Regular Expressions *

Jens Lechtenbörger

Winter Term 2021/2022

Contents

1 Introduction

Welcome to a short tour of regular expressions.

Regular expressions fall into the wider topic of formal languages.

If you know the Chomsky hierarchy of formal languages or if you know how compilers for programming languages work, not much of the following may be new to you.

In any case, this introduction is meant to be self-contained, so not much prior knowledge should be necessary.

1.1 Regular Expressions

- Regular expressions (regexps) are formalism to define languages (permissible strings/words)
- Sample regexps use cases
  - **Validate** whether string conforms to format/pattern
    - Special case: **Define** tokens in programming languages
  - **Find** strings of specific **pattern** in text
  - **Replace** strings of specific **pattern** in text
  - **Split/elementize** strings

*This PDF document is an inferior version of an OER HTML page; free/libre Org mode source repository.

Regular expressions are a formalism to define formal languages.
For our purposes it is sufficient to understand a formal language as a set of strings.
So, some strings are part of a given language while others are not.
The notion of string here should be interpreted quite broadly.
For example, a string may or may not represent a syntactically valid program in the language Java.
The purpose of regular expressions is narrower, though.
Instead of entire programs, here the focus is on individual parts, sometimes called tokens.
For example, we can use regular expressions to define the set of valid variable names in a programming language or the set of valid telephone numbers in a data integration setting.
Further use cases are shown on the slide here.

1.2 Regular Expression as Tool

- Versatile tool
  - In most programming languages
  - In reasonable text editors
  - In UNIX command line tools, e.g., `grep`, `sed`
  - In data profiling tools (patterns for phone numbers, e-mail addresses, URIs, ...)
  - In data integration tools (matching of various formats to standardized representation)

In computing, regular expressions are widely spread.
Functionality for search and replace based on regular expressions exists in most programming languages, in better text editors, and in special-purpose tools.
In our class context, you find regular expressions in data profiling and data integration tools, which is why you may want to learn more about them.

1.3 Regexp Facts

- Regexp matching can be implemented via finite-state automata
  - Some implementations such as Rust's regex guarantee linear-time behavior
  - Others may come with backtracking and **exponential** run-time, breaking the Internet
- Regular expressions and (deterministic or non-deterministic) finite state automata specify precisely the same languages, namely the **regular languages**
  - There is no regexp to match correctly nested (arbitrarily deeply) parenthesized expressions
    - (E.g., "(1 + (2 * ((3 * ...)...)...)...)")
    - Reason: finite memory!
    - Regular languages vs context-free languages, Chomsky hierarchy

This slides mentions two facts about regular expressions, for which actually formal proofs exist.
First, every regular expression can be implemented via a finite-state automaton, which is a particular machine model for computation where only a finite amount of memory is available.

Second, the expressive power of regular expressions is limited as far as formal languages are concerned.

A famous example of a language that cannot be defined via a regular expression is that of properly nested parentheses.

Different classes of languages can be arranged in the so-called Chomsky hierarchy, about which you can learn more at Wikipedia or in textbooks on theoretical computer science or on compiler construction.

Those classes of languages include regular languages, defined by regular expressions, context-free languages, which include parenthesized expressions and are fundamental in the definition of programming languages, and more.

2 Background

2.1 Alphabets, Words, and Languages

- **Alphabet** = finite set of symbols, e.g.:
  - Latin characters
  - Digits
  - ASCII, UTF-8
  - Keyboard alphabet
- **Word** (over alphabet A) = String (over alphabet A) = finite sequence of symbols (over alphabet A)
  - ϵ denotes the **empty** word (no symbols)
- **Language** (over alphabet A) = set of words (over alphabet A)

To define a formal language, we start from an alphabet, which is just a finite set of symbols.

Typical examples of alphabets are shown here.

Given an alphabet, a word is a finite sequence of symbols.

The empty sequence or word is usually represented with epsilon.

This notion of word is somewhat counter-intuitive as a Java program can be perceived as a single word (while is consists of words at the same time).

Given an alphabet, a language is a set of words, for example the set of syntactically correct Java programs or the set of valid telephone numbers.

2.2 Operations on Words

- If v and w are words, then **vw** is a word, the **concatenation** of v and w
  - E.g., v = data, w = integration. Then vw = dataintegration
  - ϵ is the neutral element w.r.t. concatenation, i.e., $\epsilon w = w\epsilon = w$ for all words w
- If w is a word and k a non-negative integer then the **power** $\mathbf{w^k}$ is defined as follows:
  - $w^0 = \epsilon$
  - $w^k = w^{k-1}w$ for $k > 0$
- E.g.;

- $\text{data}^1 = \text{data}^0\text{data} = \epsilon\text{data} = \text{data}$
- $\text{data}^2 = \text{data}^1\text{data} = \text{datadata}$

A typical operation on words is the concatenation, which just means writing words one after the other, without any intermediate symbol.

By definition, concatenating a word **w** with the empty word, epsilon, does not change **w**.

Repeated concatenation is represented with powers and defined recursively as shown here.

Raising to the power zero always results in epsilon, while for a word **w** the result of **w** to the **k** is defined as **w** to the **k** minus one, concatenated with **w**.

Examples for that definition are shown here.

2.2.1 Quiz on Words

2.3 Operations on Languages

2.3.1 Let L and M be languages

- $L \cup M$ is usual **set union**
- $LM = \{ vw \mid v \in L \text{ and } w \in M \}$ (**concatenation**)
- $L^* = L^0 \cup L^1 \cup L^2 \cup L^3 \cup \ldots$ (**Kleene star/closure**)
  - Where $L^0 = \{ \epsilon \}$ and $L^{k+1} = L^k L$ $(k \geq 0)$
  - Zero or more concatenations of L with itself
- $L^+ = L^1 \cup L^2 \cup L^3 \cup \ldots$ (**positive closure**)
  - One or more concatenations of L with itself

For the definition of regular expressions, we need certain operations on languages.

First, as languages are sets of words, we can compute their set union.

Second, the concatenation of two languages results in the set of words that arise from concatenations of pairs of words from the original languages.

Third, the so-called Kleene star, denoted by L star, represents zero or more concatenations of some language.

Finally, L plus denotes one or more concatenations of L.

2.3.2 Language Operation Examples

- $L_1 = \{ \epsilon, ab, abb \}$, $L_2 = \{ b, ba \}$
- $L_1\ L_2 = ?$
- $L_2\ L_1 = ?$
- ${L_1}^3 = ?$
- ${L_2}^3 = ?$

This slide shows two languages and asks you to apply some operations.

2.3.3 Quiz on Language Operations

3 Regular Expressions

3.1 Regexp Definition

- Regexps and their languages (over alphabet A) **defined recursively** as follows
  1. ϵ is a regexp with $L(\epsilon) = \{\epsilon\}$
  2. a is a regexp with $L(a) = \{a\}$ for every symbol $a \in A$
  3. Let r and s be regexps with languages L(r) and L(s).
     - (r)|(s) is a regexp with $L((r)|(s)) = L(r) \cup L(s)$
       - Alternative
     - (r)(s) is a regexp with $L((r)(s)) = L(r)\ L(s)$
       - Concatenation
     - $(r)^*$ is a regexp with $L((r)^*) = (L(r))^*$
       - Kleene star
     - (r) is a regexp with $L((r)) = L(r)$

We are now ready to define regular expressions, which in turn define languages.

The empty word is a regular expression, whose language contains nothing but the empty word.

Also, each individual symbol is a regular expression, whose language contains nothing but the symbol as a word.

Recursively, four rules allow to construct more complex regular expressions from given regular expressions.

First, the vertical bar denotes an alternative, so the resulting language is the union of the original languages.

Second, the concatenation of regular expressions is again a regular expression, whose language is the concatenation of the original languages.

Third, the Kleene star can be applied to a regular expression, which results in the language obtained by applying the Kleene star to the original language.

Fourth, parentheses can be applied without changing the defined language.

3.2 Parentheses

- By definition, regexps contain many parentheses
- Reduction of amount of parentheses via precedence rules for operators
  - Kleene star > Concatenation > Alternative
  - E.g.: $((a)(a))|(((a)(b))^*(c)) = aa|(ab)^*c$

Here you see typical conventions to avoid parentheses based on precedence rules.

3.3 Regexp Matching

- Regexp E **matches** word w if $w \in L(E)$
- E_1 = data
  - $L(E_1) = \{$ data $\}$

  - E_1 matches data (but not integration)
- E_2 = (data)|(integration) = data|integration
  - $L(E_2)$ = { data, integration }
  - E_2 matches data and integration (but not dataintegration)
- $E_3 = (E_2)^*$
  - $L(E_3)$ = ?

A regular expression can be understood as pattern, which is matched against strings.
Precisely, a word matches regular expression E if the word is an element of the language defined by E.
You see two examples here and are instructed to think about a third one.

3.3.1 Quiz on RegExp Language

3.4 Major Regexp Shorthands

- E regular expression, n < m integers
  - E+ = EE* (at least one E)
  - E? = E|ϵ (E is optional)
  - E{n,m} = $E^n|E^{n+1}|\ldots|E^m$ (n to m repetitions of E)
- Alphabet A = { a1, a2, . . . , an }
  - . = (a1|a2|. . . |an) (dots represents any symbol of A)
  - $[a_{i1}a_{i2}\ldots a_{ik}] = (a_{i1}|a_{i2}|\ldots|a_{ik})$ (set of symbols)
  - $[\hat{}a_{i1}a_{i2}\ldots a_{ik}]$ (complemented set; all symbols except { a_{i1}, a_{i2}, . . . , a_{ik} })
- In practice, symbols of alphabets are ordered (e.g., a < c < z; 1 < 9)
  - a', a" ∈ A, a' < a". Then [a'-a"] = a'|. . . |a" (range of symbols)
    - [0-9] = 0|1|2|3|4|5|6|7|8|9

Various implementations of regular expressions offer abbreviations, some of which are listed here.

4 The Real World

4.1 Regexps in Practice

- Various standards
  - See https://en.wikipedia.org/wiki/Regular_expression#Standards
  - In particular, PCRE (Perl Compatible Regular Expressions)
    - Used in many tools and programming languages
    - All of the above: Concatenation, alternative (|), closure (*), positive closure (+), optional (?), sets ([. . .]), repetition ({n,m})

    - And more: Matching at beginning/end of word/line (^ and $), character classes (predefined expressions for words, numbers, whitespace, ...), look-ahead, look-behind, grouping and back-reference (parentheses and numbers) ...
  - E.g.:
    - data matches data integration and MIS and data warehousing
    - ^data matches data integration but not MIS and data warehousing; data$ matches neither
    - \d matches single digit, which is [0-9], which is (0|1|...|9)
    - [1-9]\d* matches positive integers without leading zeros
    - [hc]?at matches hat, cat, and at.

Different implementations provide different shorthands, some of which are recognized as standards.

Here, you see initial pointers.

You may want to try out the given examples, for which the next slide contains suggestions.

4.2 Learning Regexps

- Lots of free software tools to build regexps and verify their effects
- Some examples
  - `https://regexr.com/`
  - `M-x regexp-builder` in GNU Emacs
  - Next slide in Python
    - **Live** code execution, **editable**
      - Based on in-browser Python implementation (skulpt), not complete
  - Try out Jupyter notebooks (or Emacs IPython Notebook)
    - Web application for documents with live code, visualizations, documentation
      - Support for lots of languages
      - Conversations **with** and **about** data

Different resources are available to learn more about regular expressions.

I recommend that you try out some patterns in tools of your choice.

4.2.1 Regexps in Python

```
import re # See https://docs.python.org/3/library/re.html
m1 = re.search("^Data", "Data Integration")
print("Match: '{}'".format(m1.group(0))) # Group 0 is entire match
m2 = re.search("^Data", "MIS and Data Warehousing")
if m2 is None:
    print("No match.")
```

This Python code searches for the pattern “Data”, but only at the beginning of strings as the regular expression starts with the hat symbol.

Note that this presentation embeds a software called klipse, to enable live code execution. Thus, you can click into the code, edit it, and see results in the output section.

4.3 Regexp Example: E-Mail

- See https://www.regular-expressions.info/email.html
- `^[A-Z0-9.\_%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}$`
  - To be used with case-insensitive matching
- Design decisions
  - Match most real addresses but not all
    - Only ASCII allowed here
  - Also allow some invalid strings
    - Multiple subsequent dots are allowed
  - Don't care about domain names
    - Could explicitly specify alternatives for top-level domains (TLDs), e.g., `.museum` missing but `.asfg` allowed

Here you see a sample regular expression to match e-mail addresses, for variants of which you find discussions on the mentioned web page.

I suggest that you think about the regular expression and read the web page.

4.4 Patterns with Regexps

- Regexp for telephone numbers?
  - One expression matching both examples:
    - **+49-251-83-38150**
    - **+49 251 83 38151**
- Different regexp for alternative format?
  - **(0251) (83) 38158**

Please think about regular expressions for telephone numbers.

Sometimes different parts are separated by spaces, sometimes by hyphens, sometimes by parentheses.

I suggest that you construct regular expressions for the examples shown here, which we will revisit in class. Also, discuss the matching accuracy similarly to the discussion for e-mail addresses mentioned on the previous slides.

Note that the plus sign as well as parentheses have a special meaning, which implies that you need to escape them. Please figure out how to do so on your own.

5 Learning Objectives

- Explain what regexps are and where they are useful
- Give examples for what regexps can and cannot do
  - In general
  - Matching accuracy
- Read and write regexps for sample use cases
  - Enumerate their languages

License Information

Source files are available on GitLab (check out embedded submodules) under free licenses.