

Handout 1

The purpose of a compiler is to transform a program a human can read and write into code machines can run as fast as possible. Developing a compiler is an old craft going back to 1952 with the first compiler written by Grace Hopper.¹ Why studying compilers nowadays? An interesting answer is given by John Regehr in his compiler blog:



“We can start off with a couple of observations about the role of compilers. First, hardware is getting weirder rather than getting clocked faster: almost all processors are multicores and it looks like there is increasing asymmetry in resources across cores. Processors come with vector units, crypto accelerators, bit twiddling instructions, and lots of features to make virtualization and concurrency work. We have DSPs, GPUs, big.little, and Xeon Phi. This is only scratching the surface. Second, we’re getting tired of low-level languages and their associated security disasters, we want to write new code, to whatever extent possible, in safer, higher-level languages. Compilers are caught right in the middle of these opposing trends: one of their main jobs is to help bridge the large and growing gap between increasingly high-level languages and increasingly wacky platforms. It’s effectively a perpetual employment act for solid compiler hackers.”

Given this, the goal of this module is to become a solid (beginner) compiler hacker and as part of the coursework to implement two small compilers for two very small programming languages.

The first part of the module is about the problem of text processing, which is needed for compilers, but also for dictionaries, DNA-data, spam-filters and so on. When looking for a particular string, say "foobar", in a large text we can use the Knuth-Morris-Pratt algorithm, which is currently the most efficient general string search algorithm. But often we do *not* just look for a particular string, but for string patterns. For example, in program code we need to identify what are the keywords (if, then, while, for, etc) and what are the identifiers (variable names). A pattern for identifiers could be stated as: they start with a letter, followed by zero or more letters, numbers and underscores.

Often we also face the problem that we are given a string, for example some user input, and we want to know whether it matches a particular pattern—is it an email address, for example. In this way we can exclude user input that would otherwise have nasty effects on our program (crashing it or making it go into an infinite loop, if not worse). This kind of “vetting” mechanism is also implemented in popular network security tools such as Snort and Zeek. They scan incoming network traffic for computer viruses or malicious packets. Similarly filtering out spam usually involves looking for some signature (essentially a string pattern). The point is that the fast Knuth-Morris-Pratt algorithm for strings is not good enough for such string *patterns*.



© Christian Urban, King’s College London, 2014, 2015, 2016, 2017, 2018, 2019, 2020, 2021, 2023

¹Who many years ago was invited on a talk show hosted by David Letterman.



Regular expressions help with conveniently specifying such patterns. The idea behind regular expressions is that they are a simple method for describing languages (or sets of strings)...at least languages we are interested in in Computer Science. For example there is no convenient regular expression for describing the English language short of enumerating all English words. But they seem useful for describing for example simple email addresses.² Consider the following regular expression

$$[a-z0-9_.-]^+ @ [a-z0-9_.-]^+ . [a-z.]{2,6} \quad (1)$$

where the first part, the user name, matches one or more lowercase letters (a-z), digits (0-9), underscores, dots and hyphens. The + at the end of the brackets ensures the “one or more”. Then comes the email @-sign, followed by the domain name which must be one or more lowercase letters, digits, underscores, dots or hyphens (but no underscores). Finally there must be a dot followed by the toplevel domain. This toplevel domain must be 2 to 6 lowercase letters including the dot. Example strings which follow this pattern are:

```
niceandsimple@example.org
very.common@example.co.uk
a.little.lengthy.but.fine@dept.example.ac.uk
other.email-with-dash@example.edu
```

But for example the following two do not

```
user@localhost
disposable.style.email.with+symbol@example.com
```

according to the regular expression we specified in line (1) above. Whether this is intended or not is a different question (the second email above is actually an acceptable email address according to the RFC 5322 standard for email addresses).

As mentioned above, identifiers, or variables, in program code are often required to satisfy the constraints that they start with a letter and then can be followed by zero or more letters or numbers and also can include underscores, but not as the first character. Such identifiers can be recognised with the regular expression

$$[a-zA-Z] [a-zA-Z0-9_]*$$

Possible identifiers that match this regular expression are x, foo, foo_bar_1, A_very_42_long_object_name, but not _i and also not 4you.

Many programming languages offer libraries that can be used to validate such strings against regular expressions. Also there are some common, and I am sure very familiar, ways of how to construct regular expressions. For example in Scala we have a library implementing the following regular expressions:

²See “8 Regular Expressions You Should Know” <http://goo.gl/5LoVX7>

<code>re*</code>	matches 0 or more occurrences of preceding expression
<code>re+</code>	matches 1 or more occurrences of preceding expression
<code>re?</code>	matches 0 or 1 occurrence of preceding expression
<code>re{n}</code>	matches exactly <i>n</i> number of occurrences of preceding expression
<code>re{n,m}</code>	matches at least <i>n</i> and at most <i>m</i> occurrences of the preceding expression
<code>[...]</code>	matches any single character inside the brackets
<code>[^...]</code>	matches any single character not inside the brackets
<code>...-...</code>	character ranges
<code>\d</code>	matches digits; equivalent to <code>[0-9]</code>
<code>.</code>	matches every character except newline
<code>(re)</code>	groups regular expressions and remembers matched text

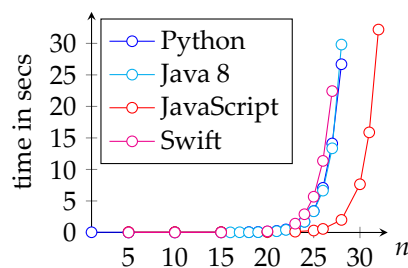
The syntax is pretty universal and can be found in many regular expression libraries. If you need a quick recap about regular expressions and how the match strings, here is a quick video: <https://youtu.be/bgWp9EI1MM>.

Why Study Regular Expressions?

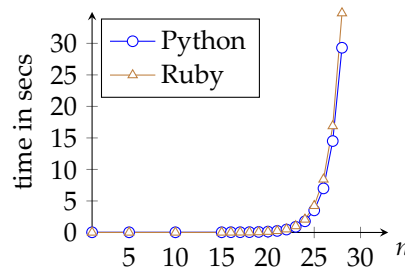
Regular expressions were introduced by Kleene in the 1950ies and they have been object of intense study since then. They are nowadays pretty much ubiquitous in Computer Science. There are many libraries implementing regular expressions. I am sure you have come across them before (remember the PRA or PEP modules?).

Why on earth then is there any interest in studying them again in depth in this module? Well, one answer is in the following two graphs about regular expression matching in Python, Ruby, JavaScript, Swift and Java (Version 8).

Graph: $(a^*)^*b$ and strings $\underbrace{a \dots a}_n$



Graph: $a?\{n\}a\{n\}$ and strings $\underbrace{a \dots a}_n$



The first graph shows that Python, JavaScript, Swift and Java 8 need approximately 30 seconds to find out that the regular expression $(a^*)^*b$ does not match strings of 28 as. Similarly, the second shows that Python and Ruby need approximately 29 seconds for finding out whether a string of 28 as matches the

regular expression `a{28}a{28}`.³ Admittedly, these regular expressions are carefully chosen to exhibit this exponential behaviour, but similar ones occur more often than one wants in “real life”. For example, on 20 July 2016 a similar regular expression brought the webpage [Stack Exchange](#) to its knees:

[http://stackstatus.tumblr.com/post/147710624694/
outage-postmortem-july-20-2016](http://stackstatus.tumblr.com/post/147710624694/outage-postmortem-july-20-2016)

I can also highly recommend a cool webtalk from an engineer from Stack Exchange on the same subject:

<https://vimeo.com/112065252>

A similar problem also occurred in the Atom editor:

<http://davidvgalbraith.com/how-i-fixed-atom/>

and also when somebody tried to match web-addresses using a relatively simple regular expression

<https://archive.ph/W50gx#selection-141.1-141.36>

Finally, on 2 July 2019 Cloudflare had a global outage because of a regular expression (they had no outage for the 6 years before). What happened is nicely explained in the blog:

[https://blog.cloudflare.com/
details-of-the-cloudflare-outage-on-july-2-2019](https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019)

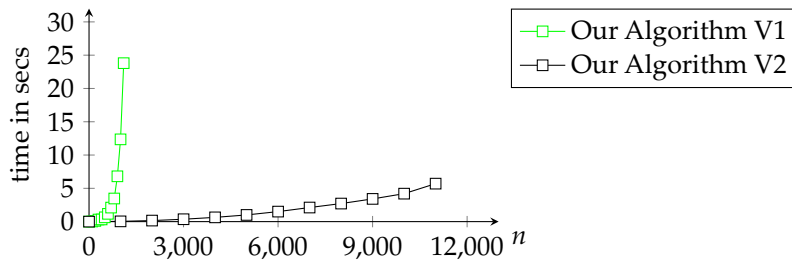
Such troublesome regular expressions are sometimes called *evil regular expressions* because they have the potential to make regular expression matching engines to topple over, like in Python, Ruby, JavaScript and Java 8. This “toppling over” is also sometimes called *catastrophic backtracking*. I have also seen the term *eternal matching* used for this. The problem with evil regular expressions and catastrophic backtracking is that they can have some serious consequences, for example, if you use them in your web-application. The reason is that hackers can look for these instances where the matching engine behaves badly and then mount a nice DoS-attack against your application. These attacks are already have their own name: *Regular Expression Denial of Service Attacks* (ReDoS).

It will be instructive to look behind the “scenes” to find out why Python and Ruby (and others) behave so badly when matching strings with evil regular expressions. But we will also look at a relatively simple algorithm that solves this problem much better than Python and Ruby do...actually it will be two versions of the algorithm: the first one will be able in the example `a{n}a{n}`

³In this example Ruby uses actually the slightly different regular expression `a?a?a?...a?a?aaa...aa`, where the `a?` and `a` each occur *n* times. More such test cases can be found at https://www.owasp.org/index.php/Regular_expression_Denial_of_Service_-_ReDoS.

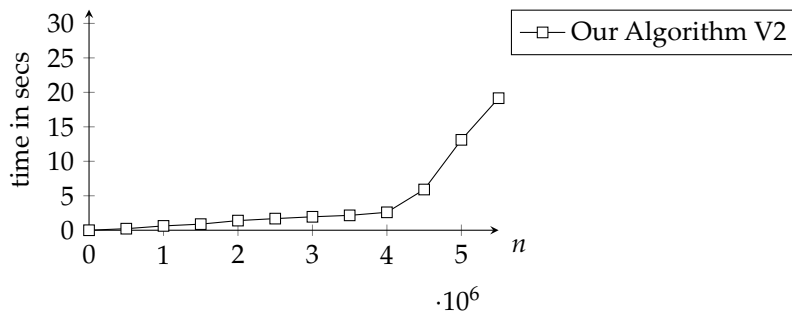
to process strings of approximately 1,100 as in 23 seconds, while the second version will even be able to process up to 11,000(!) in 5 seconds, see the graph below:

Graph: $a?\{n\}a\{n\}$ and strings $\underbrace{a \dots a}_n$



And in the case of the regular expression $(a^*)^*b$ and strings of as we will beat Java 8 by factor of approximately 1,000,000 (note the scale on the x-axis).

Graph: $(a^*)^*b$ and strings $\underbrace{a \dots a}_n$



You might have wondered above why I looked at the (now) old Java 8: the reason is that Java 9 and later versions are a bit better, but we will still beat them hands down with our regex matcher.

Basic Regular Expressions

The regular expressions shown earlier for Scala, we will in this module call *extended regular expressions*. The ones we will mainly study are *basic regular expressions*, which by convention we will just call *regular expressions*, if it is clear what we mean. The attraction of (basic) regular expressions is that many features of the extended ones are just syntactic sugar. (Basic) regular expressions are defined by the following grammar:

$r ::= \mathbf{0}$	null language
$\mathbf{1}$	empty string / "" / []
c	single character
$r_1 + r_2$	alternative / choice
$r_1 \cdot r_2$	sequence
r^*	star (zero or more)

Because we overload our notation, there are some subtleties you should be aware of. When regular expressions are referred to, then **0** (in bold font) does not stand for the number zero: rather it is a particular pattern that does not match any string. Similarly, in the context of regular expressions, **1** does not stand for the number one, but for a regular expression that matches the empty string. The letter c stands for any character from the alphabet at hand. Again in the context of regular expressions, it is a particular pattern that can match the specified character. You should also be careful with our overloading of the star: assuming you have read the handout about our basic mathematical notation, you will see that in the context of languages (sets of strings) the star stands for an operation on languages. Here r^* stands for a regular expression, which is different from the operation on sets is defined as

$$A^* \stackrel{\text{def}}{=} \bigcup_{0 \leq n} A^n$$

We will use parentheses to disambiguate regular expressions. Parentheses are not really part of a regular expression, and indeed we do not need them in our code because there the tree structure of regular expressions is always clear. But for writing them down in a more mathematical fashion, parentheses will be helpful. For example we will write $(r_1 + r_2)^*$, which is different from, say $r_1 + (r_2)^*$. This can be seen if we write regular expressions as trees:



The regular expression on the left means roughly zero or more times r_1 or r_2 , while the one on the right means r_1 , or zero or more times r_2 . This will turn out to be two different patterns, which match in general different strings. We should also write $(r_1 + r_2) + r_3$, which is different from the regular expression $r_1 + (r_2 + r_3)$, but in case of $+$ and \cdot we actually do not care about the order and just write $r_1 + r_2 + r_3$, or $r_1 \cdot r_2 \cdot r_3$, respectively. The reasons for this carelessness will become clear shortly.

In the literature you will often find that the choice $r_1 + r_2$ is written as $r_1 \mid r_2$ or $r_1 \parallel r_2$. Also, often our **0** and **1** are written \emptyset and ϵ , respectively. Following the convention in the literature, we will often omit the \cdot . This is to make some

concrete regular expressions more readable. For example the regular expression for email addresses shown in (1) would fully expanded look like

$$[\dots]^+ \cdot @ \cdot [\dots]^+ \cdot \cdot [\dots]\{2,6\}$$

which is much less readable than the regular expression in (1). Similarly for the regular expression that matches the string *hello* we should write

h · e · l · l · o

but often just write *hello*.

If you prefer to think in terms of the implementation of regular expressions in Scala, the constructors and classes relate as follows⁴

0	\mapsto	ZERO
1	\mapsto	ONE
<i>c</i>	\mapsto	CHAR(c)
$r_1 + r_2$	\mapsto	ALT(r1, r2)
$r_1 \cdot r_2$	\mapsto	SEQ(r1, r2)
r^*	\mapsto	STAR(r)

A source of confusion might arise from the fact that we use the term *basic regular expression* for the regular expressions used in “theory” and defined above, and *extended regular expression* for the ones used in “practice”, for example in Scala. If runtime is not an issue, then the latter can be seen as syntactic sugar of the former. For example we could replace

$$\begin{array}{ll} r^+ & \mapsto r \cdot r^* \\ r^? & \mapsto \mathbf{1} + r \\ \backslash d & \mapsto 0 + 1 + 2 + \dots + 9 \\ [a - z] & \mapsto a + b + \dots + z \end{array}$$

The Meaning of Regular Expressions

So far we have only considered informally what the *meaning* of a regular expression is. This is not good enough for specifications of what algorithms are supposed to do or which problems they are supposed to solve.

To define the meaning of a regular expression we will associate with every regular expression a language—a set of strings. This language contains all the strings the regular expression is supposed to match. To understand what is going on here it is crucial that you have read the handout about basic mathematical notations.

The *meaning of a regular expression* can be defined by a recursive function called L (for language), which is defined as follows

⁴More about Scala is in the handout about *A Crash-Course on Scala* from PEP.

$$\begin{array}{lll}
L(\mathbf{0}) & \stackrel{\text{def}}{=} & \{\} \\
L(\mathbf{1}) & \stackrel{\text{def}}{=} & \{\emptyset\} \\
L(c) & \stackrel{\text{def}}{=} & \{ "c" \} \quad \text{or equivalently } \stackrel{\text{def}}{=} \{ [c] \} \\
L(r_1 + r_2) & \stackrel{\text{def}}{=} & L(r_1) \cup L(r_2) \\
L(r_1 \cdot r_2) & \stackrel{\text{def}}{=} & L(r_1) @ L(r_2) \\
L(r^*) & \stackrel{\text{def}}{=} & (L(r))^\star
\end{array}$$

As a result we can now precisely state what the meaning, for example, of the regular expression $h \cdot e \cdot l \cdot l \cdot o$ is, namely

$$L(h \cdot e \cdot l \cdot l \cdot o) = \{ "hello" \}$$

This is expected because this regular expression is only supposed to match the “hello”-string. Similarly if we have the choice-regular-expression $a + b$, its meaning is

$$L(a + b) = \{ "a", "b" \}$$

You can now also see why we do not make a difference between the different regular expressions $(r_1 + r_2) + r_3$ and $r_1 + (r_2 + r_3)$...they are not the same regular expression, but they have the same meaning. For example you can do the following calculation which shows they have the same meaning:

$$\begin{aligned}
L((r_1 + r_2) + r_3) &= L(r_1 + r_2) \cup L(r_3) \\
&= L(r_1) \cup L(r_2) \cup L(r_3) \\
&= L(r_1) \cup L(r_2 + r_3) \\
&= L(r_1 + (r_2 + r_3))
\end{aligned}$$

That means both languages are the same. The point of the definition of L is that we can use it to precisely specify when a string s is matched by a regular expression r , namely if and only if $s \in L(r)$. In fact we will write a program `match` that takes a string s and a regular expression r as arguments and returns *yes*, if $s \in L(r)$ and *no*, if $s \notin L(r)$. We leave this for the next lecture.

There is one more feature of regular expressions that is worth mentioning here. Given some strings, there are in general many different regular expressions that can recognise these strings. This is obvious with the regular expression $a + b$ which can match the strings a and b . But also the regular expression $b + a$ would match the same strings. However, sometimes it is not so obvious whether two regular expressions match the same strings: for example do r^* and $\mathbf{1} + r \cdot r^*$ match the same strings? What about $\mathbf{0}^*$ and $\mathbf{1}^*$? This suggests the following relation between *equivalent regular expressions*:

$$r_1 \equiv r_2 \stackrel{\text{def}}{=} L(r_1) = L(r_2)$$

That means two regular expressions are said to be equivalent if they match the same set of strings. That is their meanings are the same. Therefore we do not really distinguish between the different regular expression $(r_1 + r_2) + r_3$ and $r_1 + (r_2 + r_3)$, because they are equivalent. I leave you to the question whether

$$0^* \equiv 1^*$$

holds or not? Such equivalences will be important for our matching algorithm, because we can use them to simplify regular expressions, which will mean we can speed up the calculations.

My Fascination for Regular Expressions

Up until a few years ago I was not really interested in regular expressions. They have been studied for the last 60 years (by smarter people than me)—surely nothing new can be found out about them. I even have the vague recollection that I did not quite understand them during my undergraduate study. If I remember correctly,⁵ I got utterly confused about **1** (which my lecturer wrote as ϵ) and the empty string (which he also wrote as ϵ).⁶ Since then, I have used regular expressions for implementing lexers and parsers as I have always been interested in all kinds of programming languages and compilers, which invariably need regular expressions in some form or shape.

To understand my fascination *nowadays* with regular expressions, you need to know that my main scientific interest for the last 17 years has been with theorem provers. I am a core developer of one of them.⁷ Theorem provers are systems in which you can formally reason about mathematical concepts, but also about programs. In this way theorem provers can help with the menacing problem of writing bug-free code. Theorem provers have proved already their value in a number of cases (even in terms of hard cash), but they are still clunky and difficult to use by average programmers.

Anyway, in about 2011 I came across the notion of *derivatives of regular expressions*. This notion allows one to do almost all calculations with regular expressions on the level of regular expressions, not needing any automata (you will see we only touch briefly on automata in lecture 3). Automata are usually the main object of study in formal language courses. The avoidance of automata is crucial for me because automata are graphs and it is rather difficult to reason about graphs in theorem provers. In contrast, reasoning about regular expressions is easy-peasy in theorem provers. Is this important? I think yes, because according to Kuklewicz nearly all POSIX-based regular expression matchers are buggy.⁸ With my PhD students Fahad Ausaf and Chengsong Tan, I proved the correctness for two such matchers that were proposed by Sulzmann and Lu in 2014.⁹ A variant of which you have already seen in PEP as CW3 and you

⁵That was really a long time ago.

⁶Obviously the lecturer must have been bad ;o)

⁷<http://isabelle.in.tum.de>

⁸http://www.haskell.org/haskellwiki/Regex_Posix

⁹<http://goo.gl/bz0eHp>

will see again in the CFL in the first two CWs. What we have not yet figured out that our matchers are universally fast, meaning they do not explode on any input. Hopefully we can also prove that the machine code(!) that implements our matchers efficiently is correct also. Writing programs in this way does not leave any room for any errors or bugs. How nice!

What also helped with my fascination with regular expressions is that we could indeed find out new things about them that have surprised some experts. Together with two colleagues from China, I was able to prove the Myhill-Nerode theorem by only using regular expressions and the notion of derivatives. Earlier versions of this theorem used always automata in the proof. Using this theorem we can show that regular languages are closed under complementation, something which Bill Gasarch in his Computational Complexity blog¹⁰ assumed can only be shown via automata. So even somebody who has written a 700+-page book¹¹ on regular expressions did not know better. Well, we showed it can also be done with regular expressions only.¹² What a feeling when you are an outsider to the subject!

To conclude: Despite my early ignorance about regular expressions, I find them now extremely interesting. They have practical importance (remember the shocking runtime of the regular expression matchers in Python, Ruby, Swift and Java in some instances and the problems in Stack Exchange and the Atom editor—even regex libraries in more modern programming languages, like Rust, have their problems). They are used in tools like Snort and Zeek in order to monitor network traffic. They have a beautiful mathematical theory behind them, which can be sometimes quite deep and which sometimes contains hidden snares. People who are not very familiar with the mathematical background of regular expressions get them consistently wrong (this is surprising given they are a supposed to be a core skill for computer scientists). The hope is that we can do better in the future—for example by proving that the algorithms actually satisfy their specification and that the corresponding implementations do not contain any bugs. We are close, but not yet quite there.

Notwithstanding my fascination, I am also happy to admit that regular expressions have their shortcomings. There are some well-known “theoretical” shortcomings, for example recognising strings of the form $a^n b^n$ is not possible with regular expressions. This means for example if we try to recognise whether parentheses are well-nested in an expression is impossible with (basic) regular expressions. I am not so bothered by these shortcomings. What I am bothered about is when regular expressions are in the way of practical programming. For example, it turns out that the regular expression for email addresses shown in (1) is hopelessly inadequate for recognising all of them (despite being touted as something every computer scientist should know about). The W3 Consortium (which standardises the Web) proposes to use the following, already more complicated regular expressions for email addresses:

¹⁰<http://goo.gl/2R11Fw>

¹¹<http://goo.gl/fD0eHx>

¹²<https://nms.kcl.ac.uk/christian.urban/Publications/rexp.pdf>

```
[a-zA-Z0-9.!\#$%&'*/=?^_`{|}~-]+@[a-zA-Z0-9-]+(?:\.[a-zA-Z0-9-]+)*
```

But they admit that by using this regular expression they wilfully violate the RFC 5322 standard, which specifies the syntax of email addresses. With their proposed regular expression they are too strict in some cases and too lax in others...not a good situation to be in. A regular expression that is claimed to be closer to the standard is shown in Figure 1. Whether this claim is true or not, I would not know—the only thing I can say about this regular expression is it is a monstrosity. However, this might actually be an argument against the RFC standard, rather than against regular expressions. A similar argument is made in

[http:
//elliott.land/post/its-impossible-to-validate-an-email-address](http://elliott.land/post/its-impossible-to-validate-an-email-address)

which explains some of the crazier parts of email addresses. Still it is good to know that some tasks in text processing just cannot be achieved by using regular expressions. But for what we want to use them (lexing) they are pretty good.

Finally there is a joke about regular expressions:

“Sometimes you have a programming problem and it seems like the best solution is to use regular expressions; now you have two problems.”

Figure 1: Nothing that can be said about this regular expression...except it is a monstrosity.