Regular Expressions

Object Oriented Programming

https://softeng.polito.it/courses/09CBI

SoftEng
http://softeng.polito.it

Version 3.0.3 - May 2021

Regular Expressions

- Represent a simple and efficient way to describe a sequence of characters
- They can be used to:
  - generate a conforming sequence of chars
  - recognize a sequence of chars as conforming with the RE
- The ability to recognize a valid sequence is fundamental in text processing.

Regular expressions

- A RE describes a sequence of characters and use a set of operators:

" \ [ ] ^ - ? . * + | ( ) $ / { } % < >

- Letters and numbers in the input text are described by themselves
    - E.g., v and x represent the same characters in the input text E.g.,
- Operators and special chars must be preceded by the quotation character \
    - E.g., \+ and \\ represent the character + \ in the input text

Sequence, optional, alternative

- Concatenation is applied by placing REs one after the other
  - `val1` represents ‘**`v`**’ ‘**`a`**’ ‘**`l`**’ ‘**`1`**’ in the input text
- The operator `?` makes the preceding expression optional:
  - `ab?c` represents both **`ac`** and **`abc`**.
- The binary operator `|` represents an alternative between two expressions:
  - `ab|cd` represents both the sequence **`ab`** and the sequence **`cd`**.

Character set

- Character sets are described using `[]`:
  - `[0123456789]` represents any numeric cipher
- In a set, the symbol `-` indicates a range of characters:
  - `[0-9]` represents any numeric character
- To include `-` in the set, it must be first or last char:
  - `[-+0-9]` represents a number in the input text.
- When a set begins with `^`, the characters are excluded:
  - `[^0-9]` represents any nonnumeric character

Repetitions

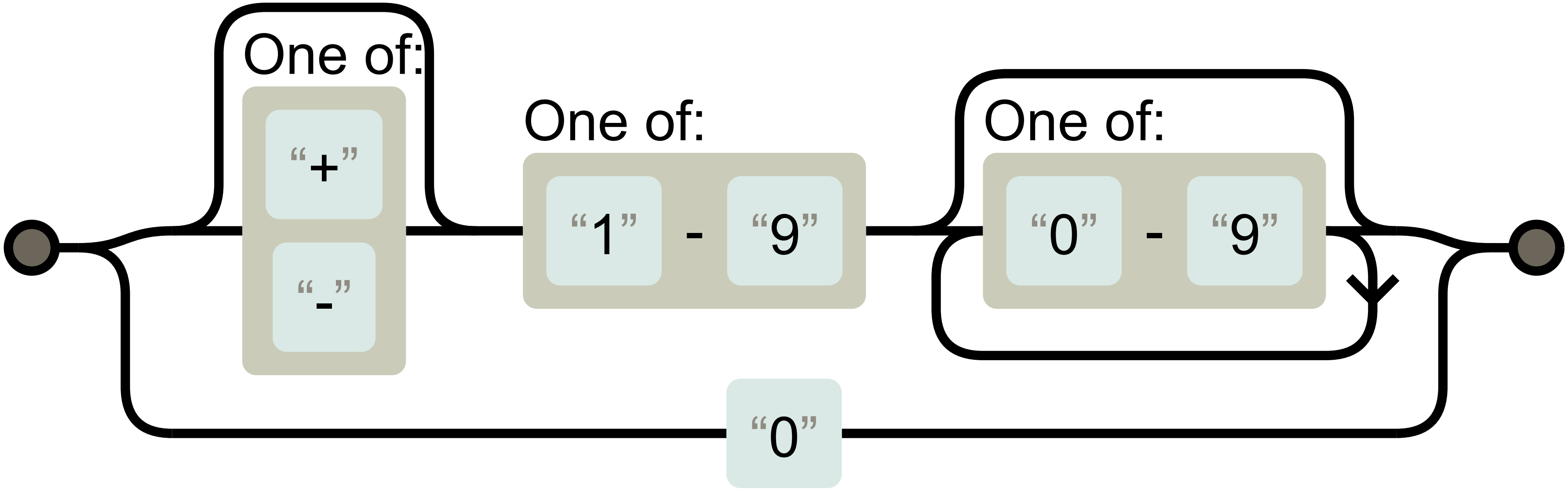
- The operator `+` indicates the preceding expression can be repeated 1 or more times:
  - `ab+c` represents sequences starting by `a`, ending in `c`, and containing at least one `b`.
- The operator `*` indicates the preceding expression can be repeated 0 or more times:
  - `ab*c` represents sequences starting by `a`, ending in `c`, and containing any number of `b`.
- The operator `{l,h}` matches from *l* to *h* repetitions of the preceding expression

Examples of RE

- Positive integer number
  - `[0-9]+`
- Positive integer number w/o leading 0
  - `[1-9][0-9]*|0`
- Integer number with optional sign
  - `[+-]?[1-9][0-9]*|0`

Railroad diagram

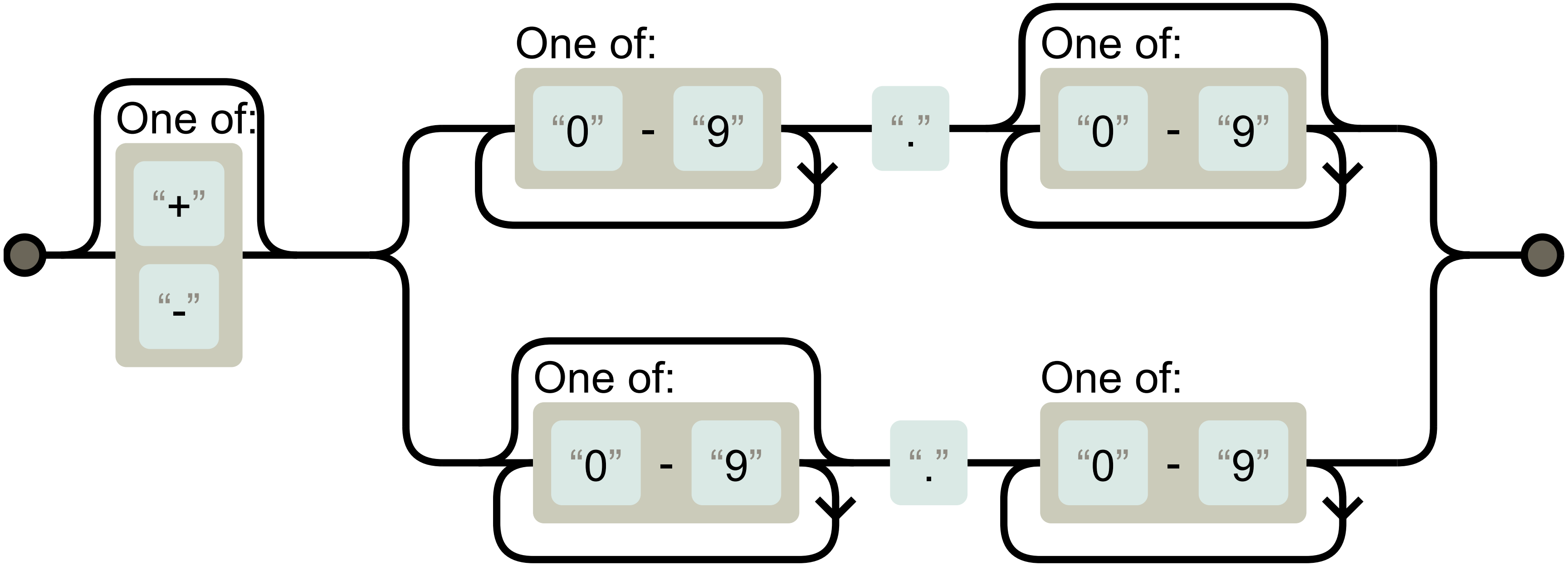
- Integer number with optional sign
  - `[+-]?[1-9][0-9]*|0`



Generated with: http://regexper.com

Examples of RE

- Floating point number
  - `[+-]?(([0-9]+\.[0-9]*)|`
    `([0-9]*\.[0-9]+))`

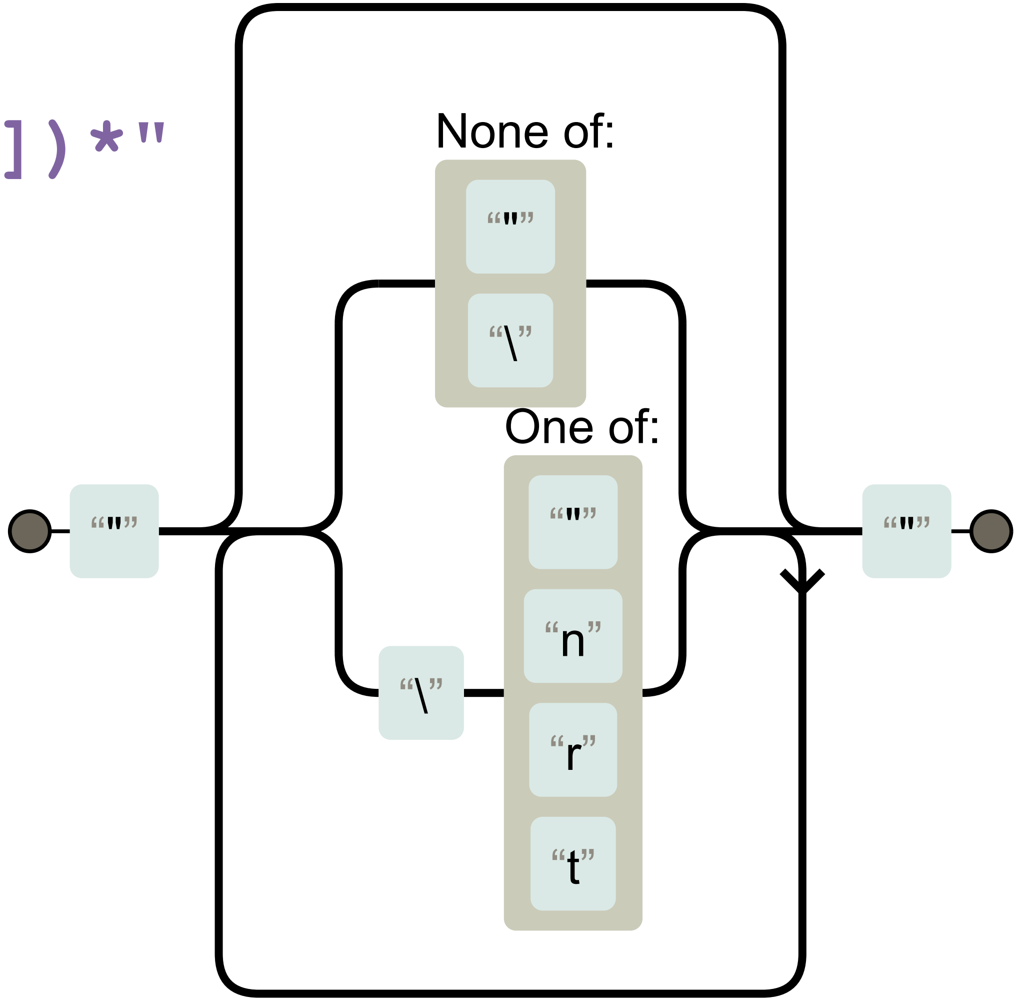


Generated with: http://regexper.com

Examples of RE

- String

```
"([^"\\]|\\["nrt])*"
```



Generated with: http://regexper.com

Special characters

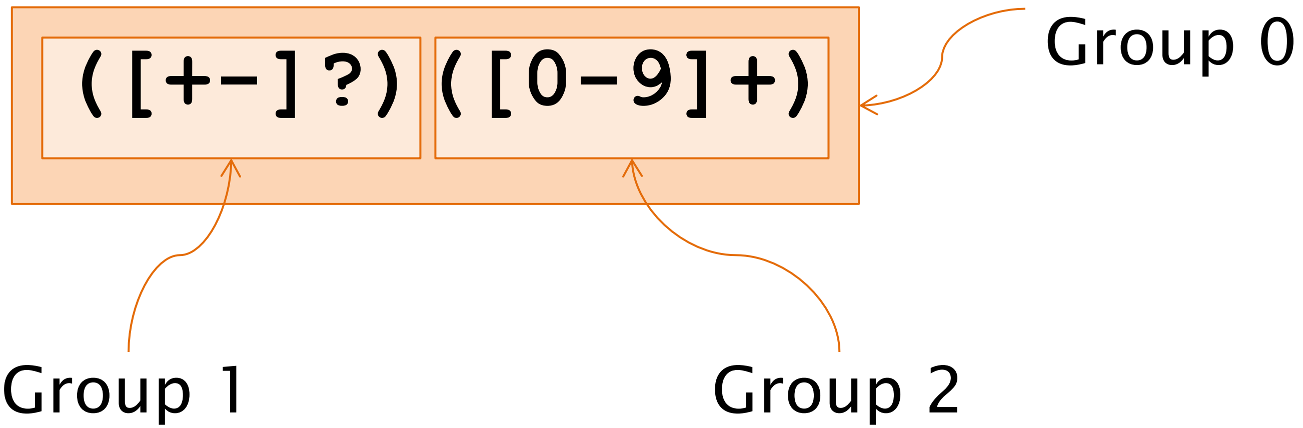
- Any character except new-line is described by a dot: `.`
- The new-line is represented by `\n`
- Any white space is described by `\s`
- Any digit is described by `\d`,
  - i.e. `[0-9]`
- Any word char is described by `\w`,
  - i.e. `[A-Za-z0-9_]`
- The beginning of text is `^`
- The end of text is `$`

Priority

- The order of priority is
  - escape \
  - character sets `[]`
  - repetition ? `+ * { }`
  - concatenation
  - alternative `|`
- The round parentheses, `(` and `)`, define a grouping and change priorities
  - `(ab|cd+)?ef` represents such sequences as **`ef`**, **`abef`**, **`cdddef`**, etc.

Capture groups

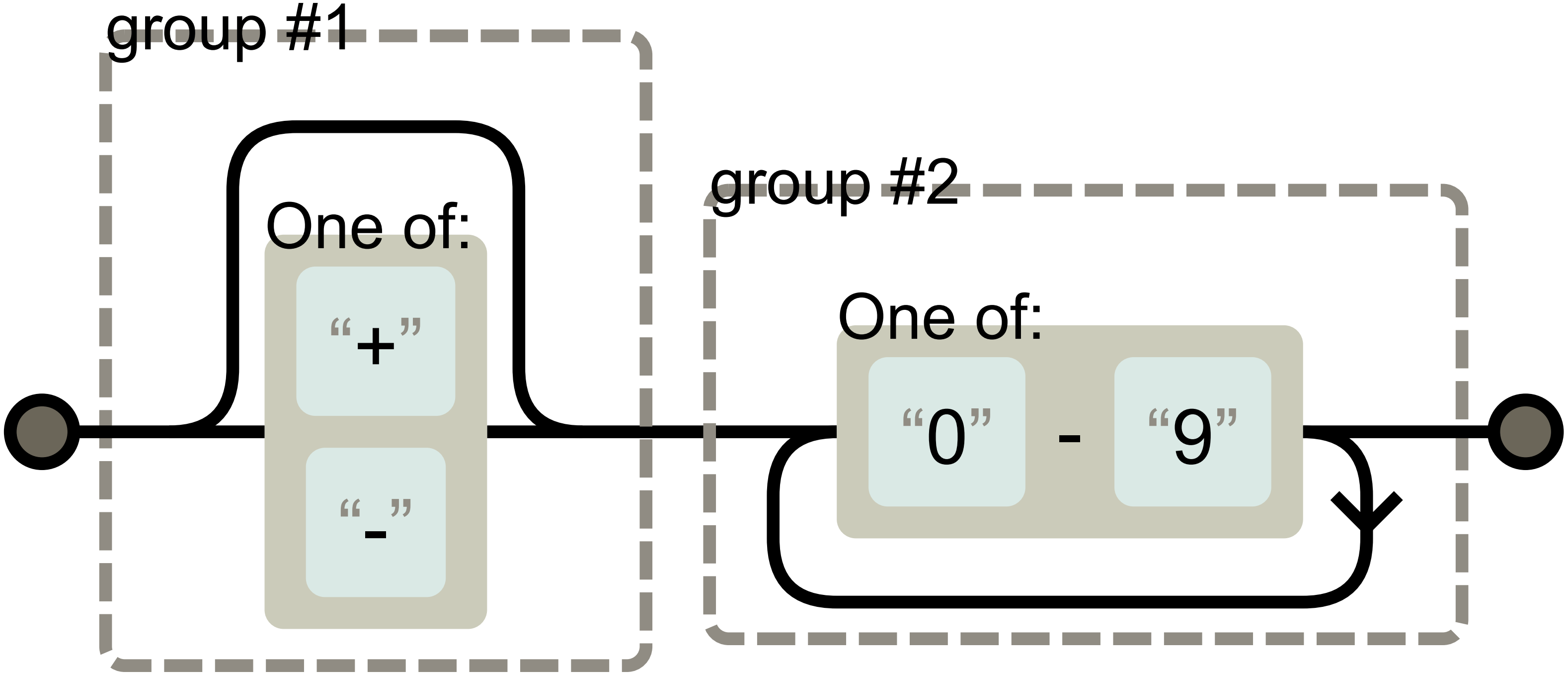
- Every pair of matching parentheses defines a capture group
  - Group 0 is the whole matched string



  - Non capturing group: `(?:E)`

Capture groups

```
([+-]?)([0-9]+)
```



Generated with: http://regexper.com

RegExp in Java

RegExp in Java

- Package
  - **java.util.regex**
- **Pattern** represents the automata:

```
Pattern p=Pattern.compile("[+-]?[0-9]+");
```

- **Matcher** represents the recognizer

```
Matcher m = p.matcher("-4560");
boolean b = m.matches();
```

Matcher

- Three recognition modes
  - **`matches()`**
    - Attemp matching the whole string
  - **`lookingAt()`**
    - Attempt a partial matching starting from beginning
  - **`find()`**
    - Attempt matching any substring
- Recognized sequences accessed with:
  - **`group()`**

Capture groups

group 1 -> first parenthesis
group 2 -> second parenthesis
group 3 -> third parenthesis

```
Pattern p=Pattern.
          compile("([+-]?)([0-9]+)");
Matcher m = p.matcher("-4560");
if(m.matches()){
   System.out.println(
        "Group 1, sign   : " + m.group(1) +
      "\nGroup 2, digits: " + m.group(2) );
}
```

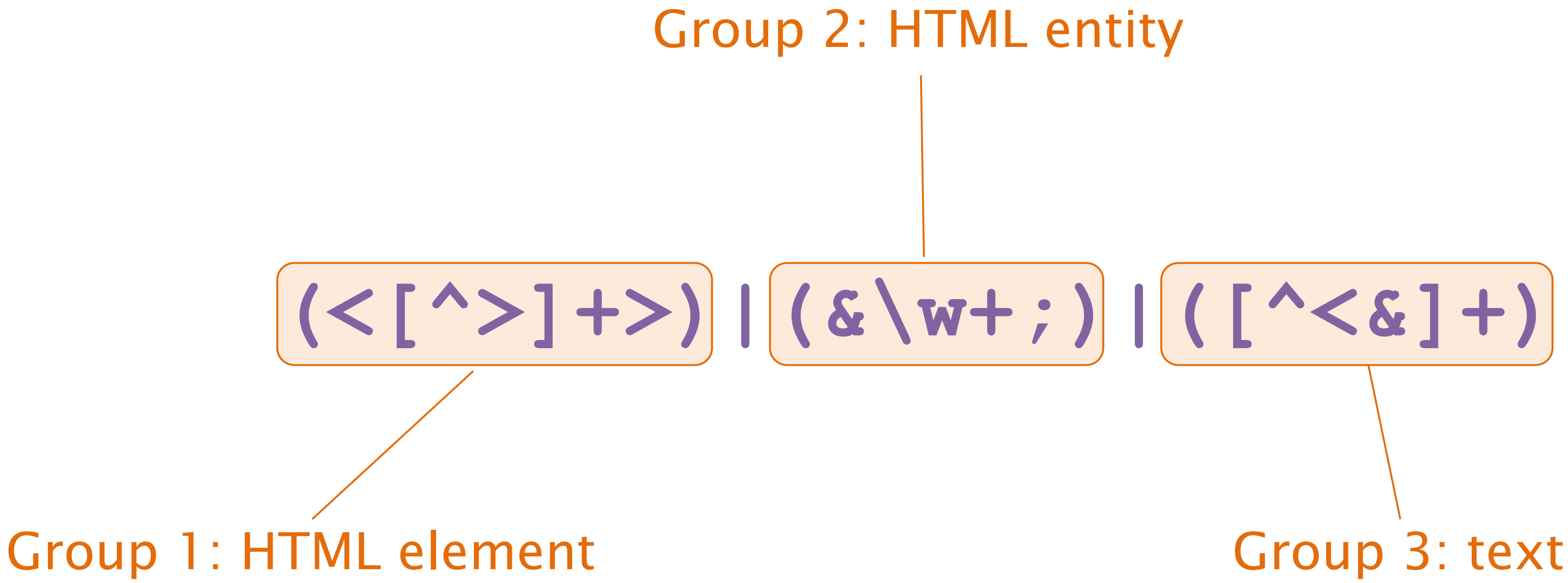
```
Group 1, sign   : -
Group 2, digits: 4560
```

Capture groups repeated

```
Pattern p=Pattern.
         compile("([+-]?)([0-9]+)");
Matcher m = p.matcher("+42 ... -3, 0");
while(m.find()){
  System.out.println(
       "Group 1, sign  : " + m.group(1) +
     "\nGroup 2, digits: " + m.group(2) );
}
```

```
Group 1, sign  : +
Group 2, digits: 42
Group 1, sign  : -
Group 2, digits: 3
Group 1, sign  :
Group 2, digits: 0
```

Example: HTML with groups



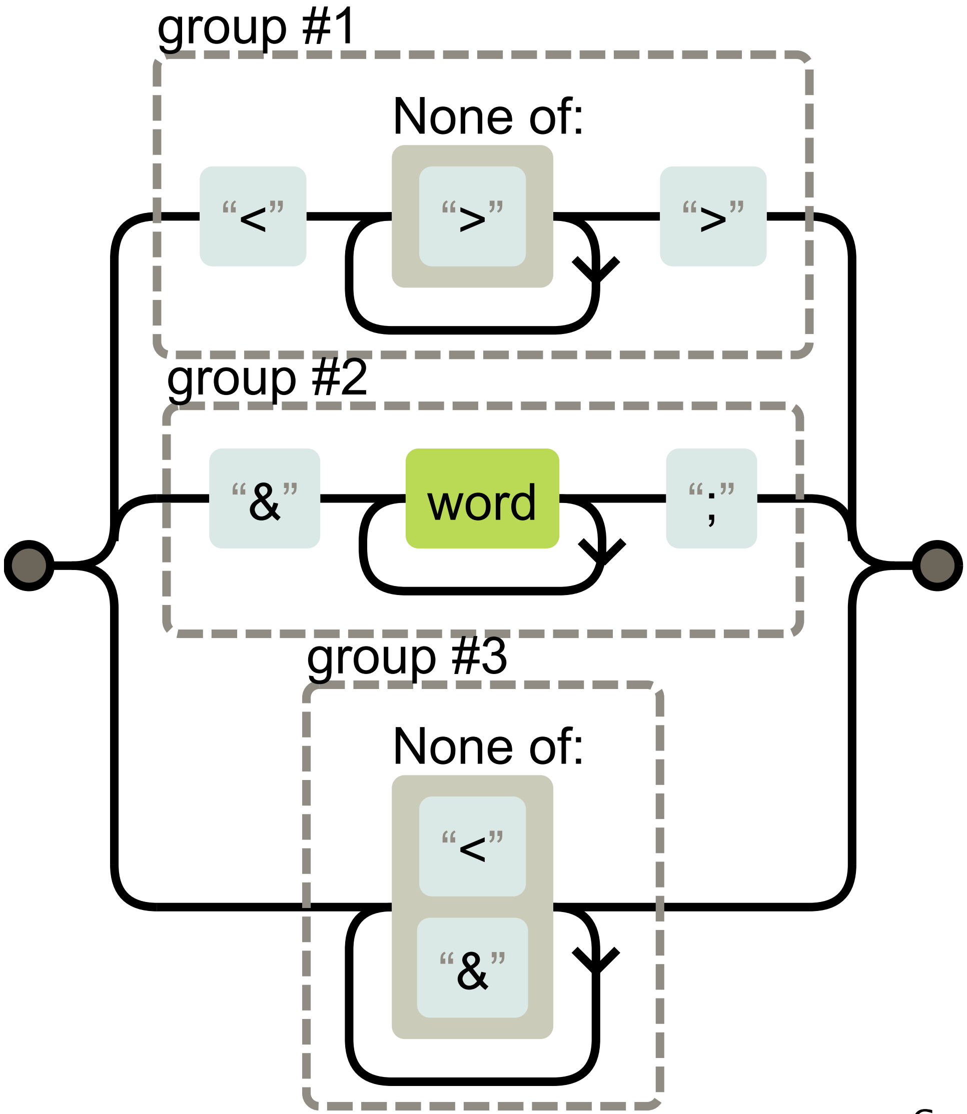
Group 2: HTML entity

```
(<[^>]+>)|(&\w+;)|([^<&]+)
```

Group 1: HTML element

Group 3: text

Example: HTML with groups



Generated with: http://regexper.com

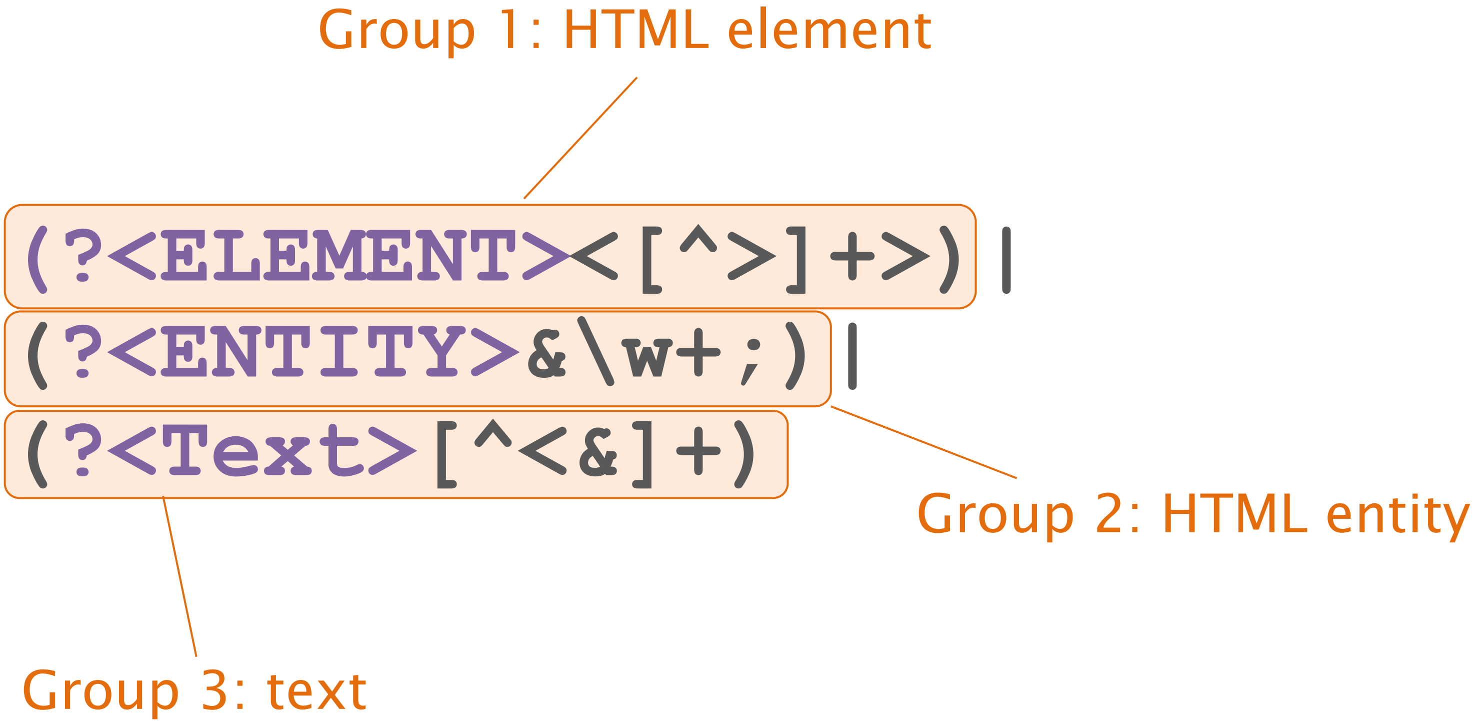
Example: HTML

```
Pattern p = Pattern.compile(
     "(<[^>]+>)|(&\\w+;)|([^<&]+)");
Matcher m = p.matcher(htmlCode);
String[] types={"ELEMENT","ENTITY","Text"};
while(m.find()){
   int type = 0;
   for(int i=1; i<=m.groupCount();++i)
     if(m.group(i)!=null) type = i;
   System.out.println(types[type-1]+" : '"+
                     m.group(type) + "'");
}
```

Named groups

- Capture groups can be named:
  - E.g. `(?<c>[^\",]*)`
- Named groups can be accessed using `group()` method:
  - E.g., `c = m.group("c");`

Ex.: HTML with named groups



Group 1: HTML element

```
(?<ELEMENT><[^>]+>)|
(?<ENTITY>&\w+;)|
(?<Text>[^<&]+)
```

Group 2: HTML entity

Group 3: text

Example: HTML w/named grps

```
Pattern p = Pattern.compile(
"(?<ELEMENT><[^>]+>)|”+”(?<ENTITY>&\\w+;)"+
"|(?<Text>[^<&]+)");
Matcher m = p.matcher(htmlCode);
String[] grps ={"ELEMENT","ENTITY","Text"};
while(m.find()){
  for(String type : grps)
    if(m.group(type)!=null)
      System.out.println(type + " : '"+
                           m.group(type)+"'");
}
```

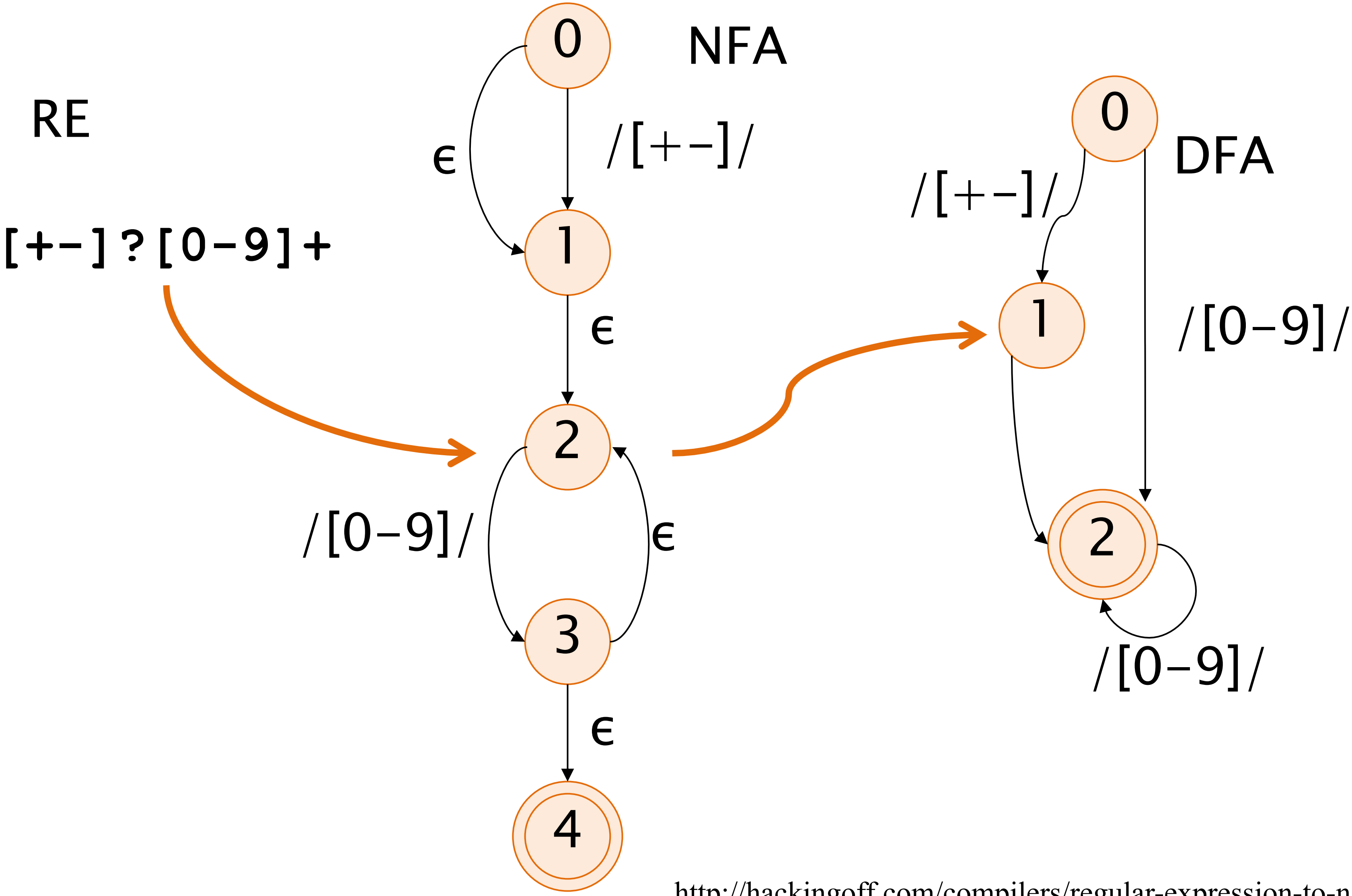
Context

- Look–behind
  - `(?<=E)` means that **E** must precede the following RE, though E is not part of the recognized RE
  - `(?<!E)` means **E** must **not** precede
- Look–ahead
  - `(?=E)` means that **E** must follow the preceding RE, though E is not part of the recognized RE
  - `(?!E)` means that **E** must **not** follow

Recognition

- A RE can be transformed into NFA (Non-deterministic Finite-state Automaton)
  - using the Thompson-McNaughton-Yamada algorithm
- Then an NFA can be transformed into a DFA (Deterministic Finite-state Automaton)
- A DFA can be encoded into a table that defines the rules *executed* by a state machine to recognize a sequence of characters

Recognizer example



RE

`[+-]?[0-9]+`

http://hackingoff.com/compilers/regular-expression-to-nfa-dfa

Class Scanner

Class **`Scanner`**

- A basic parser that can read primitive types and strings using regular expressions
- Build from:
  - File, e.g.
    - `new Scanner(new File("file.txt"))`
  - Stream, e.g.
    - `new Scanner(new FileReader("file.txt"))`
  - String, e.g.
    - `new Scanner("content,to,be,scanned"))`

Basic usage

- Built from a stream, file, or string
  - E.g., `new Scanner(new File("file.txt"))`
- Check presence of *next* token (optional)
  - E.g., `hasNextInt()`
- Parse *next* token and advance
  - E.g., `nextInt()`

Advanced usage

- Read line by line
  - E.g., `new Scanner( content )`
- Read line by line (w/optional check)
  - `hasNextLine()`
  - `nextLine()`
- Parse line looking for a given pattern
  - `findInLine( pattern )`

Scanner advanced usage

```
File file = new File("file.csv");
try(Scanner fs = new Scanner(file)){
while(true){
  String c;
  while((c=fs.findInLine(pattern))!=null){
    System.out.println(c);
  }
  if(!fs.hasNextLine()) break;
  fs.nextLine();
}}
```

[^,]+

Advanced Examples

Example: CSV with groups

```
(,|^|\n|\r\n?)
[ \t]*
(?:([^\",\n\r]*)
|"((?:[^"]*|"")*)")
[ \t]*
```

Group 1 : preceding delimiter

Group 2 : normal cell

Group 3 : delimited cell

- When translating to a string in the code pay attention to special characters:
  - Backslash: \
  - Quotes: “

Example: CSV

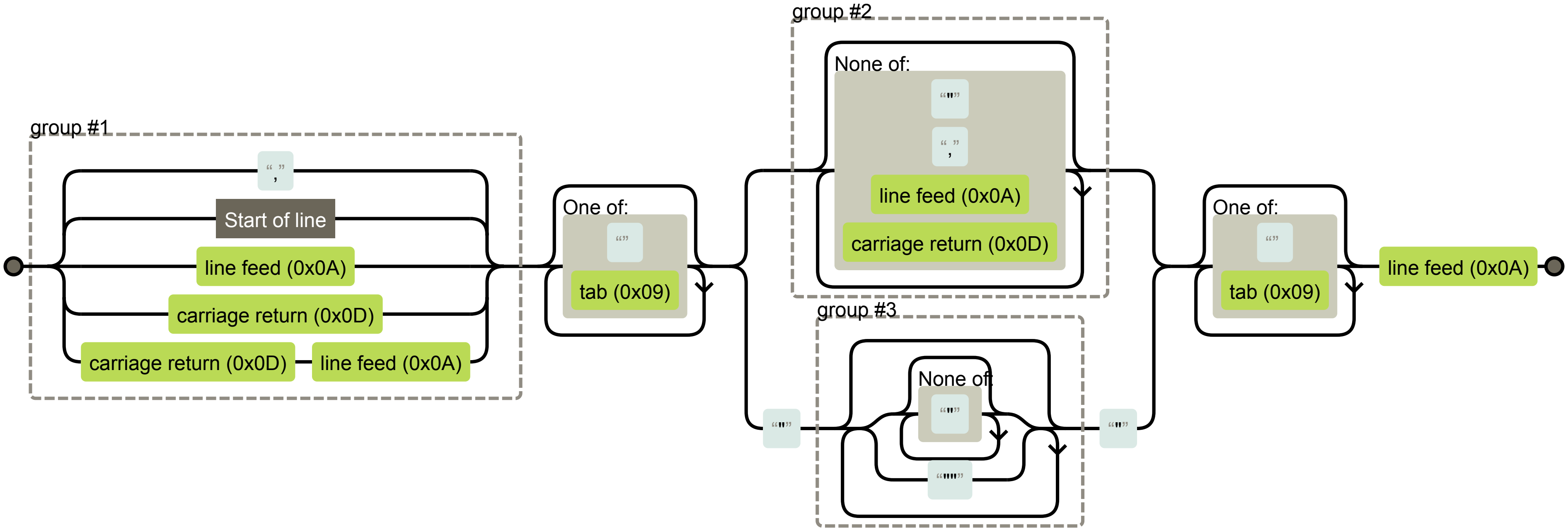
```
Pattern re  = Pattern.compile(
"(,|^|\n|\r\n?)" +      // G1 : prec sep
"[ \t]*" +              // -  : lead spaces
"(?:([^\",\n\r]*)" +    // G2 : normal cell
"|\"((?:[^\"]*|\"\")*)\")"+//G3: delim cell
"[ \t]*"                 // - : trail spaces
);
```

Example: CSV

```
Matcher m = re.matcher(csvContent);
while(m.find()){
  if(!m.group(1).equals(","))  // new row
    System.out.println("Row:");
  String c = m.group(2);
  if(c==null)
    c = m.group(3).replaceAll("\"\"","\"");
  System.out.println("\tCell:" + c);
}
```

Example CSV – Context

- Railroad diagram



Generated with: http://regexper.com

Example: CSV w/named group

```
Pattern re  = Pattern.compile(
"(?<sep>,|^|\n|\r\n?)" +     // prec sep
"[ \t]*" +                   // lead spaces
"(?:(?<c>[^\",\n\r]*)" +     // normal cell
"|\"(?<dc>(?:[^\"]*|\"\")*)\")"+//delimited
                                // cell
"[ \t]*"                    // trail spaces
);
```

Example: CSV named groups

```
Matcher m = re.matcher(csvContent);
while(m.find()){
  if(!m.group("sep").equals(",")) //new row
    System.out.println("Row:");
  String c = m.group("c");
  if(c==null)
   c=m.group("dc").replaceAll("\"\"","\"");
  System.out.println("\tCell:" + c);
}
```

Summary

- Regular expression express complex sequences of characters
- Used to recognize parts of strings
  - **`Pattern`** contains the DFA
  - **`Matcher`** implements the recognizer
- RE are used extensively
  - String: **`replaceAll()`**, **`split()`**
  - Scanner: **`findInLine()`**