punctuation—be it `?` for a look-ahead or `&` for a plain call—suppresses the capture.

Lexical Analysis and Backtracking Control

In traditional parsing literature, analyzing a piece of text typically happens in two stages: *lexical analysis*, also called *tokenization*, and then the actual *parsing*.

Lexical analysis breaks up a piece of text into *tokens*, small pieces of text with a label that classifies them. For instance, if we were to write a small calculator, the input `2 * (3 + 5)` could result in a stream of tokens like this:

```
2    number
*    product
(    opening parenthesis
3    number
+    addition
5    number
)    closing parenthesis
```

These tokens are initially in a linear list; the parsing step then transforms them into a tree that we could use for evaluating the expression:

```
*
/ \
2  +
   / \
  3   5
```

This two-step method works well for simple things like basic mathematical expressions, but tends to fail when parsing something that has context-dependent sublanguages. If you were to write a parser for Perl 6, the lexical analysis for the main language is very different from that of the insides of quoted strings, which is yet very different from the lexical analysis inside a regex.

To accommodate such situations, Perl 6 offers the keyword `token`, which introduces a regex with backtracking turned off. The idea is that lexical analysis should be simple enough that, once you decided how to cut and label a token, you don't want to give up that decision based on input that comes later. Hence, no backtracking.

By doing lexical analysis inside a normal regex match, we have all the context we need to decide what tokenization to apply.

The first example in this chapter involved a `regex byte`, which was simple enough that we typically don't want backtracking. We could therefore rewrite it to use a `token` instead:

```
my token byte {
    \d ** 1..3
    <?{ $/.Int <= 255 }>
}

my $str = '127.0.0.1';
say $str ~~ / ^ <byte> ** 4 % '.' $ /;
```

The `my token byte { ... }` does the same as explicitly disabling backtracking in the regex with `my regex byte { :r ...}`. However, it is used so often that it deserves its own syntax.

Note that when we sometimes use the word "regex" to refer to regexes declared with the `token` keyword.

8.2 Whitespace

Returning to the example of tokenizing a simple mathematical expression, we glossed over an important detail: the input contained whitespace between the tokens, and the token stream did not. Lexical analysis often discards insignificant whitespace.

Not all whitespace is insignificant. You don't want to write `"Hello, World"` and have it come out as `Hello, World`. Many languages make a distinction where space between tokens is insignificant between tokens, but space inside a token (like a quoted string) is significant.

However, it's more subtle than that: in languages like SQL, Perl, Python, JavaScript, etcetera, you can leave out whitespace when a word-like token follows a nonword token, or the other way around

(e.g. `a+b` is the same as `a + b`). Nevertheless, joining two word-like tokens without any whitespace is forbidden. Conversely, if there are blanks between two tokens, the amount does not matter.[2]

As an example, these two SQL statements produce identical parse trees:

```
SELECT username,first_login FROM account;
SELECT
       username,
       first_login
FROM   account;
```

In contrast, the following would be a syntax error, because it joins word-like tokens without any whitespace:

```
SELECTusername,first_loginFROMaccount;
```

Perl 6 defines a regex called `ws`, short for whitespace, which parses whitespace to the rules laid out before: arbitrary amounts of whitespace (blanks, tabs, newlines, ...), but at least one, unless it's at a word boundary. Or to put it in code, `regex ws { <!ww> \s* }` (where `<!ww>` matches anywhere except within a word).[3]

There is also a shortcut that helps you to avoid having to sprinkle explicit `<ws>` or `<.ws>` calls everywhere in your code.[4] If you declare your regexes with the keyword `rule` instead of `regex` or `token`, Perl 6 inserts implicit `<.ws>` calls for you wherever you use whitespace in your regex. This has the same effect as using the `:sigspace` or `:s` modifier in your regex.

---

[2]In Python, it's even more complicated, since whitespace at the start of a line is significant, except when it's inside an expression.

[3]The assertion `<!ww>` matches every exception within a word.

[4]Remember that the leading dot in `<.ws>` causes the regex not to capture. Since the `ws` routine is all about whitespace that we don't particularly care about, not capturing its match makes sense.

Since the definition of ws uses the term \s*, a single blank in a regex can match any amount of whitespace in the string, including tabs, spaces, vertical tabs, and so on.

Considering our mathematical expressions again, we could thus write the following regexes to match the simplest case of a sum of two numbers:

```
my token number { \d+ }
my rule  sum    { <number> '+' <number> }

say '1+2'   ~~ / ^ <sum> $ /;
say '1 + 2' ~~ / ^ <sum> $ /;
```

The second line is equivalent to writing my token sum { <number> <.ws> '+' <.ws> <number> <.ws> }, but is much more readable. The last two lines both match successfully, the only difference being the whitespace they matched:

```
｢1+2｣
 sum => ｢1+2｣
  number => ｢1｣
  number => ｢2｣
｢1 + 2｣
 sum => ｢1 + 2｣
  number => ｢1｣
  number => ｢2｣
```

The insertion of implicit <.ws> calls is not quite for every occurrence of whitespace in the regex. In particular, whitespace at the start of the regex, after a modifier like :r or :s, after an opening bracket or parenthesis, and after a &, &&, |, and || is not replaced with an implicit <.ws> call. The general idea is that rules parse whitespace inside and at the end of the match, but not at the start of the match.

If you use rules, you must be aware that whitespace is significant, and that even whitespace between an atom and its quantifier can make a difference:

```
say "a a" ~~ rule { a+ }
say "a a" ~~ rule { a + }
```

The first line prints ｢a ｣ while the second one produces ｢a a｣, including the final a. The blank between the a and the + is interpreted as a `<.ws>` call, so the second rule is equivalent to `token { [a <.ws>]+ }`.

Finally, you need to be aware that just like any part of a regex, an implicit call to `ws` can also make your match fail:

```
say 'ab' ~~ rule { a b }     # Output: Nil
```

Here the implicit `<.ws>` between a and b doesn't match, so the rule as a whole does not match either.

8.3 Grammars

Named regexes offer a first level of abstraction, but we often want more than one level. Most high-level programming languages have modules, namespaces, classes, or even all three to manage reuse of functions and methods.

Perl 6 offers these higher-level abstractions too, and makes them available to regexes. A *grammar* is a class that offers some tools for invoking regexes. Regexes are then methods in this grammar:

```
grammar IPv4Address {
    token byte {
        \d ** 1..3
        <?{ $/.Int <= 255 }>
    }
```

```
    token TOP {
        <byte> ** 4 % '.'
    }
}

my $str = '127.0.0.1';
if IPv4Address.parse($str) {
    say join ', ', $<byte>.list;
        # Output: 127, 0, 0, 1
}
```

This first grammar is declared with the `grammar` keyword, followed by the name of the grammar, `IPv4Address`. A grammar is a class that automatically adds the type `Grammar`[5] as a parent class, and provides several methods such as `parse` and `subparse`.

The grammar's body is delimited by curly braces. Inside the body, we declare regexes with the `regex` or `token` keyword. Note that we leave out the `my` in front of the declaration, because now regexes are scoped as methods belonging to the grammar.

The `parse` method invokes the regex called `TOP`, and implicitly anchors it to the start and the end of the string. By contrast, the subparse method anchors only to the start of the string. Both `parse` and subparse take an optional, named argument `rule` that you can use to invoke a regex other than `TOP`:

```
say IPv4Address.subparse($str, :rule<byte>);       # Output: 「127」
say IPv4Address.parse($str, :rule<byte>);          # Output: Nil
```

Here `IPv4Address.parse($str, :rule<byte>)` fails, because the regex `byte` can't match the whole input string, and `parse` implicitly anchors the end of the regex to the end of the string. In contrast, `subparse` matches the first number of the input string.

[5] https://docs.perl6.org/type/Grammar

Inside a grammar, a call to another regex of the same grammar works just as it does outside; with angle brackets: `<byte>`. But if you want to suppress the capture, you have to use a dot instead of the ampersand: `<.byte>`. The same applies to renaming, so `<octect=.byte>` calls the regex `byte`, but produces a capture named `octect`. The use of the dot (`.`) is analogous to the method call syntax, which also uses the dot.

8.4 Code Reuse with Grammars

Perl 6 offers two ways to reuse object-oriented code: inheritance and role composition. Since grammars are really classes, both of these mechanisms also apply to grammars.

Inheritance

Inheritance is a form of specialization. If you have a situation where you have a generic grammar, and then want to create a variant with some tweaks, the tweaked variant can inherit from the more general one.

Consider, for example, a grammar for SQL, the Structured Query Language.[6] There are standards for this grammar, but some implementations are dialects. One such instance is MySQL which uses backticks instead of double quotes to quote table and column names. A grammar for the MySQL dialect would then inherit from the standard SQL grammar and override the regexes for parsing table and column names.

[6] https://en.wikipedia.org/wiki/SQL

To illustrate this idea, let's focus just on parsing SQL column names. The SQL standard dictates that they are either an identifier or a double-quoted string:

```
grammar StandardSQL {
    regex TOP {
        'SELECT' \s+ <name>
    }
    regex name {
         <identifier>
        | <quoted_name>
    }
    regex quoted_name {
        \" <-["]>+ \"
    }
    regex identifier {
        « <:alpha> \w* »
    }
}

say StandardSQL.parse('SELECT salary');
say StandardSQL.parse('SELECT "monthly salary"');
```

Both of these parses succeed, and they produce this output:

```
「SELECT salary」
 name => 「salary」
  identifier => 「salary」
「SELECT "monthly salary"」
 name => 「"monthly salary"」
  quoted_name => 「"monthly salary"」
```

As mentioned before, MySQL uses backticks instead of double quotes to quote identifiers. Rather than duplicating the whole grammar, we can write a grammar `MysqlSQL` that inherits from `StandardSQL` and overwrites just the regex `quoted_name`:

```
grammar MysqlSQL is StandardSQL {
    regex quoted_name {
        \` <-[`]>+ \`
    }
}
say MysqlSQL.parse('SELECT `monthly salary`');
```

Here the text `"is StandardSQL"` in the grammar declaration defines the inheritance from the `StandardSQL` grammar. Inheritance means that if a regex or method is called in the child class, and it is not defined in the child class, the method of the same name from the parent class is called instead.

Role Composition

A role—also called a trait in other programming languages—is a piece of functionality (such as methods and regexes) that can be copied into a class or a grammar. This copying is called *composition*.

Role composition is a good fit for assembling a grammar from smaller, independent parts. For instance, parsing numbers and quoted strings is pretty universal, and both might be necessary when parsing SQL, JSON, and several types of configuration files. Parsing numbers is also applicable in parsing mathematical expressions:

```
role ParseInteger {
    token unsigned { <[0..9]>+ }
    token signed   { ['+' | '-']? <unsigned> }
}
```

```
role ParseFloat does ParseInteger {
    token escale { <[eE]> <unsigned> }
    token float {
        $<sign>=<[+-]>?
        [
          $<coeff> = [ <[0..9]>* '.' <unsigned> ] <escale>?
        | $<coeff> = [               <unsigned> ] <escale>
        ]
    }
}

grammar Sum does ParseFloat {
    token number { <signed> | <float> }
    rule TOP { <number> '+' <number> }
}
grammar JSON does ParseFloat {
    token value
           <signed>
         | <float>
           # more options go here
    }
    # rest of the grammar here
}

say Sum.parse('2 + -4');
```

This example shows two roles that contain regexes: `role ParseInteger` has two tokens, one for parsing an integer without any sign (token `unsigned`), and one for parsing an integer that potentially has a `+` or `-` sign. The second role, `ParseFloat`, makes use of the first one by declaring `does ParseInteger`. It then declares a token `float` to parse a floating-point number.

Unlike inheritance, role composition detects name conflicts at compile time, and forces you to resolve these conflicts. This makes it safe to compose multiple roles into the same class, or multiple roles into a single role.

8.5 Proto Regexes

Suppose you want to write a grammar for JSON[7] (also known as JavaScript Object Notation). The json.org homepage helpfully tells you that when parsing a value, you expect a string, number, object, array, or the literal strings `true`, `false`, or `null`. No problem, you can write that:

```
grammar JSON {
    token value {
        | <string>
        | <number>
        | <object>
        | <array>
        | 'true'
        | 'false'
        | 'null'
    }
    # more tokens and rules go here
}
```

This works, but it's not great for extensibility. Suppose you want to parse an extended JSON dialect, such as one that adds a date type in the form of an unquoted string like `2015-12-24`. You could do that by subclassing the `JSON` grammar hinted at in the previous example, write a token `date` to parse the date, and then you have to override `token value`, listing all the alternatives from the parent grammar and then your own.

[7] http://json.org/

This duplication is not only annoying and error-prone, it also makes multiple amendments to the same grammar impossible. If somebody else wants to write another extension that adds a different kind of new value (let's say a reference type), their extension won't be aware of yours, so the user can't easily mix and match them.

Perl 6 offers a solution to this problem: *proto regexes*. A proto regex is a collection of regexes that all form a big alternative, as if each of them were delimited by a `|`. Here is our part of the JSON grammar written as a proto regex:

```
grammar JSON {
    proto token value {*};
    token value:sym<string> { <string> }
    token value:sym<number> { <number> }
    token value:sym<object> { <object> }
    token value:sym<array>  { <array>  }
    token value:sym<true>   { <sym>    }
    token value:sym<false>  { <sym>    }
    token value:sym<null>   { <sym>    }
    # more tokens and rules go here
}
```

The line `proto token value {*}` introduces a proto regex (or token) with the name `value`. This instructs Perl 6 that there will be more regexes of the name `value`, with some small additions. These small additions are of the form `:sym<SOMETHING>` and help you and the compiler to keep these regexes apart. The body of such a regex can be anything you know from regexes, so instead of calling a token `number` from inside `token value:sym<number>`, we could do the implementation right there:

```
token value:sym<number> {
    '-'?
    [ 0 | <[1..9]> <[0..9]>* ]
```

```
    [ \. <[0..9]>+ ]?
    [ <[eE]> [\+|\-]? <[0..9]>+ ]?
}
```

Inside each individual candidate, the name that goes inside the `:sym<...>` is available as the special symbol `<sym>`. So instead of writing

```
token value:sym<null> { 'null' }
```

we can write

```
token value:sym<null> { <sym> }
```

to avoid repetition of the name.

Let's reduce the example slightly and turn it into runnable code:

```
grammar JSONValue {
    proto token value {*};
    token value:sym<true>  { <sym> }
    token value:sym<false> { <sym> }
    token value:sym<null>  { <sym> }
    token TOP { <value> }
}
```

This is a very minimalistic grammar that parses only the JSON values `true`, `false`, and `null`. Our extension to make it parse a date in ISO 8601 format could be a simple role that adds one candidate token:

```
role DateValue {
    token value:sym<date> {
        <[0..9]>**4 '-' <[0..9]>**2 '-' <[0..9]>**2
    }
}
```

We do not need to declare a `proto token value` in the role, because it will be mixed into a grammar that contains this declaration.

To use the DateValue extension, we can generate a grammar based on JSONValue and the role:

```
grammar JSONValueWithDate is JSONValue does DateValue { };

for <true null 2015-12-24 42> -> $str {
    say $str, ': ', ?JSONValueWithDate.parse($str);
}
```

This uses both inheritance from JSONValue and role composition with DateValue to generate a new grammar, which can now parse true, false, null, and date values. It produces this output:

```
true: True
null: True
2015-12-24: True
42: False
```

The ? operator outside a regex forces Boolean context, which is why we see True and False in the output instead of Match objects and Nil.

We can avoid explicitly creating the grammar JSONValueWithDate and simply compose an anonymous grammar on the fly by using the but operator:

```
for <true null 2015-12-24 42> -> $str {
    say $str, ': ', ?(JSONValue but DateValue).parse($str);
}
```

The but operator creates a copy of the type or object on the left-hand side, applies the role to it, and returns the result.

Both approaches can be used to add more than one extension to the grammar:

```
role IntegerValue {
    token value:sym<integer> { <[0..9]>+ }
}
```

```
my $grammar = (JSONValue but DateValue) but IntegerValue;

for <true null 2015-12-24 42> -> $str {
    say $str, ': ', ?$grammar.parse($str);
}
```

Now the grammar composed from `JSONValue`, `DateValue`, and `IntegerValue` can parse all four example input strings. The extension roles don't have to take the others into account; they are freely combinable.

The examples for parsing subsets of JSON have been taken from the JSON::Tiny[8] module, which parses JSON using a manageable Perl 6 grammar.[9] We're getting closer to being able to create real-world, extensible grammars with many practical applications.

8.6 Summary

Perl 6 offers named regexes for easier reuse. To simplify lexical analysis, the `token` and `rule` keywords disable backtracking, with the `"rule"` keyword also implicitly parsing whitespace.

Since regexes can behave as methods, you can use powerful techniques from object-oriented programming for managing and reusing regexes. Grammars group regexes, and inheritance and role composition make them accessible for reuse.

Finally, proto regexes ensure that extensions can happen in a natural way, without interfering with other potential extensions to the same place in a grammar.

[8] https://github.com/moritz/json/

[9] https://github.com/moritz/json/blob/master/lib/JSON/Tiny/Grammar.pm

CHAPTER 9

Parsing with Grammars

Grammars are the proverbial Swiss-army chain saw[1] for parsing.

In this chapter, we will explore them in more detail. Most importantly, we will discuss how to harness their power.

9.1 Understanding Grammars

Grammars implement a top-down approach to parsing. The entry point, usually the regex `TOP`, knows about the coarse-grained structure and calls further regexes that descend into the gory details. Recursion can be involved too. For example, if you parse a mathematical expression, a term can be an arbitrary expression inside a pair of parentheses.

This is a top-down structure, or more precisely a recursive descent parser.[2] If no backtracking is involved, we call it a *predictive* parser, because at each position in the string, we know exactly what we're looking for—we can predict what the next token is going to be (even if we can only predict that it might be one of a set of alternatives).

[1] Like a Swiss-army knife, but with much more power.

[2] `https://en.wikipedia.org/wiki/Recursive_descent_parser`

M. Lenz, *Parsing with Perl 6 Regexes and Grammars*,
https://doi.org/10.1007/978-1-4842-3228-6_9

The resulting match tree corresponds in structure exactly to the call structure of regexes in the grammar. Let's consider parsing a mathematical expression that only includes the operators *, +, and parentheses for grouping:

```
grammar MathExpression {
    token TOP    { <sum> }
    rule sum     { <product>+ %  '+' }
    rule product { <term>+ % '*' }
    rule term    { <number> | <group> }
    rule group   { '(' <sum> ')' }
    token number { \d+ }
}

say MathExpression.parse('2 + 4 * 5 * (1 + 3)');
```

From the grammar itself, you can already see the potential for recursion: `sum` calls `product`, which calls `term`, which calls `group`, which calls `sum` again. This allows parsing of nested expressions of arbitrary depth.

Running the previous example produces the following match object:

```
「2 + 4 * 5 * (1 + 3)」
 sum => 「2 + 4 * 5 * (1 + 3)」
  product => 「2 」
   term => 「2 」
    number => 「2」
  product => 「4 * 5 * (1 + 3)」
   term => 「4 」
    number => 「4」
   term => 「5 」
    number => 「5」
   term => 「(1 + 3)」
    group => 「(1 + 3)」
```

```
sum => ⌜1 + 3⌟
 product => ⌜1 ⌟
  term => ⌜1 ⌟
   number => ⌜1⌟
 product => ⌜3⌟
  term => ⌜3⌟
   number => ⌜3⌟
```

If you want to know how a particular number was parsed, you can follow the path backward by looking for lines above the current line that are indented less; for instance, the number `1` was parsed by token `number`, called from `term`, called from `product`, and so on.

We can verify this by raising an exception from token `number`:

```
token number {
    (\d+)
    { die "how did I get here?" if $0 eq '1' }
}
```

This indeed shows the call chain in the backtrace, with the most immediate context at the top:

```
how did I get here?
  in regex number at bt.p6 line 9
  in regex term at bt.p6 line 5
  in regex product at bt.p6 line 4
  in regex sum at bt.p6 line 3
  in regex group at bt.p6 line 6
  in regex term at bt.p6 line 5
  in regex product at bt.p6 line 4
  in regex sum at bt.p6 line 3
  in regex TOP at bt.p6 line 2
  in block <unit> at bt.p6 line 13
```

This grammar only uses tokens and rules, so there is no backtracking involved, and the grammar is a predictive parser. This is fairly typical. Many grammars work fine without backtracking, or with backtracking in just a few places.

Recursive Descent Parsing and Precedence

The `MathExpression` grammar has two rules which are structurally identical:

```
rule sum { <product>+ %  '+' }
rule product { <term>+ % '*' }
```

Instead, we could have written

```
rule  expression { <operator>+ % <term> }
token operator   { '*' | '+' }
```

or even used the `proto token` construct discussed in the previous chapter to parse different operators. The reason I chose the first, more repetitive, approach is that it makes the match structure correspond to the precedence of the operators * and +.

When evaluating the mathematical expression `1 + 2 * 5`, mathematicians and most programming languages evaluate the `2 * 5` first, because the * operator has tighter *precedence* than +. The result is then substituted back into the expression, leading to `1 + 10`, and finally `11` as the result.

When parsing such expressions with the first version of the grammar, the structure of the parse tree expresses this grouping: it has—as the top level—a single sum, with the operands being `1` and `2 * 5`.

This comes at a cost: we need a separate rule and name for each precedence level, and the nesting of the resulting match object has at least one level per precedence level. Furthermore, adding more precedence levels later on is not trivial, and very hard to do in a generic way. If you are

not willing to accept these costs, you can instead use the flat model with a single token for parsing all operators. If you then need the structure in a way that reflects precedence, you can write code that transforms the list into a tree. This is commonly called an operator precedence parser.[3]

Left Recursion and Other Traps

To avoid infinite recursion, you have to take care that each possible recursion cycle advances the cursor position by at least one character. In the `MathExpression` grammar, the only possible recursion cycle is `sum` → `product` → `term` → `group` → `sum`, and `group` can only match if it consumes an initial open parenthesis, `(`.

If recursion does not consume a character, it is called *left recursion* and needs special language support that Perl 6 does not offer. A case in point is

```
token a { <a>? 'a' }
```

which could match the same input as the regex `a+`, but instead loops infinitely without progressing.

A common technique to avoid left recursion is to have a structure where you can order regexes from generic (here `sum`) to specific (`number`). You only have to be careful and check for consumed characters when a regex deviates from that order (e.g., `group` calling `sum`).

Another potential source of infinite loops is when you quantify a regex that can match the empty string. This can happen when parsing a language that actually allows something to be empty. For instance, in UNIX shells, you can assign variables by potentially leaving the right-hand side empty:

```
VAR1=value
VAR2=
```

[3] https://en.wikipedia.org/wiki/Operator-precedence_parser

When writing a grammar for UNIX shell commands, it might be tempting to write a `token string { \w* }` that would potentially match an empty string. In a situation that allows for more than one string literal, `<string>+` can then hang, because the effective regex, `[\w*]+`, tries to match a zero-width string infinitely many times.

Once you are aware of the problem, the solution is pretty simple: change the token to not allow an empty string (`token string { \w+ }`), and explicitly take care of situations where an empty string is allowed:

```
token assignment {
    <variable> '=' <string>?
}
```

9.2 Starting Simple

Even though a grammar works from the top down, developing a grammar works best from the bottom up. It is often not obvious from the start what the overall structure of a grammar will be, but you usually have a good idea about the *terminal* tokens: those that match text directly without calling other subrules.

In the earlier example of parsing mathematical expressions, you might not have known from the start how to arrange the rules that parse sums and products, but it's likely that you knew you had to parse a number at some point, so you can start by writing:

```
grammar MathExpression {
    token number { \d+ }
}
```

This is not much, but it's also not very complicated, and it's a good way to get over the writer's block that programmers sometimes face when challenged with a new problem area. Of course, as soon as you have a token, you can start to write some tests:

```
grammar MathExpression {
    token number { \d+ }
}

multi sub MAIN(Bool :$test!) {
    use Test;
    plan 2;
    ok MathExpression.parse('1234',  :rule<number>),
        '<number> parses 1234';
    nok MathExpression.parse('1+4',  :rule<number>),
        '<number> does not parse 1+4';
}
```

Now you can start to build your way up to more complex expressions:

```
grammar MathExpression {
    token number { \d+ }
    rule product { <number>+ % '*' }
}

multi sub MAIN(Bool :$test!) {
    use Test;
    plan 5;
    ok MathExpression.parse('1234',  :rule<number>),
        '<number> parses 1234';
    nok MathExpression.parse('1+4',  :rule<number>),
        '<number> does not parse 1+4';

    ok MathExpression.parse('1234',  :rule<product>),
        '<product> can parse a simple number';
    ok MathExpression.parse('1*3*4', :rule<product>),
        '<product> can parse three terms';
    ok MathExpression.parse('1 * 3', :rule<product>),
        '<product> and whitespace';
}
```

It is worth it to include whitespace early on in the tests. The previous example looks innocent enough, but the last test actually fails. There is no rule that matches the whitespace between the `1` and the *. Adding a space in the regex between the `<number>` and the + quantifier makes the tests pass again, because the whitespace inserts an implicit `<.ws>` call.

Such subtleties are easy to catch if you start really simple and catch them as soon as possible. If instead you give in to the temptation of writing down a whole grammar from top to bottom, you can spend many hours debugging why some seemingly simple thing such as an extra space makes the parse fail.

9.3 Assembling Complete Grammars

Once you have written the basic tokens for lexical analysis, you can progress to combining them. Typically the tokens do not parse whitespace at the borders of their matches, so the rules that combine them do that.

In the `MathExpression` example in the previous section, `rule product` directly called `number`, even though we now know that the final version uses an intermediate step, rule `term`, which can also parse an expression in parentheses. Introducing this extra step does not invalidate the tests we have written for `product`, because the strings it matched in the early version still match. Introducing more layers happens naturally when you start with a grammar that handles a subset of the language, which you later expand.

9.4 Debugging Grammars

There are two failure modes for a regex or a grammar: it can match when it's not supposed to match (a false positive), or it can fail to match when it's supposed to match (a false negative). Typically, false positives are easier to understand, because you can inspect the resulting match object and see which regexes matched which part of the string.

There is a handy tool for debugging false negatives: the `Grammar::Tracer` module. If you load the module in a file containing a grammar, running the grammar produces diagnostic information that can help you find out where a match went wrong.

Note that this is only a diagnostic tool for developers; if you want to give end users better error messages, please read Chapter 11 for improvement suggestions.

You need to install the Perl 6 module `Grammar::Debugger`, which also contains `Grammar::Tracer`. If you use the `moritzlenz/perl6-regex-alpine` docker image, this is already done for you. If you installed Perl 6 via another method, you need to run

```
zef install Grammar::Debugger
```

on the command line. If `zef` is not yet installed, follow the installation instructions on the `zef` GitHub page.[4]

Let's look at the Perl 6 module `Config::INI`[5] by Tadeusz Sośnierz. It contains the following grammar[6] (slightly reformatted here):

```
grammar INI {
    token TOP {
        ^ <.eol>* <toplevel>?  <sections>* <.eol>*  $
            }
    token toplevel { <keyval>* }
    token sections { <header> <keyval>* }
    token header   { ^^ \h* '[' ~ ']' $<text>=<-[ \] \n ]>+
                     \h* <.eol>+ }
    token  keyval  { ^^ \h* <key> \h* '=' \h* <value>? \h*
                     <.eol>+ }
```

[4] https://github.com/ugexe/zef#installation

[5] https://github.com/tadzik/perl6-Config-INI

[6] https://github.com/tadzik/perl6-Config-INI/blob/master/lib/Config/INI.pm

```
    regex key      { <![#\[]> <-[;=]>+ }
    regex value    { [ <![#;]> \N ]+ }
    token eol      { [ <[#;]> \N* ]? \n }
}
```

Suppose we want to understand why it does not parse the following piece of input text:

```
a = b
[foo]
c: d
```

So, before the grammar, we insert the line

```
use Grammar::Tracer;
```

and after it, add a small piece of code that calls the `.parse` method of that grammar:

```
INI.parse(q:to/EOF/);
a = b
[foo]
c: d
EOF
```

This produces a sizable but fairly informative piece of output.

Each entry consists of a name of a regex, like `TOP` or `eol` (`for "end of line"`), followed by the indented output of the regexes it calls. After each regex comes a line containing an asterisk (*) and either `MATCH` followed by the string segment that the regex matched, or `FAIL` if the regex failed.

Let's look at the output piece by piece, even if it comes out in one chunk:

```
TOP
|   eol
|   * FAIL
|   toplevel
|   |   keyval
|   |   |  key
|   |   |  * MATCH "a "
|   |   |  value
|   |   |  * MATCH "b"
|   |   |  eol
|   |   |  * MATCH "\n"
|   |   |  eol
|   |   |  * FAIL
|   |   * MATCH "a = b\n"
|   |   keyval
|   |   |  key
|   |   |  * FAIL
|   |   * FAIL
|   * MATCH "a = b\n"
```

This tells us that `TOP` called `eol`, which failed to match. Since the call to `eol` is quantified with *, this does not cause the match of `TOP` to fail. `TOP` then calls `key`, which matches the text `"a"`, and `value`, which matches `"b"`. The `eol` regex then proceeds to match the newline character, fails on the second attempt (since there are no two newline characters in a row). This causes the initial `keyval` token to match successfully. A second call to `keyval` matches pretty quickly (in the call to `key`). Then, the match of token `toplevel` proceeds successfully, consuming the string `"a = b\n"`.

So far, this all looks as expected. Now let's take a look at the second chunk of output:

```
|  sections
|  |  header
|  |  |  eol
|  |  |  * MATCH "\n"
|  |  |  eol
|  |  |  * FAIL
|  |  * MATCH "[foo]\n"
|  |  keyval
|  |  |  key
|  |  |  * MATCH "c: d\n"
|  |  * FAIL
|  * MATCH "[foo]\n"
```

`TOP` next calls `sections`, wherein token `header` successfully matches the string `"[foo]\n"`. Then, `keyval` calls `key`, which matches the whole line `"c: d\n"`. Wait, that's not right, is it? We might have expected `key` to only match the `c`. I certainly wouldn't have expected it to match a newline character at the end. The lack of an equals sign in the input causes the regex engine to never even call regex `value`. But since `keyval` is again quantified with the star * quantifier, the match of the calling regex `sections` succeeds in matching just the header `"[foo]\n"`.

The last part of the `Grammar::Tracer` output follows:

```
|  sections
|  |  header
|  |  * FAIL
|  * FAIL
|  eol
|  * FAIL
* FAIL
```

It's FAILs from here on. The second call to `sections` again tries to parse a header, but its next input is still `"c: d\n"`, so it fails, as does the end-of-string anchor `$` in token `TOP`, failing the overall match in method `parse`.

So we have learned that regex `key` matched the whole line `c: d\n`, but since no equals sign (`=`) follows it, token `keyval` cannot parse this line. Since no other regex (notably not `header`) matches it, this is where the match fails.

As you can see from this example run, `Grammar::Tracer` enables us to pinpoint where a parse failure happens, even though we had to look carefully through its output to locate it. When you run it in a terminal, you automatically get colored output, with `FAIL` having a red and `MATCH` a green background, and token names standing out in bold white (instead of the usual gray) output. This makes it easier to scan from the bottom (where a failed match usually leaves a trail of red FAILs) up to the trailing successful matches, and then look in the vicinity of the border between matches and failures.

Since debugging imposes a significant mental burden, and the output from `Grammar::Tracer` tends to grow quickly, it is generally advisable to reduce the failing input to a minimal specimen. In the case described before, we could have removed the first line of the input string and saved ten lines of `Grammar::Tracer` output to look through.

9.5 Parsing Whitespace and Comments

As said before, the idiomatic way to parse insignificant whitespace is by calling `<.ws>`, typically implicitly by using whitespace in a rule. The default `ws` implementation, `<!ww>\s*`, works well for many languages, but it has its limits.

In a surprising number of file formats and computer languages, there is significant whitespace that `<.ws>` would just gobble up. These include INI files (where a newline typically indicates a new key/value pair), Python and YAML (where indentation is used for grouping), CSV (where a newline signals a new record), and Makefiles (where indentation is required to be with a tabulator character).

In these cases, it is best practice to override `ws` in your own grammar to match only insignificant whitespace. Let's take a look at a second, minimalistic INI parser, independently developed from the one described in the previous section:

```
grammar INIFile {
    token TOP { <section>* }
    token section {
        <header>
        <keyvalue>*
    }
    rule header {
        '['  <-[ \] \n ]>+ ']' <.eol>
    }
    rule keyvalue {
        ^^
        $<key>=[\w+]
        <[:=]>
        $<value>=[<-[\n;#]>*]
        <.eol>
    }
    token ws { <!ww> \h* }
    token eol {
        \n [\h*\n]*
    }
}
```

This parses simple INI configuration files like this:

```
[db]
driver: mysql
host: db01.example.com
port: 122
username: us123
password: s3kr1t
```

Take note how this grammar uses two paths for parsing whitespace: a custom `ws` token that only matches horizontal whitespace (blanks and tabs), and a separate token `eol` that matches (significant) line breaks. The `eol` token also gobbles up further lines consisting only of whitespace.

If a language supports comments, and you don't want them to appear in your parse tree, you can parse them either in your `ws` token, or in `eol` (or your equivalent thereof). Which one it is depends on where comments are allowed. In INI files, they are only allowed after a key/value pair or in a line on their own, so `eol` would be the fitting place. In contrast, SQL allows comments in every place where whitespace is allowed, so it is natural to parse them in `ws`:

```
# comment parsing for SQL:
token ws { <!ww> \s* [ '--' \N* \n ]* }

# comment parsing for INI files:
token eol { [ [ <[#;]> \N* ]? \n ]+ }
```

9.6 Keeping State

Some of the more interesting data formats and languages require the parser to store things (at least temporarily) to be able to correctly parse them. A case in point is the C programming language, and others inspired by its syntax (such as C++ and Java). Such languages allow variable declarations of the form `type variable = initial_value`, like this:

```
int x = 42;
```

This is valid syntax, but only if the first word is a type name. In contrast, this would be invalid, because `x` is not a type:

```
int x = 42;
x y = 23;
```

From these examples, it is pretty clear that the parser must have a record of all the types it knows. Since users can also declare types in their code files, the parser must be able to update this record.

Many languages also require that symbols (variables, types, and functions) be declared before they are referenced. This too requires the grammar to keep track of what has been declared and what hasn't. This record of what has been declared (and what is a type or not, and possibly other meta information) is called a *symbol table*.

Instead of parsing the full C programming language, let's consider a minimalist language that just allows assignments of lists of numbers, and variables to variables:

```
a = 1
b = 2
c = a, 5, b
```

If we don't impose declaration rules, it's pretty easy to write a grammar:

```
grammar VariableLists {
    token TOP        { <statement>* }
    rule  statement  { <identifier> '=' <termlist> \n }
    rule  termlist   { <term> * % ',' }
    token term       { <identifier> | <number> }
    token number     { \d+ }
    token identifier { <:alpha> \w* }
    token ws         { <!ww> \h* }
}
```

Now we demand that variables can only be used after they've been assigned to, so that the following input would be invalid, because `b` is not declared in the second line, where it's used:

```
a = 1
c = a, 5, b
b = 2
```

To maintain a symbol table, we need three new elements: a declaration of the symbol table, some code that adds a variable name to the symbol table when the assignment has been parsed, and finally a check whether a variable has been declared at the time we come across it in a term list:

```
grammar VariableLists {
    token TOP {
        :my %*SYMBOLS;
        <statement>*
    }
    token ws { <!ww> \h* }
    rule statement {
        <identifier>
        { %*SYMBOLS{ $<identifier> } = True }
        '=' <termlist>
        \n
    }
    rule termlist { <term> * % ',' }
    token term { <variable> | <number> }
    token variable {
        <identifier>
        <?{ %*SYMBOLS{ $<identifier> } }>
    }
    token number { \d+ }
    token identifier { <:alpha> \w* }
}
```

In the token `TOP`, `:my %*SYMBOLS` declares a variable. Declarations in regexes start with a colon (`:`), and end with a semicolon (`;`). In between they look like normal declarations in Perl 6. The `%` *sigil* signals that the variable is a *hash*—a mapping of string keys to values. The * makes it a *dynamic* variable—a variable that is not limited to the current scope but also visible to code (or regexes, which are also code) that is called from the current scope. Since this is an unusually large scope, it is custom to choose a variable in CAPITAL LETTERS.

The second part, adding a symbol to the symbol table, happens in the rule `statement`:

```
rule statement {
    <identifier>
    { %*SYMBOLS{ $<identifier> } = True }
    '=' <termlist>
    \n
}
```

Inside the curly braces is regular (non-regex) Perl 6 code, so we can use it to manipulate the hash `%*SYMBOLS`. The expression `$<identifier>` accesses the capture for the variable name.[7] Thus, if this rule parses a variable `a`, this statement sets `%*SYMBOLS{ 'a' } = True`.

The placement of the code block is relevant. Putting it before the call to `termlist` means that the variable is already known when the term list is parsed, so it accepts input like `a = 2, a`. If we call `termlist` first, this kind of input is rejected.

[7]At this point it is crucial that `identifier` does not parse its surrounding whitespace. Hence the principle that tokens do not care about whitespace, and the rules that call those tokens parse the whitespace.

Speaking of rejection, this part happens in token `variable. term` now calls the new token `variable` (previously it called `identifier` directly), and `variable` validates that the symbol has been declared before:

```
token term { <variable> | <number> }
token variable {
    <identifier>
    <?{ %*SYMBOLS{ $<identifier> } }>
}
```

You might remember from earlier examples that `<?{ ... }>` executes a piece of Perl 6 code, and fails the parse if it returns a false value. If `$<identifier>` is not in `%*SYMBOLS`, this is exactly what happens. At this time, the nonbacktracking nature of tokens is important. If the variable being parsed is `abc`, and a variable `a` is in `%*SYMBOLS`, backtracking would try shorter matches for `<identifier>` until it hits `a`, and then succeeds.[8]

Since `%*SYMBOLS` is declared in token `TOP`, you have to duplicate this declaration when you try to call rules other than `TOP` from outside the grammar. Without a declaration such as `my %*SYMBOLS;`, a call like

```
VariableLists.parse('abc', rule => 'variable');
```

dies with

```
Dynamic variable %*SYMBOLS not found
```

[8]In this case, this would be harmless, because no other rule could match the rest of the variable, leading to a parse error nonetheless. But in more complicated cases, this kind of unintended backtracking can lead to errors that are very puzzling for the maintainer of the grammar.

Implementing Lexical Scoping with Dynamic Variables

Many programming languages have the concept of a lexical scope. A *scope* is the area in a program where a symbol is visible. We call a scope *lexical* if the scope is determined solely by the structure of the text (and not, say, runtime features of the program).

Scopes can typically be nested. A variable declared in one scope is visible in this scope, and in all inner, nested scopes (unless an inner scope declares a variable of the same name, in which case the inner declaration hides the outer).

Coming back to the toy language of lists and assignments, we can introduce a pair of curly braces to denote a new scope, so this is valid:

```
a = 1
b = 2
{
    c = a, 5, b
}
```

but the next example is invalid, because it declares `b` only in an inner scope, and so it is not visible in the outer scope:

```
a = 1
{
  b = 2
}
c = a, 5, b
```

To implement these rules in a grammar, we can make use of an important observation: *dynamic scoping in a grammar corresponds to lexical scoping in text it parses*. If we have a regex `block` that parses both the delimiters of a scope and the things inside that scope, its dynamic

scope is confined to all of the regexes it calls (directly and indirectly), and that is also the extent of the lexical scope it matches in the input text.

Let's take a look at how we can implement dynamic scoping:

```
grammar VariableLists {
    token TOP {
        :my %*SYMBOLS;
        <statement>*
    }
    token ws { <!ww> \h* }
    token statement {
        | <declaration>
        |  <block>
    }
    rule declaration {
        <identifier>
        { %*SYMBOLS{ $<identifier> } = True; }
        '=' <termlist>
        \n
    }
    rule block {
        :my %*SYMBOLS = CALLERS::<%*SYMBOLS>;
        '{' \n*
            <statement>*
        '}' \n*
    }
    rule termlist { <term> * % ',' }
    token term { <variable> | <number> }
    token variable {
        <identifier>
        <?{ %*SYMBOLS{ $<identifier> } }>
    }
```

```
    token number { \d+ }
    token identifier { <:alpha> \w* }
}
```

There are a few changes to the previous version of this grammar: the rule `statement` has been renamed to `declaration` and the new rule `statement` parses either a `declaration` or a `block`.

All the interesting bits happen in the `block` rule. The line `:my %*SYMBOLS = CALLERS::<%*SYMBOLS>;` declares a new dynamic variable `%*SYMBOLS` and initializes it with the previous value of that variable. `CALLERS::<%*SYMBOLS>` looks through the caller, and the caller's caller, and so on for a variable `%*SYMBOLS`, and thus looks up the value corresponding to the outer scope. The initialization creates a copy of the hash, such that changes to one copy do not affect the other copies.

Let's take a look at what happens when this grammar parses the following input:

```
a = 1
b = 2
{
    c = a, 5, b
}
```

After the first two lines, `%*SYMBOLS` has the value `{a => True, b => True}`. When rule `block` parses the opening curly bracket on the third line, it creates a copy of `%*SYMBOLS`. The declaration of `c` on the fourth line inserts the pair `c => True` into the copy of `%*SYMBOLS`. After rule `block` parses the closing curly brace on the last line, it exits successfully, and the copy of `%*SYMBOLS` goes out of scope. This leaves us with the earlier version of `%*SYMBOLS` (with only the keys `a` and `b`), which then goes out of scope when `TOP` exits.

Scoping Through Explicit Symbol Tables

Using dynamic variables for managing symbol tables usually works pretty well, but there are some edge cases where a more explicit approach works better. Such edge cases include those where there are so many symbols that copying becomes prohibitively expensive, or where more than the topmost scope must be inspected, or when copying the symbol table is impractical for other reasons.

Consequently, you can write a class for your symbol table (which in the simplest case uses an array as a stack of scopes) and explicitly call methods on it when entering and leaving scopes, when declaring a variable, and for checking whether a variable is known in a scope:

```
class SymbolTable {
    has @!scopes = {}, ;
    method enter-scope() {
        @!scopes.push({})
    }
    method leave-scope() {
        @!scopes.pop();
    }
    method declare($variable) {
        @!scopes[*-1]{$variable} = True
    }
    method check-declared($variable) {
        for @!scopes.reverse -> %scope {
            return True if %scope{$variable};
        }
        return False;
    }
}
```

```
grammar VariableLists {
    token TOP {
        :my $*ST = SymbolTable.new();
        <statement>*
    }
    token ws { <!ww> \h* }
    token statement {
        | <declaration>
        |  <block>
    }
    rule declaration {
        <identifier>
        { $*ST.declare( $<identifier> ) }
        '=' <termlist>
        \n
    }
    rule block {
        '{' \n*
            { $*ST.enter-scope() }
            <statement>*
            { $*ST.leave-scope() }
        '}' \n*
    }
    rule termlist { <term> * % ',' }
    token term { <variable> | <number> }
    token variable {
         <identifier>
         <?{ $*ST.check-declared($<identifier>) }>
    }
    token number { \d+ }
    token identifier { <:alpha> \w* }
}
```

The class `SymbolTable` has the private array attribute `@!scopes`, which is initialized with a list containing a single, empty hash `{}`. Entering a scope means pushing an empty hash on top of this array, and when leaving the scope it is removed again through the `pop` method call. A variable declaration adds its name to the topmost hash, `@!scopes[*-1]`.

Checking for the presence of a variable must not just consider the topmost hash, because variables are inherited to inner scopes. Here we go through the all scopes in reverse order, from innermost to outermost scope. The order of traversal is not relevant for a simple Boolean check, but if you need to look up information associated with the variable, it is important to adhere to this order to reference the correct one.

Token `TOP` creates a new object of class `SymbolTable`, `declaration` calls the `declare` method, and token `variable` calls method `check-declared`. The rule `block` calls `enter-scope` before parsing the statement list, and `leave-scope` afterward. This works, but only if the statement list can be parsed successfully; if not, rule `block` fails before it manages to call `leave-scope`.

Perl 6 has a safety feature for such situations: if you prefix a statement with `LEAVE`, Perl 6 calls it for you at routine exit, in all circumstances where this is possible (even if an exception is thrown). Since the `LEAVE` *phaser*[9] only works in regular code and not in regexes, we need to wrap the regex in a method:

```
method block {
    $*ST.enter-scope();
    LEAVE $*ST.leave-scope();
    self.block_wrapped();
}
```

[9] https://docs.perl6.org/language/phasers

```
rule block_wrapped {
    '{' \n*
        <statement>*
    '}' \n*
}
```

Now we have the same robustness as the approach with dynamic variables, and more flexibility to add extra code to the symbol table, at the cost of more code and increased effort.

9.7 Summary

Perl 6 grammars are a declarative way to write recursive descent parsers. Without backtracking, they are predictive; at each point, we know what list of tokens to expect.

The recursive nature of grammars comes with the risk of left recursion, a situation where a recursive path does not consume any characters, and so leads to an infinite loop.

In spite of the top-down nature of grammars, writing them typically happens from the bottom up: starting with lexical analysis, and then moving up to parsing larger structures.

Complex languages require additional state for successful and precise parsing. We have seen how you can use dynamic variables to hold state in grammars, how their scope corresponds to lexical scoping in the input, and how symbol tables can be written and integrated into grammars.

CHAPTER 10

Extracting Data from Matches

So far we have put great effort into parsing various file formats, and the result of our efforts was a match object, or `Nil` if the match failed.

For most purposes, knowing whether a match succeeded or not is only the first step—we want to extract useful data from a successful match.

In principle, we could inspect the resulting match object and extract all the necessary data. If it does not offer enough resolution, we can add captures to make it more fine-grained. This approach can work, but it tends to be frustrating to write, and brittle in its result.

But before we look into a solution, let's talk more about the problem. The match object we get from a successful regex match or grammar parse is a *parse tree*. Its structure is determined directly by the structure of the grammar, and the information found in the parse tree is all about the strings matched, and the positions where they were found in the input string.

The reason we want to parse a string is typically to do some processing with its contents. If the input is an INI file, we want to extract the sections and their key/value pairs; if it is source code of a programming language, we might want to check it for some traits (like a linter), or compile it to a lower-level format. This processing step, whatever it is, usually should not be tied directly to the parse tree, but rather to a more abstract representation of the contents. This representation is called an *abstract syntax tree*, or *AST* for short.

M. Lenz, *Parsing with Perl 6 Regexes and Grammars*,
https://doi.org/10.1007/978-1-4842-3228-6_10

ASTs can look very different depending on the use case. For a JSON parser, the AST could directly be the data structure that had been serialized into the JSON string, so a mixture of arrays, hashes, strings, numbers, and Booleans. For a programming language, you typically construct an AST out of a collection of custom classes that capture all the aspects of the input that you care about; these AST classes typically also contain annotations referencing input line numbers, so that error messages can properly identify the location of the error.

10.1 Action Objects

Grammars and match objects come with two features that offer a more resilient and extensible approach to extracting data. The first and simpler one is that you can attach an arbitrary data structure to a `Match`[1] object, and later access it through the `.made` attribute:

```
if 'abc' ~~ /\w/ {
    $/.make({'a' => 'bc'});
    say $/.made;      # Output: {a => bc}
}
```

If the match object is in the special variable `$/`, you can also call `make DATA`:

```
if 'abc' ~~ /\w/ {
    make {'a' => 'bc'};
    say $/.made;     # Output: {a => bc}
}
```

You can probably see where this is going: you can build a data structure that is to be part of the final AST, and attach it to the match object with `make`.

[1] https://docs.perl6.org/type/Match

Where do you do that? One option would be inline code objects, { ... }, but that couples the grammar very tightly to the AST generation code. This is where the second feature comes in: action objects.

An action object is one that you pass to a .parse or .subparse call on a grammar. Henceforth, whenever a named regex matches successfully, the regex engine calls a method for you; it searches for a method with the same name as the regex, and if one exists, calls it with the match object as the argument. If such a method does not exist, nothing happens, and no error is raised.

Here is an example of an action that evaluates a mathematical expression as it is parsed:

```
grammar MathExpression {
    token TOP    { <sum> }
    rule sum     { <product>+  % '+' }
    rule product { <term>+ % '*' }
    rule term    { <number> | <group> }
    rule group   { '(' <sum> ')' }
    token number { \d+ }
}

class MathEvalAction {
    method TOP($/) {
        make $<sum>.made;
    }
    method sum($/) {
        make [+] $<product>».made;
    }
    method product($/) {
        make [*] $<term>».made;
    }
    method term($/) {
        make $/.values[0].made;
    }
```

```
    method group($/) {
        make $<sum>.made;
    }
    method number($/) {
        make $/.Int;
    }
}

my $match = MathExpression.parse(
    '4 + 5 * (1 + 3)',
    actions => MathEvalAction.new,
);
say $match.made;      # Output: 24
```

The `MathExpression` grammar should be familiar from the previous chapter; what's new is another class, `MathEvalAction`, that has a method for each regex in the grammar. Each method takes `$/` as its sole argument, to which the match object is bound. It calls `make` to attach something to this match object.

The idea in this class is that each method attaches a number to the current match object corresponding to the current match. Hence, if it parses the string `2 * (1+4)`, it attaches the number 2 to the match object that parsed the `2`. It does the same for the numbers 1 and 4, and then attaches the result of the sum, 5, to the match object that parsed `1+4`, then again the number 5 to the match object that parsed `(1+4)`, and finally it evaluates the product, and attaches the number 10 to the top-level match object.

Reading from bottom to top, the `number` method converts the string to a number by calling the `Int` method on the match object. It attaches the resulting integer to the match object. Match `group` simply takes the attached value from the match from `$<sum>`, and attaches it to its own match object.

The rule `term` matches one of two alternatives. `$/.values` returns a list of all match objects, which is always one element. `make $/.values[0].made` thus propagates the attached value from either match and attaches it to its own match object.

The rule `product` parses a list of terms, separated by asterisks. The action method of the same name takes the numbers attached to each term and multiplies them. The `».` syntax in `$<term>».made` calls the method `made` on each element of the list `$<term>`, and returns a list of all the results. If you are unable to write the » character with your keyboard, you can instead use `>>`, so this becomes `$<term>>>.made`.

`[*] LIST` inserts the * operator between each element of the `LIST`, hence calculating the product of all these values.

Since the rules `sum` and `product` have the same structure, so do their action methods. `sum` uses `[+]` to create the sum of all the numbers attached to the match objects one level below.

Finally, token `TOP` just calls `sum`, so method `TOP` in the action class just passes on the value attached to `$<sum>`.

Here is a match tree, annotated with the value of each `.made` in the right column:

```
sum => ｢4 + 5 * (1 + 3)｣              24
 product => ｢4 ｣                       4
  term => ｢4 ｣                         4
   number => ｢4｣                       4
 product => ｢5 * (1 + 3)｣              20
  term => ｢5 ｣                         5
   number => ｢5｣                       5
  term => ｢(1 + 3)｣                    4
   group => ｢(1 + 3)｣                  4
    sum => ｢1 + 3｣                     4
     product => ｢1 ｣                   1
      term => ｢1 ｣                     1
       number => ｢1｣                   1
```

```
product => ｢3｣                    3
 term => ｢3｣                      3
  number => ｢3｣                   3
```

In each branch, the most indented matches are those whose action method is called first, so `number` before `term` before `product` before `sum`. This ordering guarantees that each action method can rely on the presence of the `.made` attributes on submatches.

Writing an action method needs only knowledge of the captures of the corresponding regex and what their `.made` attribute contains. At no point in the action did we need to look deeper into the nested structure of a match tree. If you structure your actions this way, you are flexible to change the grammar. Whenever you modify a regex, you can rest assured that you only have to touch the action method corresponding to it, and you never have to worry that another action method might depend on the regex you just touched.

When you write an action method for proto tokens, you should be aware that action methods are called for the successfully matching candidate, but not for the proto as a whole. So if your grammar contains the lines

```
proto token value {*};
token value:sym<string> { <string> }
token value:sym<number> { <number> }
```

then the grammar engine might call a method called `value:sym<string>` or a method `value:sym<number>` for you, but not a method `value`.

10.2 Building ASTs with Action Objects

The previous section illustrated the mechanics of action objects and methods, and it built something akin to an AST, but it wasn't a tree. Since trees are the most common result of a successful parse, I want to present another action class for the same grammar that creates an abstract syntax tree during the parse.

To save us the trouble of writing a separate tree class, we'll use a nested array to represent the tree. The first element of the tree will be the operator; further elements are the operands. Thus, for the expression `1 + 2 + 3`, the result is `['+', 1, 2, 3]`. Since nested expressions are indicated by nested arrays, we don't need an operator for a group of parentheses: hence the expression `2 * (3 + 4)` is represented as `['*', 2, ['+', 3, 4]]`.

Let's take a look at the action class, and how we can use it:

```
class MathASTAction {
    method reduce($op, @list) {
        return @list[0] if @list.elems == 1;
        return [$op, |@list];
    }
    method TOP($/) {
        make $<sum>.made;
    }
    method sum($/) {
        make self.reduce('+', $<product>».made);
    }
    method product($/) {
        make self.reduce('*', $<term>».made);
    }
    method term($/) {
        make  $/.values[0].made;
    }
```

```
    method group($/) {
        make $<sum>.made;
    }
    method number($/) {
        make $/.Int;
    }
}

say $match = MathExpression.parse(
    '4 + 5 * (1 + 3)',
    actions => MathASTAction.new,
);
say $match.made.perl;
```

This code (along with the grammar definition from the previous section) produces the output

```
["+", 4, ["*", 5, ["+", 1, 3]]]
```

The action methods for `TOP`, `term`, `group`, and `number` are identical to those from the previously discussed action class. The interesting parts are the action methods for `sum` and `product`. Let's look at the latter, representative of both:

```
method reduce($op, @list) {
    return @list[0] if @list.elems == 1;
    return [$op, |@list];
}

method product($/) {
    make self.reduce('*', $<term>».made);
}
```

If we used just `make ['*', |$<term>».made]` as the body of method `product`, the AST for `1 + 2` would come out as `['+', ['*', 1], ['*', 2]]`, because each sum parses a product expression. We don't want that extra level of one-element multiplications, so we make a special case. If only one term is parsed, return just that term. Otherwise, return an array made up of the operator, and then the ASTs from all the terms.

Since this special case is used twice (for `sum` and for `product`), it makes sense to factor it out into a method, which I've called `reduce`. The method `reduce` uses the syntax `[$op, |@list]` to create an array of `$op` concatenated with the elements of `@list`. The `|` flattens out `@list`; if we didn't use it, the result would be `['+', [1, 2]]` instead of `['+', 1, 2]`.

10.3 Keeping State in Action Objects

The action classes from the previous section were a collection of methods, but they did not keep any state. One could argue they weren't "real" objects. In fact, we could remove the `.new` calls on the action classes, and pass the classes straight into the grammar, and the examples would work as before. The state is the AST that is being built and it is stored in the `.made` attributes of the match objects.

That is a common pattern, but it doesn't have to be this way. The objects that you use as action objects are normal Perl 6 objects, and you can do anything you want with them. You can store data in them, and even reuse the same action object across multiple calls to your grammar's `.parse` method.

Reading that, you might wonder why we went through the trouble of keeping a symbol table in a dynamic variable in an earlier example, instead of using an action object. The distinction here is that the symbol table affected the outcome of the parse; referencing an undeclared variable made the match fail. Grammars are meant to be independent of their action objects, so that you can modify and even swap out action classes without influencing the grammar.

Here is an example of a grammar that parses a single variable assignment and collects these variables in a hash:

```
class VariableCollection {
    has %.definitions;

    method TOP($/) {
         %.definitions{ $<identifier> } = $<termlist>.made;
    }
    method number($/)     { make $/.Int }
    method identifier($/) { make $/.Str }
    method termlist($/)   { make $<term>».made }
    method term($/)       { make $/.values[0].made }
}

grammar VariableAssignment {
    rule  TOP        { \s* <identifier> '=' <termlist> }
    rule  termlist   { <term>  * %  ',' }
    token term       { <identifier> | <number> }
    token number     { \d+ }
    token identifier { <:alpha> \w* }
    token ws         { <!ww> \h* }
}

my $actions = VariableCollection.new;

my @lines = 'a = 1', 'c = a, 5, b', 'b = 2';

for @lines -> $line {
    unless VariableAssignment.parse($line, :$actions) {
        die qq[Invalid input: "$line"];
    }
}
```

```
say $actions.definitions;
        # Output: {a => [1], b => [2], c => [a 5 b]}
```

In this example, `has %.definitions` declares a public attribute; a piece of state attached to the object that can be accessed from the outside. The syntax `:$actions` is shorthand for `actions => $actions`.

This pattern of using a stateful action object and parsing partial input one item at a time is very useful for stream processing; if you receive data piecewise and need to access results before you have received the complete input, or if your system runs continuously and there is an unceasing flow of input.

10.4 Summary

If you supply an action object to your grammar's `parse` call, its method of the same name as the regexes is called for you. You can use this to build an abstract syntax tree from your grammar.

Furthermore, you can keep state in this action object and use this mechanism to parse piecewise but related input.

CHAPTER 11

Generating Good Parse Error Messages

When a regex match or a grammar parse fails, it returns `Nil`. If this is a case where a failure was not expected, we can inform the user that some form of input (maybe a configuration file) was invalid.

For a simple input, like an e-mail address or a URL, this piece of binary information might be satisfactory. For a larger configuration or source file, we really want to give the user more information. Why is the input invalid? And if we can't answer that, where is the error?

Or put another way: when a parse fails, we want to identify the location, and possibly the reason for the parse fail, and generate a good error message.

11.1 Exploring the Problem

There is no general mechanism built into Perl 6 that produces good parse error messages, and I don't see how there could be such a mechanism. The reason is that even in a successful parsing run, it is normal for individual regexes and tokens to fail.

M. Lenz, *Parsing with Perl 6 Regexes and Grammars*,
https://doi.org/10.1007/978-1-4842-3228-6_11

Let's consider just two lines from an earlier grammar, `VariableLists`:

```
rule  termlist { <term> * % ',' }
token term     { <identifier> | <number> }
```

Suppose this matches the string `1,a`. The token `term` first tries to match `identifier` and `number` against the string. The `identifier` regex fails; however, `number` succeeds. Then, `termlist` itself matches the comma, and it's `term`'s turn again. This time the `identifier` branch successfully matches the string `a`, and `number` fails to match. Finally, `termlist` tries to match the next comma, and fails, but the overall `termlist` match succeeds.

In this example, we've seen three failed matches as part of a successful match. So what distinguishes such a "normal" failed match from one that fatally causes a whole parse to abort?

Technically, a fatally failing regex is one whose failure propagates upward all the way to `TOP`. But that doesn't help us create good error messages. By the time it fails, we don't yet know if it's going to propagate up the whole way. And once it has propagated up to regex `TOP`, it's too late to extract useful information from a failed match.

A different perspective of match failures offers more insight. There are basically two ways that a match can go wrong. One case is when we start to parse something, we can only match the first part of it, but not the rest. In the case of mathematical expressions, this could be an opening parenthesis where there is no corresponding closing parenthesis, or a plus sign not followed by any expression. The second case is where one or more alternatives are expected, but none match. For instance, a JSON parser might expect an object, an array, a number, or a string, but the input character is a $—something that simply isn't valid JSON outside of a quoted string.

11.2 Assertions

Sequential alternatives allow you to create good error messages. The general pattern is this:

```
rule group {
     '(' <sum>
    [')' || { die "Cannot find closing ')'" } ]
}
```

This regex tries to match the closing parenthesis. If this does not work, rather than silently failing, it uses a `{ ... }` code block to throw an exception using the built-in `die` function. The exception aborts the parse, and if no caller of the parse method catches it, the program exits with an unsuccessful return code. To be an effective tool for improving error reporting, we have to apply this same pattern to many more locations.

Describing the error is only enough in the most trivial of inputs. In any multiline input, we need to point out the location of the error; otherwise, the user will have a hard time fixing the problem.

We can access the position through the match object `$/`, and use that to calculate a line number. Since this is something we will need in quite a few places, it is best to extract that logic into a method. We can then call this method from a regex with the same syntax as we would call a regex, with the addition of a pair of parentheses around an argument:

```
grammar MathExpression {
    token TOP            { <sum> }
    rule sum             { <multiplication>+ %  '+' }
    rule multiplication { <term>+ % '*' }
    rule term            { <number> | <group> }
    rule group {
          '(' <sum>
        [ ')' || <error("no closing ')'")> ]
    }
```

```
    token number        { \d+ }

    method error($msg) {
        my $parsed = self.target.substr(0, self.pos);
        my $line-no = $parsed.lines.elems;
        die "Cannot parse mathematical expression: "
            ~ "$msg at line $line-no";
    }
}

say MathExpression.parse("(\n1");
```

What was previously

```
[ ')' || {die "Cannot find closing ')'" } ]
```

is now

```
[ ')' || <error("no closing ')'")> ]
```

Note that we replaced the direct `die` function call with a call to the custom `error` method. This method has a few tricks up its sleeve to determine the line number: the `.target` method returns the string that the grammar currently matches against and `self.pos` extracts the current parsing position. `self.pos` is equivalent to `$/.to`, except that the latter only works at the end of a parse, or when a capture has been generated.

Where do these methods come from? By declaring a type a grammar, it automatically gets the parent class `Grammar`[1] (which provides the `parse` and `subparse` methods), which in turn inherits from `Cursor`,[2] which supplies the `.target` and `.pos` methods.

[1] https://docs.perl6.org/type/Grammar
[2] https://docs.perl6.org/type/Cursor

The `Str.lines`[3] method decomposes a string into a list of lines it contains; calling `.elems` on this list produces the number of lines. For a string without any line breaks, this produces the number 1. Note than even the most hardcore C programmers count line numbers starting from one (not zero), so the number from `lines.elems` fits our purpose as a line number.

The output from the example code is as follows:

```
Cannot parse mathematical expression: no closing ')' at line 2
```

The error reporting correctly reports the error as being on line number 2 since the `\n` in the input string introduced a line break.

11.3 Improved Position Reporting

The `error` method we saw in the previous section is a good first approximation, but there are more things we can do to improve its accuracy.

The first is pretty simple. In many cases, an implicit `<.ws>` call can match whitespace before a `[ <something> || <error(...)>]` block. In a grammar where `ws` can match newlines, this can move the reported error location down, which doesn't match our intuitive understanding of how we want error location reported.

We can remedy this by removing the trailing whitespace from the `$parsed` string:

```
my $parsed = self.target.substr(0, self.pos)\
             .trim-trailing;
```

Another improvement strategy is to provide some context in the error message. For example, the Rakudo Perl 6 parser prints the source code around the error location and marks the error location with an eject symbol, ⏏.

[3] https://docs.perl6.org/type/Str#routine_lines

We can do the same in our own grammars:

```
method error($msg) {
    my $parsed = self.target.substr(0, self.pos)\
                 .trim-trailing;
    my $context = $parsed.substr($parsed.chars -  10 max 0)
          ~ '⏏' ~ self.target.substr($parsed.chars, 10);
    my $line-no = $parsed.lines.elems;
    die "Cannot parse mathematical expression: $msg\n"
        ~ "at line $line-no, around " ~ $context.perl
        ~ "\n(error location indicated by ⏏)\n";
}
```

This version of the `error` method constructs a context string of the ten characters before and after the error location (though it uses `$chars max 0` to prevent negative indices). It produces output like this:

```
Cannot parse mathematical expression: no closing ')'
at line 1, around "(1 + 2⏏"
(error location indicated by ⏏)
```

or with the input string `"1 + 2 5"`, you get

```
Cannot parse mathematical expression: no closing ')'
at line 1, around  "(1 +  2⏏  5"
(error location indicated by ⏏)
```

Note that this extra information is only useful if the reported position actually matches that of a syntax error. If you are not confident in the correctness of the reported error location, you should omit such details.

11.4 High-Water Marks

The technique discussed so far relies on manual insertion of `|| <error(...)>` pieces into the grammar. If you want to avoid this—by sacrificing a bit of clarity in the error messages—you can use another technique. Even if not, you might use it in combination with explicit error reporting.

Just like a high-water mark is pushed a bit higher each time a sea or river level reaches a new record, we can record the furthest position that our grammar has reached. We can do this through a dynamic variable that we set to `self.pos` if `self.pos` is larger than the current value of the variable. This high-water mark then forms the basis of our error location reporting.

A handy place to do such recording is the `ws` token, because it is already called in many places, and it is a natural delimiter of primitive tokens. Whenever the regex position reaches a new high-water mark, we can also record the name of the rule that called `ws`. This information is available through the `callframe`[4] built-in subroutine:

```
method ws() {
    if self.pos > $*HIGHWATER {
        $*HIGHWATER = self.pos;
        $*LASTRULE = callframe(1).code.name;
    }
    callsame;
}
```

The `callsame` function calls the `ws` method from the parent class (which in a grammar is `Grammar`), passing along the same arguments as the current method received. In other words, the `ws` method you see here wraps the default `ws` regex. We could delegate to any other named regex in the grammar here simply by calling it, for instance with `self.customws`.

[4] https://docs.perl6.org/type/CallFrame#sub_callframe

This ws method assumes that the dynamic variables $*HIGHWATER and $*LASTRULE have been declared somewhere. If we do that in token TOP (as seen in a previous example), we run into a problem: once the parse fails, the variables aren't available anymore for reporting the error. Again a wrapper comes to the rescue: we can wrap the parse method that all grammars have:

```
method parse($target, |c) {
    my $*HIGHWATER = 0;
    my $*LASTRULE;
    my $match = callsame;
    self.error($target) unless $match;
    return $match;
}
```

The syntax |c in a signature captures all remaining arguments, both named and positional. This ensures that calls like MathExpression.parse($string, rule => 'multiplication') continue to work. callsame then passes the same arguments to the original parse method.

The crucial part is the call to the error method when the match failed. Of course, the error method also needs some fiddling to make use of the $*HIGHWATER and $*LASTRULE dynamic variables:

```
method error($target) {
    my $parsed = $target.substr(0, $*HIGHWATER)\
                 .trim-trailing;
    my $line-no = $parsed.lines.elems;
    my $msg = "Cannot parse mathematical expression";
    $msg ~= "; error in rule $*LASTRULE" if $*LASTRULE;
    die "$msg at line $line-no";
}
```

This `error` method uses `$*HIGHWATER` instead of `self.pos` to determine the position where the parse failed and `$*LASTRULE` for reporting the last parsing rule that made progress.

There is another change of note here: instead of accessing `self.target`, the `error` method gets the target string as an argument. This is due to a detail in grammar mechanics that we have glossed over so far. When you call `SomeGrammar.parse(...)`, this is a method call on the type object `SomeGrammar`. But the grammar engine needs to keep state somewhere, so the method `parse` creates an instance of the grammar—the very same instance that you can access in `self` inside a method called from a regex. This detail is important, because method `parse` calls `error` *after* the parse has finished, so the instance is gone. Trying to call `self.target` or `self.pos` dies with the message `Cannot look up attributes in a MathExpression type object`.

This high-water mark–based error reporting correctly identifies `1+` as an unfinished sum, `1*` as an unfinished multiplication, and `(1` as an incomplete group expression.

11.5 Parser Combinator and FAILGOAL

Parsing matching pairs of delimiters with some content between them is common enough that Perl 6 has a special syntax for it. Instead of writing

```
token group {
    '(' <sum> ')'
}
```

you can write

```
token group {
    '(' ~ ')' <sum>
}
```

where the ~ is a parser combinator that rearranges the following tokens. In this example, the visual benefit is not large, but if the regex expression for the delimited content is longer, it is nice to have the delimiters closer together.

More importantly, if the closing delimiter cannot be found, the regex engine calls a method called FAILGOAL. In the default implementation, this just fails the match. We can override this method to produce an error message:

```
grammar MathExpression {
    token TOP            { <sum> }
    rule sum             { <multiplication>+ %  '+' }
    rule multiplication { <term>+ % '*' }
    rule term            { <number> | <group> }
    rule group           { '(' ~ ')' <sum> }
    token number         { \d+ }

    method FAILGOAL($goal) {
        my $cleaned = $goal.trim;
        self.error("No closing $cleaned");
    }

    method error($msg) {
        my $parsed = self.target.substr(0, self.pos)\
                     .trim-trailing;
        my $context = $parsed.substr($parsed.chars - 10 max 0)
            ~ '⏏' ~ self.target.substr($parsed.chars, 10);
        my $line-no = $parsed.lines.elems;
        die "Cannot parse mathematical expression: $msg\n"
             ~ "at line $line-no, around " ~ $context.perl
             ~ "\n(error location indicated by ⏏)\n";
    }
}

MathExpression.parse('(1');
```

The FAILGOAL method's argument is the source code of the regex part that matches the closing delimiter. The trim[5] method removes surrounding whitespace, the output of which we feed into a version of the error method seen earlier in this chapter.

The output is as follows:

```
Cannot parse mathematical expression: No closing ')'
at line 1, around "(1⏏"
(error location indicated by ⏏)
```

11.6 Which Techniques to Use?

There are no hard-and-fast rules on which error reporting techniques to use for which kind of project. The only rule is that the more people who actually use your grammar, the more effort you should put into making the error messages good.

You can combine all these techniques too: use the [<expected> || <error('...')>] pattern in place where you need it, put a FAILGOAL method into place, and use the high-water mark technique to get a good error position in those cases where you did not explicitly install an assertion for better error reporting.

You might be put off by the fact that in the later examples in this chapter, over half of the source code of the grammar is now dedicated to error reporting. But we used a pretty simplistic grammar to start with, which artificially reduced its size.

Moreover, most of the error reporting code is pretty generic, and can be factored out into a role, and reused in many grammars. That is what the Grammar::ParseError[6] module offers.

[5] https://docs.perl6.org/type/Str#method_trim
[6] https://modules.perl6.org/dist/Grammar-ErrorReporting

Finally, I'd argue that good parse errors are important enough to warrant this kind of attention.[7]

11.7 Summary

Generating good error messages from failed parsing processes is not easy because failing regexes are a normal part of parsing. Thus, we can generate error messages only if, at some point, we decide to commit to a parsing outcome, and produce an error message when that commitment did not hold up.

This commitment can be explicit in the form of a sequential alternative that triggers an exception in its second branch. Or we can make it implicit whenever a rule parses whitespace, using the high-water mark method.

Finally, the ~ parser combinator allows you to match a piece of input that is enclosed by delimiters, and offers a hook to improve the error message when the closing delimiter does not match.

[7]Sadly, much of the academic literature on parsing ignores this aspect of parsing, presumably because it is hard to quantify the quality of an error message.

CHAPTER 12

Unicode and Natural Language

Text that computers deal with tends to fall into two categories: things that are meant to be consumed by humans (like prose), and things that are meant to be consumed by software (machine code and encrypted files come to mind).

The data formats we have looked at so far fall somewhere in the middle: they are made to be unambiguously understood by software, but still able to be read and written by humans.

In this chapter, we take a brief look into the complexities hiding in the realm of natural language—text aimed squarely at human consumption, but still stored as a series of bytes.

This chapter is not an introduction to natural language processing; its intention is to discuss issues that you have to be aware of when processing multilanguage input.

12.1 Writing Systems

Written English is based on an *alphabet*, a set of letters usually corresponding to *phonemes* (elements of spoken language).

M. Lenz, *Parsing with Perl 6 Regexes and Grammars*,
https://doi.org/10.1007/978-1-4842-3228-6_12

Abjads or *consonant alphabets* follow a similar structure, but only contain consonants. There may or may not be vowel marks that indicate where vowels should be inserted while speaking the words, but those markers don't generally indicate *which* vowel to insert. Arabic and Hebrew fall into this category, as do some of the ancient North African and West Asian scripts.

A *syllabary* is a writing system in which each syllable has its own character. A famous example is Hiragana, a component of the Japanese writing system.

In contrast, *ideograms* convey an idea or a meaning. They appear in the Chinese writing system, often in combination with phonetic characters, resulting in several thousand characters in total.

There are a few places where different writing systems affect how programmers deal with text processing: you cannot assume that words are actually made out of letters. In general, *text segmentation* (the process of splitting up text into words, sentences, and other units) becomes rather language specific. Not all languages and writing systems use spaces to delimit words, or even delimit words at all. See the Unicode technical report on text segmentation[1] for more information.

The built-in character classes such as `\d` and `\w` match characters from all scripts, not just the Latin script. This means that if you use `\d+` to match a number, it could match `42` as well as ٤٢ (which are Eastern Arabic numerals[2]). While this might be surprising, converting the string "٤٢" to a number works just the same as with our "normal" Arabic digits.

[1] http://unicode.org/reports/tr29/

[2] https://en.wikipedia.org/wiki/Eastern_Arabic_numerals

12.2 Bytes, Code Points, Graphemes, and Glyphs

A computer stores text as a series of bytes. An *encoding* such as UTF-8, UTF-32, or ISO-8859-1 describes how a byte or a sequence of bytes maps to a *code point*.

A code point is the Unicode consortium's description of a character as it is used in human language. It consists of a number between zero and about 1.1 million, as well as a description of the character using uppercase ASCII characters, like `GREEK SMALL LETTER OMICRON`. The Unicode properties database records further information, such as if the character is a letter, a lowercase letter, is typically part of left-to-right writing, and so on.

Grapheme Clusters

Not every code point corresponds to a full character. Some languages allow free combination of base characters with *combining characters*. You might be familiar with the *acute* and *grave* accents used in French, as in "*née*" (born) or "*après*" (after). For compatibility with other legacy encodings, the characters é and è exist as stand-alone code points, so called *precomposed characters*. But they can also be written as the base character e, followed by the `COMBINING ACUTE ACCENT` or the `COMBINING GRAVE ACCENT`.

In French, the acute and grave accents go only on a few base characters, but in other writing systems, such restrictions don't apply. For instance in the Devanagari script, common on the Indian subcontinent, vowels can be written as combining characters that are applied to a consonant base character. In such cases, not all possible combinations of base and combining characters exist as precomposed Unicode codepoints.

This means something a reader would recognize as a character might be stored as several code points—typically a base character followed by one or more combining characters. We call this a *grapheme cluster*, or *grapheme* for short.

In Perl 6, the basic unit of a string is the grapheme cluster. If you ask a string for its number of characters by calling its `chars` method, the answer is the number of grapheme clusters:

```
say "e\c[COMBINING GRAVE ACCENT]".chars;            # Output: 1
```

The `\c[NAME]` syntax inserts a code point by name into a double-quoted string.

In this case, a precomposed character happens to exist, but this is not necessary for Perl 6 to consider it a single grapheme; therefore, if you chose `x` as the base character, no precomposed character exists, and Perl 6 still classifies it as a single character.

In regexes it is no different: any construct that can match a single character, such as a character class, can match a grapheme cluster:

```
if "e\c[COMBINING GRAVE ACCENT]" ~~ / ^ . $ / {
    say "A single grapheme cluster";
}
```

When text is rendered, either for printing or for display on a screen, the rendering engine can choose to combine several grapheme clusters into a single *glyph* (a shape that can be displayed). For instance, in English, the sequence of the letters `f` and `i` might be rendered as a *ligature*, a single glyph that contains two characters, which is typically narrower than the two characters separately.

To ignore combining characters while matching a regex, you can use the `:ignoremark` modifier; `:m` for short. Thus, `/:ignoremark uber/` matches the string `"über"`. This is useful when you suspect a user might have trouble entering diacritic marks, but still want to give them search results that include such marks.

Glyphs

The process of selecting glyphs during rendering is highly dependent on the output medium and the font, and hence Perl 6 offers no built-in handling of strings at the glyph level.

You should be aware that even in monospaced fonts, Chinese, Japanese, and Korean don't fit into the space of a single regular character, but take up twice as much horizontal space. *Full-width* characters are (typically Latin) characters stretched to the same width as a Chinese character. You might need to consider the width of a character when parsing indented blocks or horizontally aligned tables.

The main takeaway is that, even with grapheme clusters and monospaced fonts, you can't assume much about the rendered width of a character.

12.3 Unicode Properties

The huge character repertoire in Unicode makes it impractical and error-prone to enumerate all characters in any given category, like letters, numbers, punctuation, mathematical symbols, etcetera. Instead, you can specify that you want to a match a character from a given category. These are character classes defined by the Unicode consortium.

Unicode properties are written with a colon (`:`) followed by the property name inside angle brackets. For example, `<:Letter>` matches any letter—at the time of writing, a total of 63409 code points. A new Unicode version could expand that total even further.

You can freely combine Unicode properties by using `+` and `-`, thus `<:Letter+:Digit>` matches either a letter or a digit. You can also combine property-based character classes with enumerations. If you want to match a word without the character `e`, you can do that with the regex `/<:Letter-[eE]>+/`.

The classification of a character in letter, digit, punctuation, etcetera is called the *general category*, and you can query it with the `uniprop` method of a character:

```
say "a".uniprop;                          # Output: Ll
```

Here `Ll` is a shorthand for `Lowercase_Letter`. The full set of possible values is available from the official documentation.[3]

There are more categories that you can query:

```
say "a".uniprop('Script');                # Output: Latin
say "a".uniprop('Block');                 # Output: Basic Latin
```

The same combination of category and property can be used in a regex; `/<:Block('Basic Latin')>/` matches any character from the "Basic Latin" block.

In the case of grapheme clusters, both the `uniprop` method and the regex match against a property apply to the base character, and hence the string `"x\c[COMBINING TILDE]"` can be successfully be matched by `<:Letter>`, but not by `<:Mark>`, the category that identifies combining mark code points.

12.4 Summary

The wide variety of natural languages and writing systems implies that many assumptions we (often implicitly) make about language don't hold: not every logical or visual character is a single code point, not all words are made out of letters, and not all words are delimited by spaces, to name but a few.

Perl 6 offers some relief by treating grapheme clusters as single characters, and by offering solid support for matching by Unicode property inside character classes.

[3] https://docs.perl6.org/language/regexes#Unicode_properties

CHAPTER 13

Case Studies

The previous chapters have equipped you with knowledge of how grammars work, about error reporting, and data extraction.

Let's take a look at some real-world input formats and how we can parse them.

The main restriction for these case studies is size: parsing a language like SQL is quite approachable with grammars, but is a big task due to the sheer vastness of the language. You likely wouldn't enjoy reading 40 pages about it.

We will explore parsing three formats, with emphasis on different techniques. S-Expressions are a simple but fairly typical data serialization format, with a small amount of whitespace significance. Then, we will explore parsing mathematical expressions, but with a slightly different approach than in the examples from earlier chapters. We use an operator precedence parser to keep the grammar flat. Finally, a Python-like minilanguage can teach us how to parse an indentation-based format.

13.1 S-Expressions

Symbolic Expressions, better known as *S-Expressions*, are a notation for trees as nested lists that are used for the Lisp programming language.

In S-Expressions, a list is delimited by parentheses, and elements in a list are separated by whitespace. The S-Expression

```
(a b (c d))
```


M. Lenz, *Parsing with Perl 6 Regexes and Grammars*,
https://doi.org/10.1007/978-1-4842-3228-6_13

corresponds to the Perl 6 data structure

```
["a", "b", ["c", "d"]]
```

List elements are usually called *atoms*. An atom can be an unquoted identifier, and can usually contain all sorts of punctuation. For example `(+ 1 2)` is the Lisp expression for `1 + 2`. You can quote atoms containing whitespace by using double quotes:

```
(display "Hello, World")
```

Most dialects in the large family of Lisp programming languages use S-Expressions both for program code and for data structures. They differ, however, in what exactly is allowed in an S-Expression. Some allow floating-point literals, and others differ in what characters are allowed inside an identifier.

Parsing S-Expressions

Let's start with parsing atoms. An atom can be an integer like `12345`, an identifier such as `lisp`, or a quoted string `"like this"`. In an extensible grammar, such a list of alternatives should be implemented as a proto token:

```
grammar S-Expression {
    proto token atom {*}
    token atom:sym<identifier> {
        <[ a..z A..Z =*: ]>
        <[ a..z A..Z 0..9 _ =*: ]>*
    }
    token atom:sym<integer>    { <[+-]>? <[0..9]>+           }
    token atom:sym<string>     { '"' ~ '"' <string_contents> }
    token string_contents      { [ | <-[\\"]>+ | \\ . ]*     }
}
```

Matching a potentially signed integer is pretty straightforward.

The difficulty with S-Expression identifiers is not how to parse them, but deciding what to allow in them. This depends on the Lisp dialect being parsed. In the preceding case, the first character allows Latin letters and the symbols =, *, and :; any following character also allows a digit and the underscore. This is constructed such that an integer is unambiguously parsed as such, and cannot also be parsed as an identifier.

This setup does not allow a single + or - to match as an identifier, and allowing it by simply adding it to the first character class would create ambiguity. If we want to allow this, we could handle that as a separate branch:

```
token atom:sym<identifier> {
    | <[ a..z A..Z =*: ]> <[ a..z A..Z 0..9 _ =*: ]>*
    | <[+-]>
}
```

Parsing a string follows the pattern laid out in an earlier chapter: two quotes on the outside and the string contents in between. The string contents here are parsed by a separate rule to make the data extraction easier. String contents consist of "regular" permissible characters (everything except a backslash or a double quote), or an escape sequence: a backslash followed by a single, arbitrary character.

 You might have noticed that Perl 6 allows you to use the hyphen or minus character (-) inside an identifier, which I've used previously in the class name `S-Expression`, but not in the name of the regex `string_contents`. The reason for this inconsistency is that the `-` character harbors a potential ambiguity: `<-alpha-hexdigit>` could be a single negated named character class `alpha-hexdigit`, or match anything that is neither in `alpha` nor in `hexdigit`. Perl 6 disambiguates toward the first possibility, but avoiding this situation entirely still seems worthwhile.

Now that we have some grammar rules, we should test the core functionality:

```
use Test;

my %atoms =
    integer    => ('1', '01234', '-23', '+12'),
    identifier => ('abc', '=', '*_*'),
    string     => ('"', '"abc"', Q'"abc\"def"', Q'"\\"'),
    ;

my %not-atoms =
    identifier => ('', '_'),
    string     => ('', '"""', Q'"\"',),
    ;

for %atoms.keys.sort -> $atom {
    for %atoms{$atom}.list -> $test {
        ok S-Expression.parse($test,
                rule => "atom:sym<$atom>"),
            "Parsing '$test' as atom $atom";
    }
}
```

```
for %not-atoms.keys.sort -> $atom {
    for %not-atoms{$atom}.list -> $test {
        nok try {S-Expression.parse($test,
                rule => "atom:sym<$atom>") },
            "Not parsing '$test' as atom $atom";
    }
}

done-testing;
```

We set up a few input strings that an atom should match (`%atoms`) and some that they should not match (`%not-atoms`). Next, we iterate through these examples, and call `S-Expression.parse` with the example string and the corresponding rule, and wrap that in a call to `ok` for those strings we expect to be parsed, and `nok` (the negation of `ok`) for the strings we expect not to match.

For cases where we expect a parse failure, the test code wraps calls to `S-Expression.parse` in a `try { ... }` block. This is not strictly necessary, because we don't have advanced error reporting that would cause an exception to be thrown. However, such a thing might be added later, and we shouldn't punish future improvements by causing test failures.

Finally, instead of using the usual `plan NUMBER;` at the start, the test code uses `done-testing;` at the end. This alleviates the need to count tests yourself at the cost of reducing the ability to detect if we failed to run any tests we planned.

The tests all pass, so we can move on to the higher-level bits. To parse a list, we need a pair of parentheses containing a whitespace-delimited list of atoms:

```
token expression           {
    '(' ~ ')' [<atom>* % \s+]
}
```

Now we can also handle the case of nested lists, where an atom is a sublist:

```
token atom:sym<expression> { <expression>                }
```

Finally we specify that an S-Expression is a list of expressions, optionally with whitespace between, before, and after it:

```
token TOP                  { \s* [<expression> \s*] *    }
```

This is a first shot, and most likely doesn't work completely. You might have noticed that all regexes are `token` and not `rule`. This is because whitespace is significant (it delimits atoms) and using `rule` risks inadvertently using up this significant whitespace. Hence the grammar misses cases where whitespace should be allowed, but there is nothing yet in there to parse it. A good way to approach this is to write tests and run them:

```
my @tests = '()', '(abc)', ' (abc) ', '( abc )',
            '(1)', '(+1)', '(-1)', '( () ( ) )';

for @tests -> $t {
    ok S-Expression.parse($t), "can parse '$t'";
}
```

This fails two tests:

```
not ok 20 - can parse '( abc )'
not ok 24 - can parse '( () ( ) )'
```

We can fix this by adding more whitespace parsing to the `expression` token:

```
token expression          {
    \s* '(' ~ ')' [\s* <atom>* % \s+ \s*]
}
```

Here is the grammar in its entirety:

```
grammar S-Expression {
    token TOP                   { \s* [<expression> \s*] *     }
    token expression            {
        \s* '(' ~ ')' [\s* <atom>* % \s+ \s*]
    }
    proto token atom {*}
    token atom:sym<expression> { <expression>                 }
    token atom:sym<identifier> {
        <[ a..z A..Z =*:+- ]>
        <[ a..z A..Z 0..9 _ =*:+- ]>*
    }
    token atom:sym<integer>    { <[+-]>? <[0..9]>+            }
    token atom:sym<string>     { '"' ~ '"' <string_contents>  }
    token string_contents      { [ | <-[\\"]>+ | \\ . ]*      }
}
```

Data Extraction

The action class corresponding to the `S-Expression` grammar is mostly straightforward. There are just two things a bit out of the ordinary. The first is that it is useful to distinguish between the S-Expressions `(a)` and `("a")`, in other words between an identifier and a quoted string. To preserve that distinction, the action class returns an `Identifier` object from this class:

```
class Identifier { has $.str }
```

This can be done with such an action method:

```
method atom:sym<identifier>($/) {
    make Identifier.new(str => $/.Str);
}
```

The second thing to consider is what to do with backslash-escaped characters inside quoted strings. There seems to be a consensus among S-Expression implementations that \\ is a backlash and \" is a double quote character; however, the semantics of a backslash followed by another different character is not agreed upon. Here we will simply strip the escaping backslash, so \\ becomes \ and \a becomes a:

```
method string_contents($/) {
    make $/.Str.subst(:global, / \\ (.) /, -> $/ { $0 });
}
```

The action method for integers converts the match to an integer, and the other methods simply pass on the .made value of their captures:

```
class Identifier { has $.str }

class S-Actions {
    method TOP($/)                    {
        make $<expression>».made;
    }
    method expression($/)             {
        make $<atom>».made;
    }
    method atom:sym<expression>($/)   {
        make $<expression>.made;
    }
    method atom:sym<identifier>($/)   {
        make Identifier.new(str => $/.Str);
    }
    method atom:sym<integer>($/)      {
        make $/.Int;
    }
```

```
    method atom:sym<string>($/)      {
        make $<string_contents>.made;
    }
    method string_contents($/) {
        make $/.Str.subst(:global, / \\ (.) /, -> $/ { $0 });
    }
}
```

Let's write a short test for that:

```
my $m = S-Expression.parse(
    Q'((a "b") 23 "ab \\cd")',
    actions => S-Actions.new,
);
ok $m, 'Can parse S-Expression with action method';

is-deeply $m.made,
    [[[Identifier.new(str => "a"), "b"], 23, "ab \\cd"],],
    "correct data extracted";
```

This test passes, so it's time to wrap it into a subroutine that will become the public API for the S-Expression parser:

```
sub parse-s-expression(Str $input) {
    my $m = S-Expression.parse($input,
        actions => S-Actions.new);
    unless $m {
        die "Cannot parse S-Expression";
    }
    return $m.made;
}
```

For brevity, we won't improve error message generation for this grammar; any of the techniques discussed in Chapter 11 could be used here.

13.2 Mathematical Expressions and Operator Precedence Parsers

"If all you have is a hammer, everything looks like a nail[1]." If you work with a shiny new tool, such as regexes and grammars, you tend to overlook the option to use other tools to solve your problem.

In this spirit, I want to revisit the earlier example of parsing mathematical expressions, and this time do less in the grammar and more in the accompanying code. Instead of introducing a new parsing level for each precedence level, this iteration of the grammar ignores precedence altogether and simply parses a list of alternating terms and operators. The action method then uses a table of operator precedences to create the appropriate tree. This is called an *operator precedence parser*, also known as *OPP* for short.

There are two main advantages to this approach. It makes it much easier to insert a precedence level between two existing levels, and the flat structure of the grammar makes the parsing much faster.

A Simple Operator Precedence Parser

The core of the grammar is the part that parses an expression as a list of terms separated by infix operators[2]:

```
rule expression { <term> + % <infix> }
```

To make the list of infix operators extensible, it must be a proto token:

```
proto token infix   { * }
token infix:sym<*>  { <sym> }
```

[1]After Abraham Maslow, 1966. See https://en.wikipedia.org/wiki/Law_of_the_instrument.

[2]An infix operator is one that stands between two terms, like the asterisk * in `2 * 4`.

```
token infix:sym</>  { <sym> }
token infix:sym<+>  { <sym> }
token infix:sym<->  { <sym> }
token infix:sym<**> { <sym> }
```

In addition to the four usual operators, this list includes **, which is commonly used for exponentiation and which has tighter precedence than multiplication.

It won't surprise you that `term` includes numbers, but we can also parse subexpressions in parentheses in `term`, and function calls like `cos(0)` or `sqrt (4)`:

```
proto token term { * }
token term:sym<integer> {
    <[+-]>? <[0..9]>+
}
rule term:sym<parenthesized> {
    '(' ~  ')'<expression>
}
rule term:sym<function> {
    <name=.identifier> '(' ~ ')' <expression>
}
token identifier { <[a..z]>+ }
```

Time to tie it all together:

```
use Grammar::ErrorReporting;

grammar MathExpression does Grammar::ErrorReporting {
    rule TOP { <.ws> <expression> }
    rule expression { <term> + % <infix> }

    proto token infix   { * }
    token infix:sym<*>  { <sym> }
```

```
    token infix:sym</>  { <sym> }
    token infix:sym<+>  { <sym> }
    token infix:sym<->  { <sym> }
    token infix:sym<**> { <sym> }

    proto token term { * }
    token term:sym<integer> {
        <[+-]>? <[0..9]>+
    }
    rule term:sym<parenthesized> {
        '(' ~ ')' <expression>

    }
    rule term:sym<function> {
        <name=.identifier> '(' ~ ')' <expression>
    }
    token identifier { <[a..z]>+ }
}
```

`Grammar::ErrorReporting` is a module that provides an `error` method with fancy position output, but also installs a `FAILGOAL` method that triggers when a `~` expression fails to find its goal. If you use the `moritzlenz/perl6-regex-alpine` Docker image, this module is already included. Otherwise, you can install it by running

```
zef install Grammar::ErrorReporting
```

on the command line.

This grammar now produces a flat parse tree. For example, the input string `1 + 2 * 5` produces the following match object:

```
「1 + 2 * 5」
 expression => 「1 + 2 * 5」
  term => 「1」
```

```
infix => ⌜+⌟
 sym => ⌜+⌟
term => ⌜2⌟
infix => ⌜*⌟
 sym => ⌜*⌟
term => ⌜5⌟
```

Now we need to write the operator precedence parser that turns the flat array `[1, '+', 2, '*', 5]` into a tree grouped by precedence, so `[1, '+', [2, '*', 5]`, or even better `['+', 1, ['*', 2, 5]]`.

The crux is that whenever we look at an operator for the first time, we don't know if its adjacent operands actually belong to that operator (as happens in the case of `2 * 5`), or if the next operator has a higher precedence and steals the right operand.

We can solve this by first putting an operator and its left operand onto a stack. When we visit the next operator, and that second operator has a lower precedence than the first one, we can remove the previous operator and its operands from the stack, turn it into an expression, and then continue by putting the second operator on the stack. If the second operator has a higher precedence, we just put the second operator on the stack.

Once the end of the input list is reached, we can apply the reduction step until only one element is left on the stack. This is what the algorithm looks like in code:

```
class MathActions {
    method opp-parse(@tokens) {
        sub opp-reduce(@stack) {
            my ($term1, $op, $term2) = @stack.splice(*-3, 3);
            @stack.push([$op, $term1, $term2]);
        }
```

```
        my %prec = '+' => 1, '-' => 1,
                   '*' => 2, '/' => 2,
                   '**'  => 3;

        my @stack = @tokens[0];
        for @tokens[1..*] -> $op, $term {
            while @stack > 2
                    && %prec{$op} <= %prec{@stack[*-2]} {
                opp-reduce(@stack);
            }
            @stack.push($op, $term);
        }
        opp-reduce(@stack) while @stack > 1;
        return @stack[0];
    }
}
```

Method `opp-parse` expects a list of tokens, starting with a term, then an infix operator, and so on, with every other token being an operator.

The first element is a subroutine `opp-reduce`, which implements the reduction step: taking the last two terms and the operator between them from the stack, and adding them back as an array. Thus, if the stack has the elements `1, '+', 2, '*', 4`, a call to this function modifies it into `[1, '+', ['*', 2, 4]]`, and a second call further changes it to `['+', 1, ['*', 2, 4]]`.

`%prec` is a hash that maps each infix operator to a numerical precedence level, where higher numbers correspond to operators binding their arguments more tightly.

Subsequently, the `@stack` is initialized with the first term (from `@tokens[0]`), and then the loop runs over the rest of the tokens two elements at a time. As long as the stack has at least three elements (and thus at least one operator), and the current operator has lower precedence than the previous operator, the code calls the `opp-reduce` function. In any case, the current operator and term are added to the stack.

After all initial tokens have been visited, all that is left to do is to repeat the reduction step until only one element is left on the stack. This is then the parse tree we are interested in.

Note that by parsing parenthesized expressions in token `term`, we don't have to deal with parentheses as operators in the operator precedence parser. More traditional implementations only use lexical analysis and an operator precedence parser to check and handle parenthesis pairs.

The rest of the action methods are pretty straightforward in comparison:

```
method TOP($/) { make $<expression>.made }
method expression($/) {
    my @tokens = $/.caps.map({.value.made});
    make self.opp-parse(@tokens);
}
method infix:sym<*>($/) { make ~$<sym> }
method infix:sym</>($/) { make ~$<sym> }
method infix:sym<+>($/) { make ~$<sym> }
method infix:sym<->($/) { make ~$<sym> }
method infix:sym<**>($/) { make ~$<sym> }

method term:sym<integer>($/) { make $/.Int }
method term:sym<parenthesized>($/) {
    make $<expression>.made;
}
method term:sym<function>($/) {
    make [$<name>.made, $<expression>.made];
}
method identifier($/) { make $/.Str }
```

The only interesting bit here is the `expression` method. Remember that method `opp-parse` wants terms interleaved with operators, but the match of rule `expression` produces two arrays, `$<term>` and `$<infix>`. We could do the interleaving ourselves, but Perl 6 offers a better solution:

the method caps in class Match[3] returns all captures in the order that they appear in the string, which is exactly the interleaved order we want. The method caps returns the captures wrapped in Pair[4] objects, so we need to call .value to reach the match object, and then .made to access the Abstract Syntax Tree (AST) attached to the match. The map method does this to all elements that caps returns.

Again, we can wrap the calls to the grammar into a subroutine:

```
sub parse-math-expression(Str $input) {
    my $match = MathExpression.parse($input,
        actions => MathActions.new);
    die "Cannot parse input" unless $match;
    return $match.made;
}
```

and write some tests. Here is just a single one:

```
use Test;
plan 1;

is-deeply parse-math-expression('1 + 2 * 3**4 * 5 + 6'),
    ["+", ["+", 1, ["*", ["*", 2, ["**", 3, 4]], 5]], 6],
    'correct parse tree from nested expression';
```

A More Flexible Approach

The grammar and the operator precedence parser discussed in the previous section work, but the operator precedence parser is not very extensible. You can easily apply a role that adds another infix:sym<something> rule, but you can't define a precedence for it, because %prec is a lexical variable that cannot be accessed outside opp-parse.

[3] https://docs.perl6.org/type/Match.html#method_caps
[4] https://docs.perl6.org/type/Pair

A possible workaround is to create some kind of registry by which an extension could add an operator, but the problem goes deeper. There are several use cases for which we need more metadata about operators. One such case is if the parser should be able to handle different *associativities*. If an infix operator appears twice or more (or different operators with the same precedence occur in the same expression), the expression `1 OP 2 OP 3` could be interpreted as `(1 OP 2) OP 3` (left associative) or as `1 OP (2 OP 3)` (right associative). For the summation + and multiplication * operators, precedence does not matter. For difference - and division /, the typical associativity is left, but for exponentiation most programming languages assume right associativity. We need a way to store the associativity along with the precedence for an operator.

There are also other kinds of operators. *Postfix* operators stand after (post) the term. For instance, the *factorial* operator (!) stands for the product of all positive integers up to and including the number it follows. To illustrate: `6!` is `6 * 5 * 4 * 3 * 2 * 1` or 720. Now we need metainformation about the type of operator, and since postfix operators break the rule about alternating terms and operators[5], the operator precedence parser also needs a way to reliably distinguish terms from operators.

The common approach to meet these difficulties is to gather all the information about an operator into an object. Then, the operator precedence parser can do a type check to see if something is an operator and then ask it for its precedence, associativity, and type (infix, postfix, prefix). What's more, there no longer needs to be a central place where all the types are stored. When you create an extension role for the grammar, you also create an extension role for the action class, which in turn returns an object for the newly introduced operator.

[5]For example, the expression `3! + 2` has a term, followed by a postfix operator, followed by an infix operator.

The minimalistic approach (that does not yet handle associativity or other types of operators) goes like this:

```
class Operator {
    has Str $.symbol is required;
    has Numeric $.precedence is required;
    method new(Str $symbol, Numeric $precedence) {
        self.bless(:$symbol, :$precedence);
    }
}
```

The action methods for the operators then return `Operator` objects:

```
method infix:sym<+>($/)  { make Operator.new(~$<sym>, 1) }
method infix:sym<*>($/)  { make Operator.new(~$<sym>, 2) }
method infix:sym<**>($/) { make Operator.new(~$<sym>, 3) }
# more action methods go here
```

The method `opp-parse` now has access to the operators as objects and can thus use their precedence directly for comparison:

```
for @tokens[1..*] -> $op, $term {
    opp-reduce(@stack)
        while @stack >= 3
           && $op.precedence <= @stack[*-2].precedence;
    @stack.push($op, $term);
}
```

The comparison `$op.precedence <= @stack[*-2].precedence` is for left-associative operators; when you change the less-than-or-equal (`<=`) to a strict less-than (`<`), you get the behavior of right-associative operators. If you want to explore operator precedence parsers more, you could add an `associativity` attribute to class `Operator`, give the exponentiation operator right precedence, and change the comparison to take the associativity into account. The expression `2 ** 3 ** 4` should

come out as `['**' 2, ['**', 3, 4]]`. If you feel really adventurous, you might even try to add postfix operators to the grammar and the operator precedence parser.

13.3 Pythonesque, an Indentation-Based Language

Python, YAML, CoffeeScript, and other programming languages and data formats use indentation to encode the structure of the program or data. For instance, this Perl 6 program:

```
if 1 < 2 {
    say "Inside the branch";
}
say "outside the branch";
```

would look like this in Python 3:

```
if 1 < 2:
    print("Inside the branch")
print("outside the branch")
```

After a line that ends with colon (`:`), the next line must be more indented than the previous one. All further lines that share the same indentation as the second line belong to the block. If a line has fewer leading spaces than a previous block, this completes all open blocks with more leading spaces.

A full grammar for Python or YAML would be too much to fit into this book, so let's explore the concepts behind parsing an indentation-based language without most of the unrelated complexities.

The language we will parse here is a small subset of Python, and I've named it *Pythonesque*. It consists of expressions that contain numbers, variables, and common infix operators, as well as `if`-statements with indented bodies. The `if`-statements can be nested.

The following code is a small example program written in Pythonesque:

```
a =  1
if a:
    x = 1
    y = 2
    if x + 1 < 3:
          z = x + y
    a = z * 4
b = 5
```

The top level starts out with zero indentation, the first indentation level uses at least one space (here four), and the second level must be further indented than the second one (here ten spaces).

A Grammar for Pythonesque

We'll keep the expression parsing pretty minimalistic:

```
grammar Pythonesque {
    token ws { \h* }
    rule expression  { <term> + % <operator>  }
    token term       { <identifier> | <number>}
    token number     { \d+                    }
    token identifier { <:alpha> \w*           }
    token operator   {
         <[-+=<>*/,]>| '==' | '<=' | '>=' | '!='
    }
    # more rules go here later
}
```

In a real-world use case, we'd likely use proto tokens for both terms and operators; here we just use the bare minimum to parse expressions. If you want to enforce operator precedence, you could use the operator precedence parser from the previous case study.

Since whitespace is significant in Pythonesque, `ws` only matches horizontal space (which we allow between tokens; i.e., you can write `a=1` as `a = 1`). Nevertheless, we can only do that when we make sure that whitespace at the start of a line is matched by a different rule:

```
token line {
    ^^ ( \h* ) { self.handle_indentation($0) }
    <statement>  $$ \n*
}
proto token statement { * }
token statement:sym<expression> {
    <expression>
}
```

The core concept here is that a line of input consists of leading indentation (which might be zero in length, hence the `\h*`), then a statement, and finally the end-of-line `$$` followed by one or more newline characters. Note that the amount of leading whitespace must be checked. If it's the same as the previous line, all is fine. If it matches the indentation of an outer scope, the inner scopes must be finished. If it matches none of these, we've found an error. To do that check, we need an array of currently active (not yet finished) indentation levels:

```
token TOP {
    :my @*INDENTATION = (0,);
    <line>*
    $
}
```

```
method handle_indentation($match) {
    my $current = $match.Str.chars;
    my $last = @*INDENTATION[*-1];
    if $current > $last {
        die "Inconsistent indentation: expected "
          ~ "at most $last, got $current spaces";
    }
    elsif $current < $last {
        my $idx = @*INDENTATION.first(:k, $current);
        if defined $idx {
            for $idx + 1 .. @*INDENTATION.end {
                @*INDENTATION.pop;
            }
        }
        else {
            die "Unexpected indentation level: $current.";
        }
    }
}
```

Token `TOP` initializes a dynamic variable `@*INDENTATION` with a zero as its first element. As the next step, we'll implement adding elements to the end of the array, but for now we just assume that it contains the number of spaces of each indentation level. When we parse this input:

```
if a:
    x = 1
    if x + 1 < 3:
          z = x + y
```

the array `@*INDENTATION` contains the numbers 0, 4, and 10. If the current indentation is ten spaces (as determined by `$match.Str.chars`; a more sophisticated analysis might weight tabs differently than spaces, or forbid mixed tabs and spaces), nothing happens. If there are currently more leading spaces, this must be an error, because increased indentation is only allowed after an `if`-statement.

If the current indentation level is less than the last one recorded in `@*INDENTATION`, we have to search `@*INDENTATION` for the current value. This is what `my $idx = @*INDENTATION.first(:k, $current);` does. The method `first`[6] searches for the first array element that matches a certain criterion, here being equal to `$current`. With the `:k` modifier, it returns the index of that array element. If none was found, an undefined value comes back, and the `if defined $idx` check triggers, throwing an exception. If it is found, the current indentation is that of a previous scope. We then need to close all more deeply indented inner scopes, which is what `@*INDENTATION.pop` does for each such scope.

When we parse an `if`-statement, we know that the line after it must have more indentation than the current line, but not yet how much more. We could signal that by setting a second variable, or by pushing a special value onto `@*INDENTATION` that signals this fact. This *sentinel value*[7] could be a value that is not a valid indentation (like -1 or 0.5), or a separate type:

```
grammar Pythonesque {
    my class NEW_INDENTATION { }
    rule statement:sym<if> {
        'if'  <expression> ':'
        { @*INDENTATION.push(NEW_INDENTATION) }
    }
    # rest of the grammar here
}
```

[6] https://docs.perl6.org/type/List#routine_first
[7] https://en.wikipedia.org/wiki/Sentinel_value

Now we have to handle the case of NEW_INDENTATION in method handle_indentation:

```
method handle_indentation($match) {
    my $current = $match.Str.chars;
    my $last = @*INDENTATION[*-1];
    if $last ~~ NEW_INDENTATION {
        my $before = @*INDENTATION[*-2];
        if $current > $before {
            @*INDENTATION[*-1] = $current;
        }
         else {
             die "Inconsistent indentation: expected "
               ~ "more than $before, got $current spaces";
        }
    }
    elsif $current > $last {
        die "Inconsistent indentation: expected "
         ~ "at most $last, got $current spaces";
    }
    # rest of the method as before
}
```

This marks the point where our grammar can successfully parse the example given at the start of the section, and equally important, reject examples containing invalid indentation.

However, if you try to write action methods for this grammar, you'll soon notice that it is unreasonably hard to do, because the grammar does not communicate when a parsed scope begins or ends. Thus, the action methods need to look at the amount of whitespace at the start of a line, and then use @*INDENTATION (or maintain its own version thereof) to reconstruct a picture of the current and outer scopes. This is an unfortunate duplication of effort and also a rather high amount of coupling of the action methods to the grammar's internals.

A cleaner solution is to offer some kind of API that indicates to the action object when the grammar enters or leaves a scope. A nice trick to do this is to call rules that always match zero characters as a kind of marker:

```
token enter_scope { <?> }
token leave_scope { <?> }
token TOP {
    :my @*INDENTATION = (0,);
    <.enter_scope>
    <line>*
    $
    <.leave_scope>
}
```

The construst `<?>` in the tokens `enter_scope` and `leave_scope` is an empty zero-width assertion, which always successfully matches the empty string.

In addition to the calls from token `TOP`, we need to insert a call to `self.enter_scope()` into method `handle_indentation`, where it encounters the `NEW_INDENTATION` sentinel, and add a call `self.leave_scope()` when a scope is finished, thus accompanying the line that does `@*INDENTATION.pop()`:

```
method handle_indentation($match) {
    my $current = $match.Str.chars;
    my $last = @*INDENTATION[*-1];
    if $last ~~ NEW_INDENTATION {
        my $before = @*INDENTATION[*-2];
        if $current > $before {
            @*INDENTATION[*-1] = $current;
             self.enter_scope();
         }
    ...
```

```
    elsif $current < $last {
        my $idx = @*INDENTATION.first(:k, $current);
        if defined $idx {
            for $idx + 1 .. @*INDENTATION.end {
                @*INDENTATION.pop;
                self.leave_scope();
            }
        }
    ...
```

Now the action objects have a place to hook into that allows them to handle nesting of statements with reasonable effort.

Action Objects

When I first wrote action objects for Pythonesque, I made everything an array: an `if-statement` was an array whose first element was the string `"if"`, the second was the condition, and the third the body of the `if`-statement. Expressions were also arrays, so the second element was also an array. I quickly lost track of the generated AST.

To solve this issue I kept expressions as arrays, but moved operators, variables, and `if`-statements into separate classes:

```
class Operator {
    has Str $.action is required;
}
class Variable {
    has Str $.name is required;
}

class If {
    has $.condition;
    has @.block;
}
```

With this setup done, the action methods for the simple statements are not very surprising:

```
class Pythonesque::Actions {
    # more methods go here
    method statement:sym<if>($/) {
        make If.new(condition => $<expression>.made);
    }
    method statement:sym<expression>($/) {
        make $<expression>.made;
    }
    method expression($/) { make $/.caps».value».made.Array }
    method identifier($/) { make Variable.new(name => $/.Str) }
    method term($/)     { make $/.caps[0].value.made; }
    method number($/)   { make $/.Int }
    method operator($/) { make Operator.new(action => $/.Str) }
}
```

Note how the `If`-object in method `statement:sym<if>` is initialized with its condition, but not with its body. This is because it's not yet parsed at the time that `statement:sym<if>` runs.

To get the nesting of statements right, we need to maintain a stack of scopes. The bottom element (or index 0, when implemented as an array) is the outermost scope, and the top element (or index `*-1`) corresponds to the current scope.

When we leave a scope (except the outermost scope, also called the "mainline"), it belongs to an `if`-statement, and that `if`-statement is the last statement to have been parsed in the next-outermost scope. So we need to remove the current scope for the stack and set it as the body of the `if`-statement:

```
class Pythonesque::Actions {
    has @.scopes;
    method TOP($/)  {
```

```
        make @.scopes[0];
    }
    method line($/) {
         @.scopes[*-1].push($<statement>.made)
    }
    method enter_scope($/) { @.scopes.push([]) }
    method leave_scope($/) {
        if @.scopes > 1 {
            my $last = @.scopes.pop;
            @.scopes[*-1][*-1].block = $last.list;
        }
    }
    # rest of the class goes here as before
}
```

Testing this with various example strings shows that this mostly works, but fails in an interesting corner case: if the last line of the input has at least two levels of indentation, only the first level of indentation is present in the AST. For instance, this input:

```
if 1:
    if 2:
        x = 2
```

produces this AST:

```
[If.new(condition => [1], block => [])
```

which lacks the second `if`-statement and its body. Since the body of `If`-objects is populated in `leave_scope` calls, it stands to reason that `leave_scope` is not called. Adding debug `say` statements to the action methods `enter_scope` and `leave_scope` confirms this: for the input shown before, `enter_scope` is called three times (once for the mainline and once for each inner scope), but `leave_scope` is called only once.

leave_scope can be called from two places: from token TOP (which works), and from method handle_indentation. This makes the error clearer: handle_indentation only calls leave_scope when handle_indentation is called with an indentation less than the current level. But, when no unindented line is parsed at the end, no such call to handle_indentation happens. We can fix this bug by inserting such a call artificially at the end:

```
token TOP {
    :my @*INDENTATION = (0,);
    <.enter_scope>
    <line>*
    $

    # leave all open scopes:
    { self.handle_indentation('') }
    <.leave_scope>
}
```

With this fix in place, all tests pass, and we can consider this experiment a success.

You can view the full code, including tests (and some small enhancements that make testing of error conditions easier), on GitHub[8].

13.4 Summary

This chapter has shown you elements of real-world parsers that are used to process real-world data: recursion, operator precedence parser, and parsing a structure based on indentation levels. The last example also demonstrated tradeoffs in the interplay between regexes and code.

[8] https://github.com/apress/perl-6-regexes-and-grammars/blob/master/chapter-13-case-studies/pythonesque.p6

Throughout the course of this book, you have learned about the basics about Perl 6, the building blocks of Perl 6 regexes, how regexes and Perl 6 code interact, and captures, which offer a way to extract information from regex matches.

The chapters on regex mechanics and techniques have given you the background to understand how regexes work, and how you can use them to parse common data formats. Grammars allow you to structure and reuse regexes. The action object mechanism facilitates access to the resulting matches, and the discussion on error reporting shed light on what to do when a match fails.

Armed with the background from the previous chapters, and the applications from this chapter, you should be able to write your own grammars and parsers. I will leave you with a quote from Larry Wall, creator of Perl and inventor of Perl 6: *Have the appropriate amount of fun!*

Index

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

D

E

F

G, H

I, J

K

L

M

N

O

P

Q

R

S

T

U

V

W, X, Y

Z