Regular Expression

COMP3220 – Principle of Programming Languages

Zhitao Gong

2016 Spring

Outline

Concatenation of Languages

The concatenation of two languages L_1 and L_2 is

$$L_1 L_2 = \{xy \mid x \in L_1 \text{ and } y \in L_2\}$$

Basically, concatenate every string from the first language with every string from the second. E.g., if $L_1 = \{a, b\}$ and $L_2 = \{c, d\}$, then $L_1 L_2 = \{ac, ad, bc, bd\}$.

Union of Languages

The union of two languages L_1 and L_2 is

$$L_1 \bigcup L_2 = \{x \mid x \in L_1 \text{ or } x \in L_2\}$$

Basically, it is a plain *set union*.

Kleene Closure of A Language

Given a language L, its Kleene closure L^* may be defined recursively.

- Base clause: $\epsilon \in L^*$
- Inductive clause: $xw \in L^*$ $\forall x \in L^*$ and $w \in L$
- Extremal clause: A string is in L if and only if it can be obtained from the above two clauses.

E.g., if $L = \{ab, cd\}$,
$L^* = \{\epsilon, ab, cd, abab, abcd, cdab, cdcd, ababab, \ldots\}$.

Outline

Recursive Definition

The regular language over an alphabet Σ may be defined *recursively*.

Basic clause $\emptyset$, $\{\epsilon\}$, $\{a\}$ $\forall a \in \Sigma$ are regular language.

Inductive clause If L_r and L_s are regular languages, then $L_r \bigcup L_s$, $L_r L_s$, L_r^* are regular languages.

Extremal clause A language is regular if and only if it can be obtained from the above two clauses.

Example

Given an alphabet $\Sigma = \{a, b\}$

- $\{a\}$ and $\{b\}$ are regular language.
- $\{a, b\}$ $(= \{a\} \bigcup \{b\})$, $\{ab\}$ $(= \{a\}\{b\})$ are regular language.
- $\{a\}^*$ $(= \{\epsilon, a, aa, aaa, \dots\})$ is regular language.
- Σ^* is also a regular language since $\{a, b\}$ is regular.

Outline

Recursive Definition

Theorem

A regular expression is a string r that denotes a regular language $L(r)$ over some alphabet Σ.

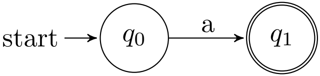
Basic clause $\varnothing$, ϵ and a are regular expressions denoting $\emptyset$, $\{\epsilon\}$ and $\{a\}$ $\forall a \in \Sigma$, respectively. They are called *atomic* regular expression.

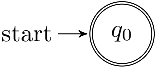
Inductive clause r and s are regular expressions denoting regular language L_r and L_s, then the follows are *compound* reguar expressions

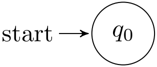
- $r + s$ denotes $L_r \bigcup L_s$
- rs denotes $L_r L_s$
- r^* denotes L_r^*

Extremal clause A string is a regular expression if and only if it can be obtained from above two clauses.

Base Clause NFAs

- $\forall a \in \Sigma, \mathcal{L}(a) = a$, 

- $\mathcal{L}(\epsilon) = \epsilon$, 

- $\mathcal{L}(\varnothing) = \emptyset$, 

Inductive Clause – Union

M_0, M_1 recognize $\mathcal{L}(R_0)$ and $\mathcal{L}(R_1)$ respectively.

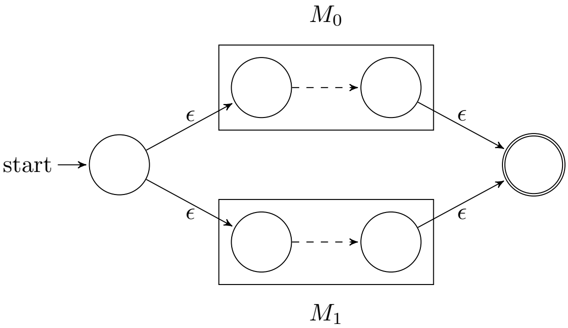


Figure: $R = R_0 + R_1$

Inductive Clause – Concatenation

M_0, M_1 recognize $\mathcal{L}(R_0)$ and $\mathcal{L}(R_1)$ respectively.

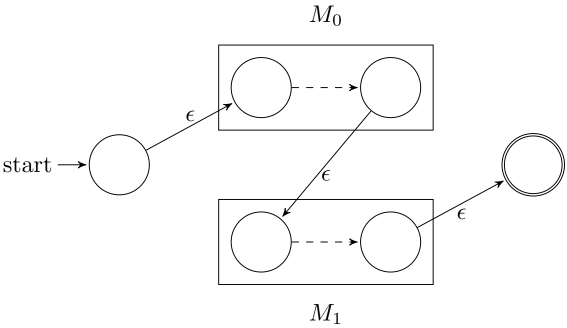


Figure: $R = R_0 R_1$

Inductive Clause – Kleene Closure

M_0 recognizes $\mathcal{L}(R_0)$

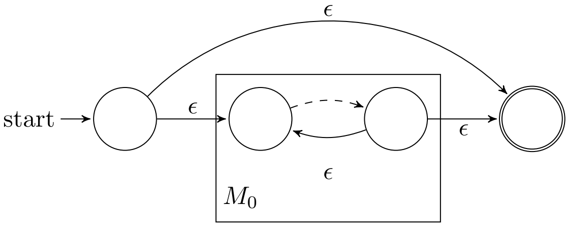


Figure: $R = R_0^*$

Equivalence of RE and FSA

1. We need to show for every RE, there is a FSA that accepts the language. Done with NFA-ϵ.
2. We need to show for every FSA, there is a RE denoting its language. Trickier and ignored for now.

Regular Expression Conventions

1. For precedence, $(R) > R^* > R_1 R_2 > R_1 + R_2$.
   - $(a + (b(c^*))))$ is the same as $a + bc^*$.
   - $ab^*c + d$ is the same as $((a(b^*))c) + d$.
2. Similar to string concatenation, concatenation of k regular expression r's is written as $r^k = \overbrace{rr \ldots r}^{k}$, specifically $r^0 = \varnothing$.

Regular Expression Examples

- $\mathcal{L}(01) = \{01\}$
- $\mathcal{L}(01 + 0) = \{01, 0\}$
- $\mathcal{L}(0(1 + 0)) = \{01, 00\}$
- $\mathcal{L}(0^*) = \{\epsilon, 0, 00, 000, \dots\}$
- $\mathcal{L}((0 + 01)^*(\epsilon + 1)) =$

Regular Expression Examples

- $\mathcal{L}(01) = \{01\}$
- $\mathcal{L}(01 + 0) = \{01, 0\}$
- $\mathcal{L}(0(1 + 0)) = \{01, 00\}$
- $\mathcal{L}(0^*) = \{\epsilon, 0, 00, 000, \dots\}$
- $\mathcal{L}((0 + 01)^*(\epsilon + 1)) =$all strings of 0's and 1's without two consecutive 1's.

Why It Matters

Theorem

For any regular expression R, there is FSA M such that

$$\mathcal{L}(R) = \mathcal{L}(M)$$

Why It Matters

Theorem

For any regular expression R, there is FSA M such that

$$\mathcal{L}(R) = \mathcal{L}(M)$$

Working solution for matching regular expressions against text.

- Convert regular expression to an NFA.
- (Optionally) convert NFA to a DFA.
- Run the text through the DFA.

Example Regular Expression → NFA

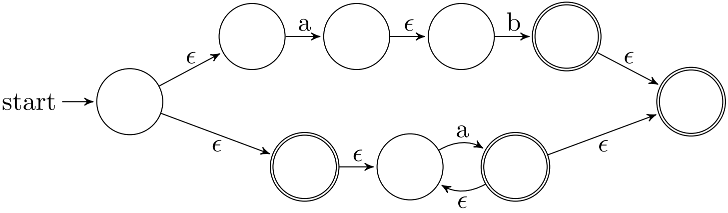
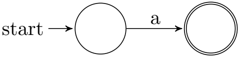
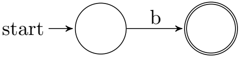
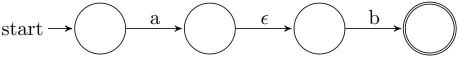
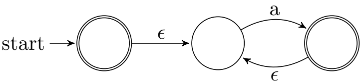


Figure: $L = ab \bigcup a^*$

- a: 

, b: 

- ab: 

- a^*: 

D/N-FA/Regular Expression Accept Regular Languages

$\mathcal{L}_{DFA}$ All languages accepted by DFAs

$\mathcal{L}_{NFA}$ All languages accepted by NFAs

$\mathcal{L}_{REG}$ All languages denoted by regular expressions

$\mathcal{L}_{R}$ Regular Languages

$$\mathcal{L}_{DFA} = \mathcal{L}_{NFA} = \mathcal{L}_{REG} = \mathcal{L}_{R}$$

Outline

Regular Expression Caveat

Some people, when confronted with a problem, think "I know I'll use regular expressions." Now they have two problems.
– Jamie Zawinski (flame war on alt.religion.emacs)

Writing RE is like writing a program.

- Need to understand programming model.
- Can be easier to write than read.
- Can be difficult to debug.

Are You Smart Enough

http://www.ex-parrot.com/~pdw/Mail-RFC822-Address.html

```
(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?
\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\]
\t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,
](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\
(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(
?[ \t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;:\\".\[\] \00
 \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\0
)*))*(?:,@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031
)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?
*:(?:(?:\r\n)?[ \t])*)?(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(
\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \0
]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:
?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".
:\r\n)?[ \t])*))*\>(?:(?:\r\n)?[ \t])*)|(?:[^()<>@,;:\\".\[\
[ \t]))*"(?:(?:\r\n)?[ \t])*)*:(?:(?:\r\n)?[ \t])*(?:(?:(?:
\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n
(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)*
```

RE Standard

IEEE POSIX standard has three set of compliance:

- *B*-asic RE (BRE)
- *E*-xtended RE (ERE) and
- ~~*S*-imple RE (SRE)~~ (deprecated).

ERE extends BRE

1. it adds ?, $+$, $|$, and
2. removes the need to escape *meta-characters* () and $\{\}$.

Notations

- Regular expression is abbreviated as *regex* or *RE* in the following discussions.
- We use `regex` to denote a regex, `text` a plain text string.
- `regex` $=$ `text` denotes that `regex` matches `text`.
- `regex` $\neq$ `text` denotes that `regex` does not match `text`.
- `regex1` $=$ `regex2` denotes that both regexes are equivalent.

Literal Characters

Single literal character, such as `a`, matches the *first occurrence* of the character.

- Given the string `Jack is a boy` it matches `a` after the `J`.
- The fact that `a` is in middle of the string does not matter in this case unless you use *word delimiter*.

Matching is *case sensitive* by default. E.g., `cat` $\neq$ `Cat`. It is configurable however.

Meta Characters

- escape, backslash `\`
- anchors, caret `^`, dollar sign `$`
- matches all (almost), period `.`
- alternation, vertical bar `|`
- question mark `?`, asterisk `*`, plus sign `+`
- grouping, opening and closing parenthesis `(`, `)`
- character set, opening square bracket `[`, `]` and
- repetition, opening curly brace `{`, `}`.

Meta Character Match

To match meta characters, you need to *escape* them with a `\`.

- to match `1+1=2`, the correct regex is `1\+1=2`.
- `1+1=2` is a valid regex, matching `111=2`, `1111=2`, `123+111=234`, etc.

Non-Printable Characters

Conventional formatter, `\t` (tab, 0x09), `\r` (carriage return, 0x0D), `\n` (line feed, 0x0A) and etc. The followings are inconsistent.

- Control sequences, `\cA` = Control+A.
- Unicode characters, `\u20AC` (or `\x{20AC}`) = `€`.

Character Class

With a "character class", also called "character set", you can tell the regex engine to match *only one* out of several characters.

- `[ae]` = `a` or `e`. E.g., `gr[ae]y` = `gray` or `grey`. The order in the bracket *does not* matter.
- Hyphen specifies a range.
  - `[0-9]` = *single* digit between 0 and 9
  - `[a-z]` = *single* character between a and z.
  - `0[xX][A-Fa-f0-9]+` = C-style hexadecimal number.

Negated Character Class

A caret `^` after the opening square bracket *negates* the character class. The result is that the character class matches any character that is *not* in the character class.

- `[^0-9\r\n]` = any single character that is *not* a digit or a line break.
- `q[^u]` does *not* mean "a `q` not followed by a `u`". Instead it means "a `q` followed by a character that is not a `u`". E.g., `q[^u]` ≠ `q`, `q[^u]` = `q `, note that space after `q`.

Meta Character inside Character Class

Inside character class, only `]`, `^`, `\` and `-` are special.

- `[\]]` = `]`, `[\^]` = `^`, etc.
- `[\+]` = `[+]` = `+`.
- `[0^]` = `0` or `^`
- The hyphen can be included a) right after the opening bracket, or b) right before the closing bracket, or c) right after the negating caret. E.g., `[-x]` = `[x-]` = `x` or `-`.

Shorthand Character Class

Some character classes are often used.

- digit `\d` = `[0-9]`
- word character `\w` = `[A-Za-z0-9_]`
- whitespace character `\s` = `[ \t\r\n\f]`.
- Shorthand may also be used inside square brackets. E.g., `[\s\d]` = a whitespace or a digit.

Negated Shorthand Character Class

Most shorthand character classes have negated shorthand.

- non-digit `\D` $=$ `[^\d]`
- non-whitespace `\S` $=$ `[^\s]`
- non-word character `\W` $=$ `[^\w]`
- `[\S\D]` $\neq$ `[^\s\d]`

Dot

Dot `.` matches any character *but line breaks*.
Given the four date formats

- `mm/dd/yy`
- `mm-dd-yy`
- `mm.dd.yy`
- `mm dd yy`
- A quick solution is `\d\d.\d\d.\d\d`.

Dot

Dot `.` matches any character *but line breaks*.
Given the four date formats

- `mm/dd/yy`
- `mm-dd-yy`
- `mm.dd.yy`
- `mm dd yy`
- A quick solution is `\d\d.\d\d.\d\d`. However `02512703` is also a match.

Dot

Dot `.` matches any character *but line breaks*.
Given the four date formats

- `mm/dd/yy`
- `mm-dd-yy`
- `mm.dd.yy`
- `mm dd yy`
- A quick solution is `\d\d.\d\d.\d\d`. However `02512703` is also a match.
- Modified version is `\d\d[- /.]\d\d[- /.]\d\d`.

Dot

Dot `.` matches any character *but line breaks*.
Given the four date formats

- `mm/dd/yy`
- `mm-dd-yy`
- `mm.dd.yy`
- `mm dd yy`
- A quick solution is `\d\d.\d\d.\d\d`. However `02512703` is also a match.
- Modified version is `\d\d[- /.]\d\d[- /.]\d\d`. However it matches `99/99/99`.

Anchors

Anchors match positions instead of characters.

- `^` = the position before the first character in the string. E.g., `^a` = `a` in `abc`, `^b` $\neq$ `b` in `abc`.
- `$` = the position right after the last character in the string. E.g., `c$` = `c` in `abc`, `a$` $\neq$ `a` in `abc`.
- In *multi-line mode*, `^` and `$` matches the start and end of each line respectively.

Word Boundary

`\b` matches a position called "word boundary". It allows whole-words-only search. Its negation is `\B`.

- Before the first character in the string, if the first character is a word character.
- After the last character in the string, if the last character is a word character.
- Between two characters in the string, where one is a word character and the other is not a word character.

E.g., `\b4\b` $\neq$ `4` in `44 sheets of a4`.

Alternation

Alternation `|` is used to match a single regular expression out of several possible ones.

- `cat|dog` = `cat` or `dog`,
- `cat|dog|mouse|fish` = `cat` or `dog` or `mouse` or `fish`

The `|` has the *lowest precedence* of all regex operators.

- `\bcat|dog\b` = `cat` in `Hi cats!!` or `dog` in `hotdog!`.
- `\b(cat|dog)\b` matches nothing in `Hi cats!!` or `hotdog!`.

Optional Item

The question mark `?` the preceding token optional. The question mark is called a *quantifier*.

- `colou?r` = `color` and `colour`.
- `Nov(ember)?` = `Nov` and `November`.
- `Feb 23(rd)?` = `Feb 23rd` in `Today is Feb 23rd` since `?` is *greedy*, discussed later.

Repetition

Two more quantifiers.

- `*` tells the regex engine to attempt to match the preceding token zero or more times.
- `+` tells the regex engine to attempt to match the preceding token once or more.

E.g., `<[A-Za-z][A-Za-z0-9]*>` = an HTML tag without attributes, i.e., `<B>`, `<html>`, etc.

Limiting Repetion

An additional quantifier allows you to specify how many times a token can be repeated, the syntax is `{min, max}`.

- `0?` = `0{0,1}`
- `0+` = `0{1,}`
- `0*` = `0{0,}`
- `0{3}` = `000`

Greedy and Lazy Quantifiers

`?`, `*`, `+`, `{min,max}` are all greedy, they tell the regex engine to match as many as possible unless it results in a match fail.

- `<.+>` = `<em>emph</em>` in `a <em>emph</em> test`, not `<em>` or `</em>`.

Putting a `?` after these greedy quantifiers makes them *lazy* (or *ungreedy*, *reluctant*).

- `<.+?>` = `<em>` and `</em>` in `a <em>emph</em> test`.

Atomic Grouping

An atomic group is a group that, when the regex engine exits from it, automatically throws away all backtracking positions remembered by any tokens inside the group.

- Atomic groups are non-capturing.
- The syntax is `(?>group)`

- `a(bc|b)c` = `abcc` and `abc`.
- `a(?>bc|b)c` = `abcc`
- `a(?>bc|b)c` ≠ `abc`
- `\b(integer|insert|in)\b` ≠ `integers`. Optimized version is `\b(?>integer|insert|in)\b`

Possessive Quantifiers

Possessive quantifiers are a syntax sugar to place an atomic group around a single quantifier. `X*+` $=$ `(?>X*)`.

- `".*"` $=$ `"abc"` in `"abc"x`
- `".*+"` $\neq$ `"abc"` in `"abc"x`

Grouping and Capturing

- Only parentheses, `(` and `)` can be used for grouping, which allows you to apply a quantifier to the entire group or to restrict alternation to part of the regex.
  - `cat|dog` = `cat` and `dog` in `rains cats and dogs`.
  - `ca(t|d)og` = `catog` or `cadog`.
- `( )` not only creates groups, but *numbered capturing groups*.
  - `get(value)\1` = `getvaluevalue`.
- To disable capturing, put `?:` right after opening `(`.
  - `get(?:value)\1` results in error.

Backreference

Previously, `get(value)\1` $=$ `getvaluevalue`.

- `\1` is called backreference, matching the *exact same text* that was matched by the *first capturing group*.
- `\2` matches the *second capturing group*, etc.
- Backreference may be reused many times.

Backreference stores the last match.

- `([abc]+)=\1` $=$ `cab=cab`
- `([abc])+=\1` $\neq$ `cab=cab`
- `([abc])+=\1` $=$ `cab=b`

`\b(\w+)\s+\1\b` $=$ doubled word, e.g, `the the`.

Backreference to Failed Groups

- `(q?)b\1` $=$ `b` `q?` matches nothing, so does \1.
- `(q)?b\1` $\neq$ `b` `(q)?` proceeds to `b`, then \1 which fails by mimicking the result of the group.
- `(q)?b\1?` $=$ `b`

Regex Engine is *Eager*

Regex engine stops searching as soon as it finds a valid match.

- `Get|GetValue` = `Get` in `GetValue`.
- `GetValue|Get` = `GetValue` in `GetValue`.
- `\b(Get|GetValue)\b` = `GetValue` in `GetValue`
- `Get(Value)?` = `GetValue` in `GetValue` since `?` is *greedy*.

Advanced Topics

- Lookahead and lookaround
- Conditionals
- Recursiion
- Subroutine
- etc.

Outline

Regular Expression and Regular Language

1. RE is used to *denote* regular language.
2. RE may be implemented using NFA.
3. Without knowing regular language, RE is also useful
4. RE is crazy.