LC 101

Unit 3 - Regular Expressions

April 10, 2017

Regular Expressions

- A *regular expression* is a pattern that describes a set of strings
  - (Often abbreviated as *re* or *regex*)
- Most often used to test if a candidate string matches a pattern

Basics

- A regular expression is like a mathematical formula
    - But built from characters and three basic operators: concatenation, alternation, and Kleene closure (the * operator)
        - Can also use parentheses to group parts

Concatenation and Alternation

- Individual characters are valid regular expressions
  - **`a`** is a regular expression that matches exactly one string: "**`a`**"
- Multi-character sequences formed by *concatenation* are valid regular expressions
  - **`ab`** is a regex that matches exactly one string: "**`ab`**"
- Alternation is the "or" operator. It can be used to select between two (or more) subexpressions
  - **`a|b`** is a regex that matches exactly two strings: "**`a`**" and "**`b`**"
  - **`a|b|c`** matches three strings: "**`a`**", "**`b`**", and "**`c`**"

Concatenation and Alternation

- The operands of the alternation or concatenation operators can be any valid subexpressions
- Concatenation takes precedence over alternation
- Parentheses can be used to override precedence
    - **`ab|c`** matches the strings “**`ab`**” and “**`c`**”
    - **`a(b|c)`** matches the strings “**`ab`**” and “**`ac`**”
    - **`a(b|cd|e)f`** matches the strings “**`abf`**”, “**`acdf`**”, and “**`aef`**”

Kleene Closure

- The *Kleene closure* (or *star* operator) is a unary operator that means zero or more repetitions of the preceding subexpression
  - Creates an infinite set of matching strings
    - **`a*`** matches the strings “”, “**`a`**”, “**`aa`**”, “**`aaa`**”, “**`aaaa`**”, “**`aaaaa`**”, …
  - Has higher precedence than concatenation
    - **`ab*`** matches “**`a`**”, “**`ab`**”, “**`abb`**”, “**`abbb`**”, “**`abbbb`**”, …
    - **`(ab)*`** matches “”, “**`ab`**”, “**`abab`**”, “**`ababab`**”, “**`abababab`**”, …
    - **`(a|b)*`** matches “”, “**`a`**”, “**`b`**”, “**`aa`**”, “**`ab`**”, “**`ba`**”, “**`bb`**”, “**`aaa`**”, “**`aab`**”, “**`aba`**”, “**`abb`**”, “**`baa`**”, …

Other Operations

- Although those three basic operations (concatenation, alternation, and Kleene closure) are all that are needed to create any expression, a few other helpful operations are usually defined
  - The plus operator means one or more of the preceding subexpression
    - **`a+`** matches the strings “**`a`**”, “**`aa`**”, “**`aaa`**”, “**`aaaa`**”, …
  - The question mark operator means zero or one of the preceding subexpression
    - **`a?`** matches “” and “**`a`**”
    - **`ab?c`** matches “**`ac`**” and “**`abc`**”
  - {*n*} means exactly *n* iterations of the preceding subexpression
    - **`ab{3}c`** matches the string “**`abbbc`**”
  - {*m*,*n*} means at least *m* but no more than *n* of the preceding subexpression
    - **`ab{1,2}c`** matches the strings “**`abc`**” and “**`abbc`**”

Limitations

- While powerful, regular expressions are limited in the types of sets they can describe
  - There are seemingly simple cases that regular expressions cannot handle
    - For example, there is no regular expression that can be used to determine if a string has properly balanced nested parentheses (without additional restrictions)
      - This means that we cannot use a regular expression to determine if a string is a valid regular expression!
        - Or mathematical expression

From Theory to Practice

- Many programming languages have regular expressions as either part of the language itself or as part of their standard libraries
- Though there is a POSIX standard, most languages and libraries seem to have their own slightly different variant
  - The theory is the same but the exact syntax varies

Any Character

- A dot matches any single valid character
  - Includes special characters but sometimes does not include the newline character
    - `.` matches “**`a`**”, “**`b`**”, …, “**`1`**”, “**`2`**”, …, “**`%`**”, “**`@`**”, …

Character Sets

- A character set matches any character contained in the brackets
  - `[abc]` matches “`a`”, “`b`”, and “`c`”
- Character sets can use a dash to indicate a range of characters
  - `[a-e]` is the same as `[abcde]`
- A character set can be negated by starting it with the hat character
  - `[^abc]` means any character *except* `a`, `b`, or `c`

Special Characters

- `\d` matches a digit character. Equivalent to `[0-9]`
- `\D` matches a non-digit character. Equivalent to `[^0-9]`
- `\s` matches a whitespace character
  - Space, tab, form feed, line feed
- `\S` matches a non-whitespace character
- `\t` matches a tab
- `\n` matches a form feed character
- `\r` matches a carriage return
- `\w` matches an alphanumeric character or underscore. Equivalent to `[A-Za-Z0-9_]`
- `\W` matches a non-word character. Equivalent to `[^A-Za-z0-9_]`

Escaping Special Characters

- The backslash **`\`** is used to escape special characters
  - **`\*`** matches the literal "**`*`**" and does not mean the Kleene closure
  - **`\.`** matches a literal "**`.`**" and does not mean any character
- Must also be used to escape a literal backslash
  - **`\\s`** matches "**`\s`**"

Creating Regular Expressions

- Regular Expressions can be created using either a regular expression literal or the RegExp constructor
  - The literals are compiled when the script is loaded, so use them for better performance for constant expressions
  - The RegExp constructor allows specification of an expression at runtime

```
var re1 = /ab*/;
var re2 = new RegExp('ab*');
```

Finding Matches

- `re.test(str)` searches for a match in a string and returns `true` or `false`
- `str.search(re)` searches for a match in a string and returns the index of the first match found
  - And returns -1 if no match is found
- Note that these search for a match in a string. They do not check that the entire string matches.
- Also note that `test` is a method of `RegExp` while `search` is a method of `String`

```
var re = /ab*/;
var str = 'cabbc';
console.log(re.test(str));    // outputs true
console.log(str.search(re));  // outputs 1
```

Start and End of Input

- Often we want to see if an entire string matches an expression and not just if it contains a match
- Special characters can be used in regular expressions to match the start or end of input
  - **^** matches the start of input and **$** matches the end of input
  - So using **^** at the beginning of an expression and **$** at the end will then require the entire string to match

```
var re = /^ab*$/;
var str1 = 'abb';
var str2 = 'cabbc';
console.log(re.test(str1));  // outputs true
console.log(re.test(str2));  // outputs false
```

Spaces

- Note that the literal space character matters in regular expressions!

```
var re = /hello goodby/;
var str1 = 'hello goodby';
var str2 = 'hellogoodby';
console.log(re.test(str1));  // outputs true
console.log(re.test(str2));  // outputs false
```

Capturing Groups

- When parentheses are used in regular expressions, you can access the substring that matched the subexpression in the parentheses
    - `re.exec(str)` returns a result array on successful match or `null` if not found
    - `str.match(re)` returns a result array on successful match or `null` if not found

```
var re = /^(ab*)(c*d)$/;
var str = 'abbccd';

console.log(re.exec(str));
// outputs ["abbccd", "abb", "ccd", index: 0, input: "abbccd"]

console.log(str.match(re));
// outputs ["abbccd", "abb", "ccd", index: 0, input: "abbccd"]
```

Non-Capturing Parentheses

- If you need to use parentheses but don't care about the subgroup then you can use **`(?:x)`**

```
var re = /^(?:ab*)(c*d)$/;
var str = 'abbccd';

console.log(re.exec(str));
// outputs ["abbccd", "ccd", index: 0, input: "abbccd"]
```

Greedy vs Lazy

- The *, +, and ? operators are greedy
  - The match as much as they can to get a valid match
- You can make the operators lazy by putting another ? after them
  - Then they will match as little as they can to get a valid match

```
var str = 'abcbc';

var re1 = /a.*c/;   // greedy
console.log(re1.exec(str));
// outputs ["abcbc", index: 0, input: "abcbc"]

var re2 = /a.*?c/;  // lazy
console.log(re2.exec(str));
// outputs ["abc", index: 0, input: "abcbc"]
```

Replace

- The **`string.replace`** method can be used to replace parts of a string that match a regular expression
  - By default it will just replace the first match. To replace all matches, the *global modifier* must be used on the regular expression
    - The modifiers come after the closing slash in a regexp literal or as the second parameter of the RegExp constructor

```
var re = /fish/g;
var str = 'red fish blue fish';
console.log(str.replace(re, 'bird'));  // outputs "red bird blue bird"
```

Multiple Matches

- The global modifier also changes how match and exec work
  - match will return a simple array of the strings that match instead of the first match
  - exec can be called multiple times to return subsequent matches

```
var str = 'red fish blue fish';
console.log(str.match(/fish/));   // outputs ["fish", index: 4, input: "red fish blue fish"]
console.log(str.match(/fish/g));  // outputs ["fish", "fish"]

var re = /fish/g;
console.log(re.exec(str));  // outputs ["fish", index: 4, input: "red fish blue fish"]
console.log(re.exec(str));  // outputs ["fish", index: 14, input: "red fish blue fish"]
```

Real-Life Example

- In JavaScript, there is no existing function to determine if an entire string is an int
  - parseInt will return a number if the first part of a string is digits
  - The Number constructor will work for both floats and ints

```
var str1 = '42a';
console.log(parseInt(str1, 10));  // outputs 42

var str2 = '4.2';
console.log(new Number(str2));  // outputs Number {[[PrimitiveValue]]: 4.2}
```

Real-Life Example

- We can use a regular expression to determine if a string contains only digits
  - What would happen if the expression below was just “`/\d+/`”?

```
var intRegExp = /^\d+$/;

console.log(intRegExp.test('42a'));  // outputs false
console.log(intRegExp.test('4.2'));  // outputs false
console.log(intRegExp.test('42'));   // outputs true
```