Regular Expressions

Simon Niklaus

1

preface

- regular expressions are another way to describe regular languages
- regular expressions are the prefered way when writing a program, in order to utilize a regular language
- it is just handier to write a short string instead of an entire finite state machine

regular expressions are defined recursively

- everything in our alphabet Σ is a regular language
- $R_1 \cup R_2$ is a regular expression
- $R_1 \circ R_2$ is a regular expression
- R^* is a regular expression
- ε is a regular expression
- $\emptyset$ is a regular expression
- (R) is a regular expression

- regular expressions are therefore per definition closed under union, concatenation and kleene / star

the operations have a tightness of binding

- $R^* \; > \; R_1 \circ R_2 > R_1 \cup R_2$
- kleene / star therefore has the highest priority, followed by concatenation and last but not least union
- if there are any doubts, i would suggest simply using parentheses

a few more notations

- $R_1 \circ R_2$ is sometimes simply written as $R_1 R_2$
- $R_1 \cup R_2$ is sometimes written as $R_1 | R_2$
- R^+ is equal to RR^*

just a simple practice to start with

- try to place parenthesis in between the following regular expressions
- $a \cup a b^* a b^*$
- $a a b \cup a a b \cup b^* a$
- $a + b \mid a b a$

just a hint on common pitfalls

- $R \cup \varepsilon \neq R$
- $R \circ \varepsilon = R$
- $R \cup \emptyset = R$
- $R \circ \emptyset = \emptyset \neq R$
- $\emptyset^* = \{\varepsilon\}$

every regular expression defines a regular language

- we again use $L(R)$ to refer to the language of a regular expression
- every regular expression can be converted into a finite state machine and vice versa
- DFA ↔ NFA ↔ REGEX
- you will see the proof for this later on in your graduate program

let us have some exercises again

For each of the following languages, give two strings that are members and two strings that are *not* members—a total of four strings for each part. Assume the alphabet $\Sigma = \{a,b\}$ in all parts.

a. a^*b^*

b. $a(ba)^*b$

c. $a^* \cup b^*$

d. $(aaa)^*$

e. $\Sigma^*a\Sigma^*b\Sigma^*a\Sigma^*$

f. $aba \cup bab$

g. $(\varepsilon \cup a)b$

h. $(a \cup ba \cup bb)\Sigma^*$

source: Sipser

let us have some exercises again

- give a regular expression for each of the following languages
- $\Sigma = \{a, b\}$
- $L = \{w \mid w = aba\}$
- $L = \{w \mid w = aba$ or $w = aaa\}$
- $L = \{w \mid w$ does contain aba in it$\}$
- $L = \{w \mid w$ contains at least three a's$\}$
- $L = \{w \mid w$ has an a at every odd position$\}$

let us have some exercises again

- design a DFA for $a^*b^*a^+$
- design a DFA for $a^*(bba^+)^*$
- define a regular expression, that describes email addresses in the form $\{w \mid w$ starts with an arbitrary nonzero number of a's, b's and c's and ends with $@pdx.edu\}$
- given $\{w \mid w$ contains an even number of a's and an odd number of b's and does not contain the substring ab, define a regular expression and design a DFA with no more than five states

12

Practical Regular Expressions

Simon Niklaus

regular expressions in programming languages

- are generally broader than what the theory defines
- there is not a particular standard, but every programming languages uses similar notations for regular expressions
- using regular expressions in a program can come in handy
  - to validate a certain input
  - to search or replace somethin within a string

regular expressions in python

- are being made available trough the re module
- we define them as a string, which is then being compiled into a different object
- this way, we can use a regular expression multiple times, without the computer having to figure out how to utilize it over and over again

the case of simply matching characters

- if we want to check, whether a string equals another fixed string by using a regular expression, we can do so as shown below

```
import re
regex = re.compile('abc')
result = regex.match('abc')
```

- some metacharacters have to be escaped with a leading \, because they are being used within the regular expressions themseleves

```
. ^ $ * + ? { } [ ] \ | ( )
```

escaping metacharacters

- parenthesise are for example being used within the definition of regular expressions
- so if we want to match parenthesise, we have to escape them as already mentioned – the example below gives you example of how this is being done

```
import re
regex = re.compile('\(abc\)')
result = regex.match('(abc)')
```

always use the online documentation

`regex.match`(*string*[, *pos*[, *endpos*]])

If zero or more characters at the *beginning* of *string* match this regular expression, return a corresponding *match object*. Return `None` if the string does not match the pattern; note that this is different from a zero-length match.

The optional *pos* and *endpos* parameters have the same meaning as for the `search()` method.

```
>>> pattern = re.compile("o")
>>> pattern.match("dog")      # No match as "o" is not at the start of "dog".
>>> pattern.match("dog", 1)   # Match as "o" is the 2nd character of "dog".
<_sre.SRE_Match object; span=(1, 2), match='o'>
```

If you want to locate a match anywhere in *string*, use `search()` instead (see also *search() vs. match()*).

- so match behaves slightly different to what we actually expected

source: https://docs.python.org/3/library/re.html#regular-expression-objects

always use the online documentation

- since the third version of python, they actually added another function

regex.**fullmatch**(*string*[, *pos*[, *endpos*]])

If the whole *string* matches this regular expression, return a corresponding *match object*. Return `None` if the string does not match the pattern; note that this is different from a zero-length match.

The optional *pos* and *endpos* parameters have the same meaning as for the `search()` method.

```
>>> pattern = re.compile("o[gh]")
>>> pattern.fullmatch("dog")      # No match as "o" is not at the start of "dog".
>>> pattern.fullmatch("ogre")     # No match as not the full string matches.
>>> pattern.fullmatch("doggie", 1, 3)   # Matches within given limits.
<_sre.SRE_Match object; span=(1, 3), match='og'>
```

New in version 3.4.

source: https://docs.python.org/3/library/re.html#regular-expression-objects

we can use this with our notion of regular expressions

- go ahead and create a new python file – or simply use the console if you prefer that
- we have already seen the expression $a^*b^*a^+$ earlier and use this expression within python

```
import re
regex = re.compile('a*b*a+')
result = regex.fullmatch('ba')
print(result)
```

- use this, in order to match different strings

some extensions to regular expressions

- brackets can be used, in order to require that one symbol out of the group has to occur
  - $[abc]$ refers to either an a or a b or a c
  - $[abcdefghijklmnopqrstuvwxyz]$ is not nice
- with the minus sign within brackets, a whole range of characters can be defined at once
  - $[a-z]$ refers to lowercase letters
  - $[A-Z]$ refers to uppercase letters
  - $[0-9]$ refers to digits

some extensions to regular expressions

- circumflexes can be used in combination with brackets, in order to require that something that is not within the group has to occur
  - $[\hat{}abc]$ refers to anything but a or a b or a c
  - $[\hat{}a - z]$ refers to anything but a lowercase letter
- there are several shorthands for common used groups
  - \d is equal to $[0 - 9]$
  - \s is equal to $[\ \ \backslash t \backslash r \backslash n]$
  - \w is equal to $[a - zA - Z0 - 9\_]$

some extensions to regular expressions

- a dot will refer to any character, except a new line character
  - $\cdot$ will refer to anything but $\backslash\mathrm{r}$ or $\backslash\mathrm{n}$
  - the excpetion of new lines is kind of special to pyton, but this behavior can be changed while compiling the regular expression
- a questionmark will define, that something occurs zero times or excalty once
  - $a?$ refers to a or ε
  - $[a-z]?$ refers to a lowercase letter or ε

let us practice what we have learned so far

- define an extended regular expression, that describes email addresses in the form $\{w \mid w$ starts with an arbitrary nonzero number of a's, b's and c's and ends with $@pdx.edu\}$
- define an extended regular expression, that is able to describe western names – id est only having a first and a last name out of the latin alphabet

utilizing groups

- everything inside a parenthesis is a group
- groups can be used, to figure out what the string that we matched initially contained
- the object that we are getting back can be used, in order to access this information

```
import re
regex = re.compile('(a*)(b*)(a+)')
result = regex.fullmatch('ba')
print(result)
print(result.group(0))
print(result.group(1))
print(result.group(2))
```

using search instead of match

- sometimes, we want to search for multiple occurences of a regular expression within a string
- there are several ways to do that, one of them is listed below
- regular expressions are then searched from left to right and it will always return the biggest match

```
import re
regex = re.compile('[a-z]+')
result = regex.findall('jlasdf alksjdfk
                        asdAflj lasd4fklj')
print(result)
```

backup slide

- if you see this, we were faster than i expected
- but do not worry, since i have this nice backup slide and a good looking potato