Regular Expressions in Swift

Guy Lapalme
RALI-DIRO
Université de Montréal
October 2024

The Swift programming language's regular expression notation is unique enough to warrant an explanation. While it adheres to most of the conventions found in languages such as PERL, Java, JavaScript, PHP and Python, it also has some significant differences that I found challenging to understand and had to figure out through trial and error. This document is the product of my efforts to understand and I am sharing it in the hope that it will help others.

I will first review what regular expressions are, and then demonstrate how they can be represented and used in Swift. I will use three examples to highlight some unique aspects of Swift's regular expression implementation: parsing Roman numerals, a tokenizer, and an Eliza-like chatbot. The appendix includes links to additional resources and a handy cheat sheet.

This document serves as a personal reference for me, as I couldn't find a comprehensive guide on Swift regular expressions. I have searched the internet for tutorials and videos on Swift regexes, but I have not found a clear definition of what they can be used for and why. Most examples I have seen are repetitive, so I have decided to develop my own that each illustrate a specific point.

[HTML version] [Markdown version] [PDF version]

1. What is a *regex* ?

In 1951, S. C. Kleene introduced the concept of a *regular expression*, also known as *regex*, as a formal language construct that describes the collection of strings formed through the operations of concatenation, alternation, and repetition, or quantification. This concept was later applied in the 1970s to define patterns for matching lines in a file. For instance, the Unix `grep` command, whose name derives from "global regular expression print", searches for and displays lines that match a specified regular expression. This notation has proven to be extremely practical and effective for extracting data from extensive texts, as well as for defining modifications in text editors.

Most modern programming languages offer a regular expression syntax for extracting the string parts that match a regex, or for indicating failure if no match is found.

Swift is a strongly typed programming language, and its creators have taken great care to ensure that regular expressions and their outputs comply with type constraints. Swift's regular expression syntax was introduced in 2022, along with Swift 5.7. Previous Swift versions relied on a port of the NSRegularExpression Objective-C library, which was less than ideal due to the complexity of bridging Objective-C's `NSString` and Swift's `String`. As a result, many people, including me, created custom functions for common tasks. However, these *new* regular expressions only work on iOS 16.0+ and MacOS 13.0+.

Swift's syntax may be new, but it shares similarities with well-known languages like Perl, Java, JavaScript, PHP, and Python. Despite some difficulties, I have managed to understand the main aspects of the Swift regex API. This document summarizes my understanding, which I believe can benefit others.

We first briefly recall what is a regular expression to define the terms used in the rest of the document. In this section, we focus on the features that are common to all regex notations. Swift peculiarities will be detailed later.

1.1. Components of Regex Definitions

A regular expression defines a *pattern* whose occurrences must be *matched* within a string, the *subject*. We follow the time-honored terminology of SNOBOL4 which, in the sixties, was the first programming language to allow the definition of patterns as first-class objects for matching and operating on strings.

A *pattern* (a `RegexComponent` in the Swift parlance) is a combination of :

- `Character` : a letter (e.g., `a` or `w`) or a period (`.`) which stands for any letter. It can also be specified by a set of characters within square brackets, such as `[abc]` or a range of characters, such as `[a-z]` for any English lowercase letter. A character set can be complemented by starting it with `^`. There are also predefined patterns, such as `\d` for matching a digit, `\w` for matching a character that can appear in a word (letter, digit or underline), `\W` for matching a *non-word* character (the set complement of `\w`) or `\s` for matching any space character (e.g., newline, tab, space, carriage return).

  A character in a Swift string can be a *Unicode extended grapheme cluster* that can span more than one byte; so it must be retrieved using `String.Index` values and not by integer values on the byte representation. The rules of character equality in Swift based on canonical equivalence also apply to regexes. For example, a character with a diacritic may be represented by a single Unicode character or by a base character followed by a *combining accent*. Therefore, the strings `"è"` and `"e\u{300}"` (*e* followed by a *combining grave accent*) will match `.` corresponding to any single character .
- *sequence* of patterns that the subject must match consecutively in the subject
- *alternation* between patterns denoted by a vertical bar (`|`) between patterns, alternation matches when one of the choices matches the subject.
- *position* : check if the subject matches on certain conditions: e.g. `^` matches only at the start of the string, `$` matches at the end of the string, `\b` matches at a word boundary, i.e. between a `\w` and a `\W`.

Sequence has priority over alternation, but parentheses can be used to change this ordering. For example, the pattern `abc|def|ghi` matches occurrences of `abc` , `def` or `ghi` , while `a(bc|de)f|ghi` matches occurrences of `abcf` , `adef` or `ghi` . But we will see later that parentheses are also used for delimiting parts of patterns that matches that are called *capture groups*.

A pattern can be repeated a certain number of times by adding a *quantifier* after it:

- `?` : [0,1]

- `*` : [0,+∞)
- `+` : [1,+∞)
- `{m,n}` : [m,n] `m` defaults to 0 and `n` to +∞
- `{m}` : [m,m]

By default, repetition of an indeterminate number of times is `eager` to find the longest match which is most often what is needed. In some cases, this strategy can lead to unexpected results. For example, given the subject string `"hello" my "friends"`, the pattern `".*"` will match once the whole subject, matching the quotes at the start and at the end of the string because the dot matches any character including a quote.

Repetition can be specified as either:

- `reluctant` by adding `?` after it. So `".*?"` will instead match the two quoted words, each match stopping as soon as a quote is encountered.
- `possessive` by adding `+` after it. *Possessive* matching finds the longest match without ever backtracking which is more efficient in some cases, but it can lead to surprising results. For example, `".*+"` fails on our subject string, because `.*` matches all following characters without ever reconsidering its choices, so it does not find the trailing quote, because it was already matched by the dot `.` . In this case, instead of the dot, we should match any character except a quote using `"[^"]*+"`. This pattern will now match the two quoted words.

When a regex must match a special character used for alternation or quantifier, this character must be escaped by a backslash, e.g., so matching `a` and `b` separated by `+` must be defined as `a\+b` otherwise it would match one or more `a` followed by a `b`.

Matching a character looks simple, but there can be variations. Should matching be case insensitive? Should diacritics be taken into account? For a multiline string, should position matches apply to each line ? Each language has its own way (usually *flags*) of specifying changes in matching behavior.

This very brief refresher on regex syntax is very far from complete, but it is adequate for explaining how regexes work in Swift. For a more complete list of metacharacters and operators.

1.2. Result of the matching process

When applying a *pattern* to a *subject*, one of two things may happen:

- *Failure* : the pattern was not found on the subject. Failure returns `nil` in Swift (corresponding to `None` in Python or to `null` in Java or JavaScript). For old-timers: in SNOBOL4, control went to the instruction in the `:F(..)` *GOTO field*.
- *Success* : At least one occurrence of the pattern was found in the subject, so a *non-failure* value is an object with properties that provide information about the match. These include the start and end positions of the occurrence (`range` in Swift) or the substring itself (`output` in Swift). When a pattern contains *subpatterns* (called capture groups), the object provides access to them.

  When a pattern appears more than once in a subject, a collection of successes can be returned, either as a list or as a generator that yields a match each time.

Given that regular expressions in most programming languages are defined using strings, errors or exceptions can occur at run-time if the regex string is not syntactically correct (e.g., unbalanced parentheses). Some compilers check the syntax of string literals in some special cases, but not in all of them. Swift goes to great lengths to limit the risk of such failures by applying static type checking to regexes as well. In other type-checked languages such as Java, regexes are plain strings whose peculiar structure is checked by a *compile* method, which is called at run-time.

2. Regex in Swift

The matching process can be relatively slow because the pattern must be interpreted while scanning the subject. Most programming languages allow the *compilation* of a regex from the string to create an automaton for faster matching. A Swift regex is compiled by *default* to ensure that it is well formed and type-checked. Swift defines a `Regex` type used for type checking any expression involving regexes.

2.1. Swift Regex Definition

Swift provides three notations for defining a regular expression:

- `Regex` literal enclosing the expression between two slashes, e.g. `/a(bc|de)f|ghi/` or `/".*?"/`. In order to avoid ambiguity with the single slash used for division, a regex cannot start with a space. A regex literal cannot be empty because `//` is used for line-ending comments; an empty regex would not be very useful anyway.
- `RegexBuilder` expression, a more verbose but more expressive for complex patterns. The first two columns of the following table show `RegexBuilder` expressions corresponding to our two previous examples. We will present this notation in the next section.
- *extended regex literal*, `#/.../#` which avoids the need to escape forward slashes within the regex. When the opening delimiter is followed by a new line, it defines a *multi-line literal* in which whitespace and line-ending comments starting with `#` are ignored. The third column of the next table shows how our second example can be written. This can be useful for documenting complex regular expressions.

RegexBuilder	RegexBuilder	Extended literal

`Regex {`
`  ChoiceOf {`
`  Regex {`
`  "a"`
`  ChoiceOf {`
`  "bc" ; "de"`
`  }`
`  "f"`
`  }`
`  "ghi"`
`  }`
	`Regex {`
	`  /"/`
	`  ZeroOrMore(.reluctant) {`
	`  /./`
	`  }`
	`  /"/`
		`#/`
		`  "  # opening quote`
		`  .*? # up to next quote`
		`  "  # ending quote`
		`/#`

We will later show how to define run-time regular expressions using strings. In the remainder of this section, we use regex literals as the other notations are equivalent. We want to focus on the matching process, not on the syntax of the regular expressions.

2.1.1. Regex Modifications

Changing some aspect of the matching behavior, obtained using *flags* in other programming languages, is performed by calling a `Regex` method that returns a new `Regex`:

- `.ignoresCase()` for case-insensitive matching, e.g. `/a(bc|de)f|ghi/.ignoresCase()`
- `.dotMatchesNewlines()` the any character (`.`) also matches an end of line.

2.2. Swift Regex Operations

Regular expressions are used for identifying which parts of a subject correspond to the pattern. In some cases, it is enough to only check whether a pattern occurs in the subject, but more often it is necessary to get more information about the match.

Here are the definitions of a pattern and a subject used in the following examples of calls.

```
let identifier = /[A-Za-z]\w*/  // a letter, possibly followed by word characters (letter, digit or underline)
let subject = "Here are 10 tokens to be (matched) !"
```

2.2.1. Checking for an Occurrence of a Pattern

The `String` methods `.contains(..)` and `.starts(with:..)` return a `Bool` (the value of each expression is shown here after `// =>`)

```
subject.contains(identifier)          // => true
"123 + 456".contains(identifier)      // => false

subject.starts(with:pattern)          // => true
"123 + 456".starts(with:identifier)   // => false
```

2.2.2. Looking in a Subject for Matches of a Pattern

2.2.2.1. Finding a single match

In Swift, the result of applying a pattern to a subject results in *match object* of *optional* type `Regex<Output>.Match?` . The `String` methods `.firstMatch(of: Regex)`, `.prefixMatch(of: Regex)` and `.wholeMatch(of: Regex)` return *nil* when no instance of the pattern is found, otherwise they return an object whose properties give access to matching information such as

- `.output`: the substring of the subject matched; as this is a substring of the subject, it must often be transformed into a new string using the `String(...)` constructor, when it must be printed or used as a `String` parameter for another function.
- `.count`: the length of the match
- `.range` : the interval of string indices spanning the match

Caution: `.output` and `.count` have slightly different meanings for a dynamic regex defined by a string.

Here are examples of calls that use *optional chaining* operator `?.` that returns `nil` when its left part is `nil`; the binary *nil coalescing operator* `??` *unwraps* its left `Optional` operand if it is not `nil` otherwise it returns its right operand. `a ?? b` can be understood as `a != nil ? a! : b`.

In this case, the subject is matched for an instance of a pattern

```
subject.firstMatch(of: identifier)?.output ?? "no match!"    // => "Here"
subject.firstMatch(of: identifier)?.count ?? "no match!"    // => 4

subject.prefixMatch(of: identifier)?.output ?? "no match!"   // => "Here"
subject.wholeMatch(of: identifier)?.output ?? "no match!"    // => "no match!"
```

2.2.2.2. Finding all matches

To get an array of all matches, we use the `String` method `.matches(of: Regex)` . When no match is found in the subject, the array is empty in the spirit of replacing failure by a list of successes..

This is useful for transforming all matches using a closure. Here is an example that returns the matched substrings in upper case.

```
subject.matches(of: identifier).map{m in m.output.uppercased()}
  // => ["HERE", "ARE", "TOKENS", "TO", "BE", "MATCHED"]
```

Here is a simple minded tokenizer that return strides of numbers or letters and single arithmetic operators, while ignoring the rest; see this section for a more comprehensive tokenizer. Note that within a character class, it is not necessary to escape special characters used in regex(e.g. `*` or `+` and parentheses), but `-` must be the first character of the range. `/` must be escaped so that it does not indicate the end of the regex

```
"12 + (a_bc*435)- 78".matches(of:/\w+|\d+|[-+*\/()]/).map{String($0.output)}
  // => ["12", "+", "(", "a_bc", "*", "435", ")", "-", "78"]
```

2.2.3. Given a Pattern, Finding Matches within a Subject

A pattern can also be searched within a string using `Regex` methods bearing the same names as in the previous section but with a different keyword for their parameter, because it is a pattern that is searched in a string, not a string in which occurrences of a matches are looked for. Note that there is not equivalent to `.matches(..)` for patterns.

As it will be explained later, in some regexes the result of a match can be transformed with code that might raise an exception, so a call to one of these functions must be prefixed by `try` , even though there is no transformation involved here.

```
try identifier.firstMatch(in: subject)?.output ?? "no match!"  // => "Here"
try identifier.firstMatch(in: subject)?.count ?? "no match!"   // => 4

try identifier.wholeMatch(in: subject)?.output ?? "no match!"  // => "no match!"
try identifier.prefixMatch(in: subject)?.output ?? "no match!" // => "Here"
```

2.2.4. Replacing the Match in the Subject

Once a match is found, we often want to change the matching part of the subject by another string, the *replacement*. There are two related `String` methods:

- *subject* `.replacing(` *pattern*, `with:` *replacement* `)` returns a new string in which occurrences of the *pattern* in the *subject* have been replaced with the *replacement*. The replacement can be another string so that all occurrences will be changed by the same string. If the replacement is a closure, the replacement can depend on each occurrence of the pattern in the subject because it called at each occurence with the match as parameter.
- *subject* `.replace(` *pattern*, `with:` *replacement* `)` replaces the occurrences of the *pattern* within the *subject* which must be declared as `var`.

In the next example, the replacement is a string, so it creates a new string in which each *identifier* is replaced by `*id*`, unmatched substrings (here `10` and punctuation signs) are not modified.

```
subject.replacing(identifier, with: "*id*")
  // => "*id* *id* 10 *id* *id* *id* (*id*) !"
```

A closure is needed when the replacement depends on the content of the subject such as in the following example where each identifier is wrapped in square brackets:

```
subject.replacing(identifier, with: {m in return "[\(m.output)]"})
  // => "[Here] [are] 10 [tokens] [to] [be] ([matched]) !"
```

This notation can be simplified with a trailing closure accessing to the match with `$0` , the implicit first parameter for a closure in Swift. `$n` happens to be also the notation used in regex replacements in many programming languages.

```
subject.replacing(identifier){"{\($0.output)}"}
  // => "{Here} {are} 10 {tokens} {to} {be} ({matched}) !"
```

Here is an example of a replacement within a *variable* string. Caution `.replace(...)` returns `Void` as a reminder of the side effect of this expression.

```
var subjectV = subject
subjectV.replace(identifier){"#\($0.output)#"}
  //  => ()
subjectV
  //  => "#Here# #are# 10 #tokens# #to# #be# (#matched#) !"
```

2.2.5. Determining the Ranges of Matches

To get the list (possibly empty) of the *ranges* of matches within a string, we can use `.range(of:..)`. A range in Swift can be used for indexing a string to get the corresponding substring. For example, here to get the list of matched substrings.

```
subject.ranges(of: identifier).map{rng in subject[rng]}
  // => ["Here", "are", "tokens", "to", "be", "matched"]
```

2.2.6. Splitting a String According to a Pattern

A regex can also be used to split a string to get an array of substrings between *separators*. For example, a *simple minded* tokenizer can be implemented by splitting using a non-empty sequence of characters that cannot be part of a word `/\W+/`:

```
subject.split(separator: /\W+/)
  // => ["Here", "are", "10", "tokens", "to", "be", "matched"]
```

This removes the separators, but keeping them in the split is a bit more involved. We could instead try matching an empty string (called a *lookBefore*) before the non-word pattern , here `/(?=\W+)/`:

```
subject.split(separator: /(?=\W+)/)
// => ["Here", " are", " 10", " tokens", " to", " be", " ", "(matched", ")", " ", "!"]
```

Unfortunately, this is not adequate because some tokens have spaces or a parenthesis before them. To avoid this problem, the separator should also use a *lookBehind* `(?<=\W+)` match, so the expression should be the following:

```
subject.split(separator: /(?=\W+)|(?<=\W+)/)  // *** DON'T DO THIS
// => ["Here", " ", "are", " ", "10", " ", "tokens", " ", "to", " ", "be", " ", "(",
       "matched", ")", " ", "!"]
```

While Swift recognizes the `lookBehind` syntax, it warns that it is not yet implemented. Getting the list of both the substrings and the separators can be achieved using the `.ranges(of:)` method described in the previous section. Using the list of ranges of separator occurrences, we build the list substrings between each range while adding also the content of each separator. This can be implemented with this `String` extension.

```
extension String {
    func splitKeeping(separator:Regex<Substring>)->[Substring]{
        var result=[Substring]()
        var pos = self.startIndex
        for rng in self.ranges(of:separator) {
            if rng.lowerBound != pos { // add substring before the separator
                result.append(self[pos..<rng.lowerBound])
            }
            result.append(self[rng]) // add the separator
            pos = rng.upperBound
        }
        if pos != self.endIndex { // add rest of string after last separator
            result.append(self[pos..<self.endIndex])
```

```
        }
        return result
    }
}
```

The following expression separates a string at any *non-word* ignoring tokens comprising only a single space which are not considered useful.

```
subject.splitKeeping(separator: /\W/).filter{$0 != " "}
  // => ["Here", "are", "10", "tokens", "to", "be", "(", "matched", ")", "!"]
```

2.2.7. Removing a Match at the Start of a Subject

To remove a match at the start of a subject, `.trimmingPrefix(..)` can be used. For example

```
subject.trimmingPrefix(identifier)
  // => " are 10 tokens to be (matched) !"
```

If the pattern does not match the beginning of the subject, the subject is returned unchanged. Similar to the `.replacing(..)/.replace(..)` pair, `.trimPrefix(..)` removes the start of the subject which must have been declared as `var`.

2.3. Capturing Information within a Match

Regular expressions are useful for extracting information from strings. Once a match is found, parts of it can be *captured*. For illustrating this concept, we define a pattern to extract key-value pairs: the key is an identifier and the value is a series of digits. They are separated by an equal sign with optional spacing between them. The value field with the preceding equal sign can be omitted. The following is an example subject.

```
let keyValues = "a=3, b, c = 5, d, e=   10"
```

The following pattern can be used to extract the substring associated with the key and the value substrings. Capture groups are delimited by parentheses and numbered from the left according to their open parentheses (the comment line below shows the opening parenthesis starting each group), group 0 is the substring corresponding to the whole match.

```
let kvPat = /([A-Za-z]\w*)(\s*=\s*(\d+))?/
//           1            2       3
```

In this pattern, group 1 is the key, while group 3 is the value: these values can be accessed through indexing (e.g. `m.1` or `m.3`) like any Swift tuple.

This convention is widely used in regular expressions in programming languages. In Swift, however, because of the strong typing discipline, adding capture groups changes the type of the match. This is because it creates a tuple of *n+1* substrings, *n* being the number of capture groups. In our example, the type becomes `Regex<(Substring, Substring, Substring?, Substring?)>`. We see that groups numbered 2 and 3 are associated with the part that can be omitted, so it is given an `Optional` type indicated by a trailing `?`. So although the match succeeds, some capture groups can still be `nil`.

The next example returns the list of values as integers defaulting to 1 when no value is given.

```
keyValues.matches(of: kvPat).map{Int($0.3 ?? "1")!}
  // => [3, 1, 5, 1, 10]
```

In the resulting array of `.matches(of:..)`, it is guaranteed that each match is not `nil`, but this is not the case for `$0.3`. Using the *nil coalescing operator*, the string `"1"` is returned when it is `nil` which is passed to the `Int` constructor to return an integer corresponding to this substring. But the `Int` constructor itself returns an `Optional` value; it might return `nil` if the substring does not correspond to the syntax of an integer. Here given that the substring contains only digits, it can be unwrapped unconditionally to get an integer value. This

explains the use of the final `!` operator.

Contrarily to other programming languages such as Python or Java, capture groups cannot be indexed by an integer variable. As Swift tuple components may be of different types, they must be accessed by a known subscript to allow to determine the type of the chosen component. In some cases, it is possible to use *reflection tricks* to transform a tuple into an array whose all elements must be of the same type and thus indexable by a variable.

2.3.1. Naming Capture Groups

Keeping track of group numbers is error-prone, especially when, during the development, adding or removing groups within a pattern. It is thus possible to assign names to groups by starting the group with `?<name>` . This allows documenting the kind of values expected in the groups from the subject. We can thus give a more explicit version of the previous pattern as

```
let kvPatN = /(?<key>\p{alpha}\w*)(\s*=\s*(?<value>\d+))?/
```

in which group 1 is given the name `key` and group 3 the name `value`. This change is also reflected in the type of the match: `Regex<(Substring, key: Substring, Substring?, value: Substring?)>` in which some fields have been given the name of the capture group.

As spacing around the equal sign is not relevant, we can ignore a group by starting it with `?:` in the output but the parentheses are kept for delimiting the optional grouping. Our previous example becomes

```
let kvPatN = /(?<key>\p{alpha}\w*)(?:\s*=\s*(?<value>\d+))?/
```

which now has the type `Regex<(Substring, key: Substring, value: Substring?)>` ignoring the capture group starting with `?:`. We can now rewrite our example of extracting the integer values of the subject with the following version easier to understand by using the name of the field (here `value`).

```
keyValues.matches(of: kvPatN).map{Int($0.value ?? "1")!}
    // => [3, 1, 5, 1, 10]
```

It is still possible to use indexing (i.e here use `$0.2` instead `$0.value`) but this *defeats the purpose* of defining names for captured groups.

2.3.2. Matching Capture Groups

A capture group can also be used within the same regular expression to match a repetition of a previous match. Technically, this does not fit the theoretical definition of a regular expression, but this is sometimes useful. The reference is obtained by using the `\n` pattern where `n` is the number of the group.

For example, `/([a-z]+)\1/` matches a substring of consecutive repeated lowercase letters such as `abcabc`. For named capture groups, reusing a previous match is achieved with `\k<name>`. So the previous example, could have been written as `/(?<x>[a-z])+\k<x>/`.

To illustrate the use of matching captured string, we develop a pattern for removing XML tags from a subject. We first recall the main rules for XML tags, see this document for more details. We do not advocate using regular expressions to parse XML, but this is an interesting pedagogical exercise.

An XML tag is an NCNAME inside angle brackets. *NCNAME* (name without a colon) is an identifier starting with a letter or an underscore, possibly followed by a list of letters, digits, underscores, hyphens or periods, corresponding to the `/[a-zA-Z_][-a-zA-Z0-9_.]*/` expression literal (note that the hyphen at the start and the period in the character class are not considered as special characters).

There are three types of XML tags:

- *start-tag* : `<` followed by a *NCNAME* and attributes; an attribute is a key-value pair, the value being within quotes separated by an equal sign; it is terminated by `>`.
- *end-tag* : `</` followed by the same *NCNAME* as its corresponding *start-tag* terminated by `>`; no attributes are allowed within the *end-tag*.
- *empty-tag*: similar to a *start-tag*, but terminated by `/>` without a corresponding *start-tag*.

We show a regular expression that matches an XML tags, skipping attributes in start-tag, but capturing the *content* between corresponding *start-tag*s and *end-tag*s. As this expression is quite involved, we define it using an *extended regex literal* which allows commenting *subtleties*.

```
let xml_tagX = #/
   <                          # begin of start-tag
    (?<name>[a-zA-Z_][-a-zA-Z_0-9.]*) # save name
    \s*(?:.*?)                # skip attributes
    (?:/>                     # empty tag
     |
       >                      # end of start-tag
       (?<content>.*?)        # content
       </\k<name>>)           # end-tag
/#
```

In this expression, we take for granted that no nested XML tags of the same name exist. This case will be dealt later in the document.

We can use this to remove XML tags from a string. This example shows how we keep only the content group of the first two tags; we remove the third empty tag, but the last tags do not match because their start-tag and end-tag names are different.

```
let xml_string = "<_a>xx</_a> <b.1 c='d' e='f'>yy</b.1> <w-90/> <good>content</bad>"
xml_string.replacing(xml_tagX){$0.content ?? ""}
// => xx yy  <good>content</bad>
```

In this section, we have shown how to achieve in Swift what regular expressions can do in other programming languages. Now we describe a way of writing regular expressions that sets Swift apart and allows many variations and combinations while keeping the strong typing discipline.

3. RegexBuilder

In addition to the Regex literal notation shown in the previous section, Swift provides an alternative notation based on the *overloaded* `Regex` constructor which accepts different kinds of parameters. Most often it is a closure (written as a *trailing* closure) that returns a `RegexComponent` created with the *Result Builder* notation of Swift. See this document for an introduction to this original notion for *Domain Specific Languages* (DSL) similar to SwiftUI code. This provides a more readable and type-safe notation for regular expressions and it also allows a systematic composition of regular expressions.

A `RegexBuilder` expression combines simple strings and other regexes by concatenation combined with components such as `CharacterClass`, `LookAhead` or `ChoiceOf` and quantifiers such as `Optionally` or `ZeroOrMore`.

To use this notation, the `RegexBuilder` module must be imported. The regex of an *identifier* can now be rewritten as

```
import RegexBuilder
let identifierRB = Regex {
    CharacterClass(("A"..."Z"),("a"..."z"))
    ZeroOrMore(.word)
}
```

which is more verbose, but more readable and maintainable. Now `identifierRB`, whose type is `Regex<Substring>` can be used as a any regular expression literal.

Xcode provides a refactoring tool to transform a `Regex` literal into a `RegexBuilder` expression. Because `RegexBuilder` allows constructions that cannot be written as a literal string, an automatic tool to transform a `RegexBuilder` expression to a regex literal is not available.

3.1. Capturing Information

We now define an alternative regex for the example used in the named capture group example for parsing key-value pairs, values being optional. It will be built from simpler expressions. Named captures are obtained through `Reference` in this context and serve for *subscripting* the resulting match i.e., using square brackets. A reference is not a property name as it is the case for regex literals. A reference is a value created with the *Reference* constructor called with a *type* as parameter, hence the `.self` after the type name. One important feature is that the resulting value can be a transformation of the matched substring, which may be of a different type than a substring.

First the regex for the key with its *captured* reference whose type is `Regex<(Substring, Substring)>`:

```
let key = Reference(Substring.self)
let kPatRB = Regex {
    Capture(as: key) {
        identifierRB
        //  /[A-Za-z]\w*/ could also have been used instead of the lines above
    }
}
```

The regex for the value to be *transformed* into an integer. In principle, the `Int` constructor could fail (but not in this case as the substring only contains digits. The type of `vPatRB` is `Regex<(Substring, Int)>`.

```
let value = Reference(Int.self)
let vPatRB = Regex {
    Capture(as: value) {
        OneOrMore(.digit)
    } transform: {Int($0)!}
}
```

With these definitions, we can now build a key-value regex, which combines the pattern for capturing the key and optionally parses the equal sign with surrounding spacing and captures the value. Usually elements in a `ResultBuilder` are put on separate lines like Swift constructs but here we use semicolons to separate some of them on the same line for compactness. The type of `kvPatRB` is now `Regex<(Substring, Substring, Int?)>`. The trailing `?` indicates that the integer value is optional.

```
let kvPatRB = Regex {
    kPatRB
    Optionally {
        ZeroOrMore(.whitespace); /=/; ZeroOrMore(.whitespace)
        vPatRB
    }
}
```

The list of all values in the subject is obtained like the following (to be compared with the literal regex version). A check is made that the value part is present and if so its integer value is obtained by subscripting.

```
keyValues.matches(of: kvPatRB).map{$0.2 != nil ? $0[value] : 1}
   // => [3, 1, 5, 1, 10]
```

Note the subscripting within the match using the `Reference` variable `value`. The result does not have to be unwrapped because the `Regex` transform has already performed the conversion from the `Substring` to an `Int`.

If a capture is not given a *name*, its substring or value is referenced using indexing like an unnamed capture in a regex literal. Throwing an error from a `transform` closure aborts matching and propagates the error out to the caller, if this is not what is wanted `TryCapture` can be used as a transformation that can fail, where a `nil` result forces backtracking within the regex matching process.

To match a previously captured string within the same regular expression is only a matter of using the name of the captured value as a `RegexComponent`. Here is a version of our previous example of matching an XML tag using the `RegexBuilder` notation.

```
1  // letter or underscore followed by letter, digit, underscore, hyphen or period
2  let nc_name = /[a-zA-Z_][-a-zA-Z_0-9.]*/
3
4  let tag_name = Reference(Substring.self)
5  let content = Reference(Substring?.self)
6
7  let xml_tag_RB = Regex {
8      "<"     // begin of start-tag
9      Capture (as: tag_name) {nc_name}
10     /.*?/   // skip attributes
11     ChoiceOf {
12         "/>"    // empty-element tag
13         Regex {
14             ">"   // end of start-tag
15             /.*?/
16             Capture (as: content) { /.*?/ } transform: {$0}
17             Regex{"</" ; tag_name ; ">"}  // end-tag
18         }
19     }
20 }
```

Line 2 defines a regex literal for the *NCNAME* which is used on line 9 where its value is captured. The captured value is used in the Regex on line 17. As a matched tag does not necessarily have content, the `Reference` on line 5 is marked as *Optional*. Because the two alternatives of `ChoiceOf` must have type `String`, a transform closure is used on line 16 to create a `String` from the captured value.

This regex can be used like this

```
xml_string.replacing(xml_tag_RB){$0[content] ?? ""}
// => xx yy  <good>content</bad>
```

We thus see that the `RegexBuilder` notation is more versatile, readable and compositional than the literal one.

3.2. *Foundation* Parsers

Swift's regexes allow combining regular expressions with existing parsers for commonly occurring strings, such as URLs, locale-dependent numbers, dates and currencies. These are called *Foundation parsers* in the Swift terminology. These *industrial strength* parsers can be used like any other regular expression component and return properly typed values. Such specialized parsers, that are often error prone to develop, are more efficient than regular expression interpretation. The next section will show that these parsers are merely implementing a protocol that users can follow for implementing their own parsers.

3.2.1. Customizing Foundation Parsers

- **Matching a Date**

The API defines 6 methods to match different ways of writing a date and capturing it as a `Date` object. We choose one in which the format is specified by a string. It is also possible to match an ISO 8601-formatted date string. As each method follows the convention of a given locale, it is a very flexible tool.

```
let date = Reference(Date.self)
let dateRB = Regex{
    Capture (as:date){
     .date(format:"\(day:.defaultDigits)/\(month:.defaultDigits)/\(year:.defaultDigits)",
           locale: Locale(identifier: "fr_CA"),
           timeZone:.current)
    }
}
```

The Swift interpolated string for the `format:` parameter declares the field names of the `Date` object followed by a writing specification (here `.defaultDigits`). It drives the matching process to create the date.

- **Matching a Currency**

Matching a currency is specified by locale properties and the result can be specified either as an `Integer`, in which case the *cents* are ignored. To deal with cents, we must use a Decimal number which is a Swift structure representing a base-10 number with its own arithmetic operators.

```
let price = Reference(Decimal.self)
let priceRB = Regex {
   Capture (as:price){
       .localizedCurrency(code: Locale.Currency("CAD"),
                          locale: Locale(identifier: "fr_CA"))
    }
}
```

- **Matching a URL**

Matching a url , which is a quite elaborate regex, creates a structure with the usual fields such as `scheme`, `host`, `part`, `query`...

```
let link = Reference(URL.self)
let linkRB = Regex {Capture (as:link){.url()}}
```

3.2.2. Using Foundation Parsers

We now show an example of use of these parsers to process strings representing orders to an online store. The following string will be used as an example subject for a regular expression that combines Swift predefined parsers.The order contains a localized date and currency, using the French Canada locale, followed by a URL. These fields are separated by a colon with spacing around it.

```
let order = "19/2/2024 : Computer Screen : 258,92 $ : https://www.azamon.ca/gp/aw/d/B0CJVK87Y7/?ref_=sbx_be_s_spar
```

The order for a computer screen was placed on *February 19th, 2024* and costed *$258.92* followed by a URL describing the item. The format of the date and the currency in the subject are written according to the writing convention for French in Canada.

- **Separating fields**

The ordered item is a list of characters, while a separator is a colon with some spacing around it. Using a quantifier such as `ZeroOrMore` can sometimes lead to some inefficiencies because the regex engine might have to backtrack a few times with different starting points. But in many cases, such as in the separator here, once a separator has been matched, we are sure of the choice and we can avoid any backtracking over this choice by marking it `Local`.

In other regex formalisms, this is called an *atomic* or *non-backtracking* group indicated by `?>` which is also allowed in Swift. This idea is similar to the `FENCE` pattern in SNOBOL4 or the cut `!` in Prolog. For example, the regular expression `a(bc|b)c` (capturing group) matches `abcc` and `abc`, but `a(?>bc|b)c` matches `abcc` but not `abc`, because once it has matched `bc`, the remaining choice for `b` is lost because of the local marking. Such `Local` or *atomic* group does not create a capture and thus does not add a component to the type.

```
let item = Reference(Substring.self)
let itemRB = Regex {Capture (as:item){OneOrMore(.any)}}

let sep = Regex{
    Local{ZeroOrMore(.horizontalWhitespace); ":"; ZeroOrMore(.horizontalWhitespace)}
}
```

- **Matching the Order**

Matching the complete order is now only a matter of composing the previous regexes with embedded separators.

```
let orderRB = Regex{
    dateRB ; sep ; itemRB ; sep ; priceRB ; sep ; linkRB
}
```

- **Creating an Invoice**

We now create an invoice from the captured values by matching the subject in the `orderRB` regex.

We define an English locale aware format for a `Date` object.

```
let en_CA_date = Date.FormatStyle()
    .year()
    .day(.defaultDigits)
    .month(.defaultDigits)
    .locale(Locale(identifier: "en_CA"))
```

We want `Decimal` numbers displayed with 6 digits for the dollar part and two for the cents, but this format adds leading 0 and spaces. We define a function to format the value and use a regex (of course...) to replace leading 0 and spaces by spaces and add a dollar sign.

```
let decimalFormat = Decimal.FormatStyle(locale: Locale(identifier: "fr_CA"))
    .precision(.integerAndFractionLength(integer: 6, fraction: 2))
func fmt(_ val:Decimal)->String{
    val.formatted(decimalFormat)
       .replacing(/^[\s0]*/){String(repeating:" ",count:$0.count)}+" $"
}
```

We can now apply the `orderRB` to the subject `order` and extract captures. The sales tax rate for Québec in computed and added as a `Decimal` number. A multi-line literal string in which the captured and computed values are then printed.

```
let m = order.firstMatch(of: orderRB)!
let m_price = m[price]
let taxes = m_price * Decimal(0.14975) // Québec sales rate is 14.975%
let total = m_price + taxes
print("""
On \(m[date].formatted(en_CA_date)),
you ordered a \(m[item])
   Price: \(fmt(m_price))
   Taxes: \(fmt(taxes))
   Total: \(fmt(total))
Thank you
\(m[link].host!)
""")
```

To produce the following in which the date (month/day/year) is now formatted according to the English locale.

```
On 2/19/2024,
you ordered a Computer Screen
   Price:     258,92 $
   Taxes:      38,77 $
   Total:     297,69 $
Thank you
www.azamon.ca
```

3.3. Custom RegexComponent

We now show how to build a specialized matcher and use it like a regex. This approach relies on implementing the `CustomConsumingRegexComponent` protocol with the `consuming` function that receives a string, a starting index and bounds to work within. When the function considers it has a match, it returns a pair whose first value is the index following the end of the match, the second value being the matched substring. The function returns `nil` when no match is found. This is the protocol implemented by *Foundation Parsers* used in the previous section.

3.3.1. Matching Balanced Parentheses

For illustrating a custom regex component, we define a matcher for a well parenthesized expression, similar to the predefined `BAL` pattern in SNOBOL4. This is a classical example of a pattern that cannot be written using a *formal* regular expression. Some programming languages allow the definition of *recursive* regular expressions, but Swift does not; we cannot refer to a regex within itself.

In this function, when the match begins with an open parenthesis, the `level` variable is set to 1. The function iterates over the characters of the string decrementing `level` when a close parenthesis is encountered and incrementing when an open parenthesis is seen. A match is found as soon as `level` reaches 0 and a pair is returned containing the index of the next character and the substring between the start and current indices. Should this match fails, the *global* regex engine calls it at another starting position.

```
struct BalancedParentheses: CustomConsumingRegexComponent {
    typealias RegexOutput = Substring
```

```
    func consuming(_ input: String,
                   startingAt index: String.Index,
                   in bounds: Range<String.Index>)
    throws -> (upperBound: String.Index, output: Substring)? {
        guard index < input.endIndex && input[index]=="(" else {return nil}
        var level=1
        var pos = input.index(after: index)
        while pos != input.endIndex {
            if input[pos] == ")" {
                level -= 1
                if level == 0 {
                    return (input.index(after:pos),input[index ... pos])
                }
            } else if input[pos] == "(" {
                level += 1
            }
            pos = input.index(after:pos)
        }
        return nil
    }
}
```

Here are some tests returning a list of the *balanced parenthesized* substrings within a subject.

```
let bal = BalancedParentheses()
"(2+(3+4)) )(((()))".matches(of: bal).map{"\($0.output)"}
// => ["(2+(3+4))", "(())"]
" (2+(3+4)())+( abc (1+2) ".matches(of: bal).map{"\($0.output)"}
// => ["(2+(3+4)())", "(1+2)"]
```

3.3.2. Matching Nested XML Tags

The next example is a custom `RegexComponent` for matching nested XML tags, building on our previous example, but defining them as `RegexBuilder` expressions instead of literals.

```
let xml_attr = Regex {  // an attribute
    let quotesym = Reference(Substring.self)
    /\s+/
    nc_name             // attribute name
    /\s*=\s*/
    Capture (as:quotesym) {/["']/} // start quote ' or "
    /.*?/               // attribute value
    quotesym            // ending quote same as start
}

let xml_tagN = Regex {
    Capture (as:tag_name){nc_name} // tag name
    Regex {ZeroOrMore {xml_attr}   // followed by 0 or more attributes
        /\s*/
    }
}

// define three types of tags
let xml_start_tag = Regex {"<"  ; xml_tagN ; ">" }
let xml_empty_tag = Regex {"<"  ; xml_tagN ; /\/>/ }
let xml_end_tag   = Regex {"</" ; xml_tagN ; ">" }
```

With these definitions, we can define a custom `RegexComponent` that uses a global stack (line 1) for keeping track of open tags with their starting index. When an end tag of the same name as the one on the top of the stack is encountered, it returns a match containing the portion of the subject between the starting position and the end of the current match. An empty tag is considered as balanced. In the case of nested XML tags, both inner and outer tags are matched. Errors are raised for badly nested tags or for a `<` not followed by a tag.

```
1  var tags = [(Substring,String.Index)]()   // stack of (start-tag-name, string index of start)
2
3  struct NestedXML: CustomConsumingRegexComponent {
4      typealias RegexOutput = Substring
5      func consuming(_ input: String,
6                     startingAt index: String.Index,
7                     in bounds: Range<String.Index>)
8      throws -> (upperBound: String.Index, output: Substring)? {
9          guard index < input.endIndex else {return nil}
10         var pos = index                          // current position
11         while let m = input[pos...].firstMatch(of: /(?><)/) { // skip to before the next <
12             let start = m.range.lowerBound
13             if let m = input[start...].prefixMatch(of: xml_end_tag){ // end-tag encountered
14                 if m[tag_name] == tags.last!.0 { // check if the tag-name is the same as the top of the stack
15                     let (_,tag_start) = tags.popLast()! // remove it
16                     return (m.range.upperBound,input[tag_start ..< m.range.upperBound])
17                 } else {                 // should never happen (bad nesting of tags)
18                     fatalError("Bad XML: \(m[tag_name]) should match \(tags.last?.0 ?? "strange tag")")
19                 }
20             } else if let m = input[start...].prefixMatch(of: xml_empty_tag){ // empty tag
21                 return (input.index(after: start),input[start ..< m.range.upperBound]) // return it
22             } else if let m = input[start...].prefixMatch(of: xml_start_tag){ // start tag
23                 tags.append((m[tag_name],start))               // add it to the stack
24                 pos = input.index(start, offsetBy: m.0.count)   // update position
25             } else {      // should never happen: < not followed by a tag
26                 fatalError("no match of tag: \(start) : \(input[start...])")
27             }
28         }
29         return nil
30     }
31 }
```

This function can be called as follows to print all balanced XML elements within a multi-line string containing nested XML tags.

```
1  let doc = """
2  <a d="f" w="e"> hello </a>
3  <p><p>info</p></p>  nothing <q><p>test</p></q> <x z='2' />
4  <z-1> a value spanning
5  two lines</z-1>
6  <b><c>good</c>friends</b>
7  <d><e/></d>
8  """
9
10 doc.matches(of: NestedXML()).forEach{print("\($0.output)")}
11 // output
12 <a d="f" w="e"> hello </a>
13 <p>info</p>
14 <p><p>info</p></p>
15 <p>test</p>
16 <q><p>test</p></q>
17 <x z='2' />
```

```
<z-1> a value spanning
two lines</z-1>
<c>good</c>
<b><c>good</c>friends</b>
<e/>
<d><e/></d>
```

4. Run-time Regular Expressions

In the previous sections, the regular expression was defined in Swift code, which allowed its static typing. However, in some cases, a regular expression must be created from a string provided by the user, read from a file, or generated on the fly. This is how regexes are defined in most other programming languages, including ones advocating a strong typing discipline, such as Java. In these cases, the regular expression syntax can be checked when it is *compiled*, which occurs at run-time, though. Swift also allows this mode of definition of regexes.

To create a regular expression from a `String`, we use the `Regex` constructor with a string as parameter. When it is a string literal, such as `"a(bc|de)f|ghi"` or `"\".*?\""`, care must be taken to escape special characters such as in the second expression where the quotes that delimit standard Swift strings must be matched. To reduce the number of characters that need to be escaped, *extended string delimiters* can be used. The second example can thus be written as `#"".*?""#`.

But the creation of a regex from an arbitrary string may raise an error in the case of a malformed pattern. It is thus necessary to embed the call to the `Regex` constructor within a `try` block. If we are confident the regex is well formed, then using `try!` creates a regex that can be used directly, such as the following:

```
1  let exprS  = try! Regex("a(bc|de)f|ghi")
2  let quoteES = try! Regex(#"".*?""#)
```

These regular expressions can be used with the `String` methods `.contains(..)` or `.startsWith(..)` which return a boolean result (see Line 2 in the following example).

The result of a matching method for a run-time regex is an object of the *erased* type `AnyRegexOutput` (see Line 3) in a way similar to the result of a regex match in other programming languages such as Java or Python.

The match result gives access to the `range` of the whole match, but its `output` property is an array of *Elements* corresponding to the capture groups. *Subscripting* (e.g. `[n]`) is used to retrieve the value of a capture group. This differs from the index (e.g. `.n`) used for accessing fields of a tuple in the case of statically typed regexes. We can use `output[0]` to get information about the whole match or `output[n]` to get information about the capture group *n*. The `count` property indicates the number of capture groups plus 1.

Each *element* of the `output` property has the `substring` property to get the match substring of this group, `range` for its range and `name` to get the name of the capture group, `nil` if it does not have a name. Here are a few examples of access to the result of matching a run-time regex.

```
1  let subj1 = "labcf ghk ghi def ccc"
2  subj1.contains(exprS)        // => true
3  subj1.firstMatch(of: exprS)  // => Optional(Match(anyRegexOutput: _StringProcessing.AnyRegexOutput(...))
4  subj1.firstMatch(of: exprS)?.range // positions 1...5 in subj1
5  subj1.firstMatch(of: exprS)?.count // 2 because there one capture group
6  subj1.firstMatch(of: exprS)?.output[0].substring // abcf
7  subj1.firstMatch(of: exprS)?.output[0].range // positions 1...5 in subj1
8  subj1.firstMatch(of: exprS)?.output[0].name  // nil
```

4.1. Specifying the Expected Type

Whenever possible, the generic parameters for the `Regex` constructor should be specified as in the following examples so that the compiler can detect some potential errors at compile time. In the following examples, we could have used literal regexes in which case, the compiler would have inferred the appropriate types, but we use literal strings instead of string variables in the constructor calls for simplification.

Line 1 shows a case where the regular expression matches a substring. The type of the expression in line 2 is a pair: the parentheses, used to change the priority of alternation over concatenation, also create a typed capture group that is combined with the substring for the whole match. The second type of the pair is marked as *Optional* because the group appears on the left of the alternation with `ghi`. In line 3, the group is marked as *not captured* by prefixing it with `?:`, so its type is not added in the signature; `?:` groups should be specified when a given group is not needed as it simplifies the type.

```
let quoteESt = try! Regex<Substring>(#"".*?""#)
let exprSt = try! Regex<(Substring,Substring?)>("a(bc|de)f|ghi")
let exprSt1 = try! Regex<Substring>("a(?:bc|de)f|ghi")
```

It is also possible to specify the type as a parameter of the `Regex` constructor, so the following examples are equivalent to the preceding ones. Note the use of `.self` to refer to the type.

```
let quoteESta = try! Regex(#"".*?""#,as:Substring.self)
let exprSta = try! Regex("a(bc|de)f|ghi",as:(Substring,Substring?).self)
let exprSt1a = try! Regex("a(?:bc|de)f|ghi",as: Substring.self)
```

As we have specified that the type of the result of the match is a substring, then using the result of the match is similar to what we have shown in the previous sections for literal regexes and regex created by calls to the `RegexBuilder`. The result of the call to `firstMatch` is still an optional because the subject might not have an occurrence that matches the pattern, this explain unwrapping the result in line 1.

```
subj1.firstMatch(of: exprSt1)!   // => "abcf"
subj1.matches(of: exprSt).map{"\($0.0)"} // => ["abcf","ghi"]
```

This section shows that although it is possible to use strings to define regular expressions, it is simpler and more reliable to use regex literals or `Regexbuilder` expressions whenever this is possible because type checking occurs when the program is compiled and not at run-time. Moreover, access to the components of the match is simpler when the types are specified.

4.2. Using Run-Time Regexes

We now illustrate a use of a run-time regex in a struct for replacing French words with their corresponding English word. This struct is initialized with a dictionary (line 5) and a regex which is an alternation joining keys of the dictionary (line 7) separated by `|` , As we want to match complete words, the alternation must be enclosed by word boundaries `\b` (line 8). As the alternation is the only expression in this regex, it is not necessary to capture the result, so `(?:…)` is used. Note the use of the *raw string* notation between `#"` and `"#` which avoids escaping backslashes. But then to use string interpolation, we need to use `\#(…)` instead of `\(…)`. As the built regex always has the same form, we can specify its type `Regex<Susstring>` when it is declared (line 3) and does not need to be repeated when the regex is created (line 8).

The `replacing(in:...)` method (lines 11-13) replaces matches of created regex on a string by the corresponding value of the original dictionary.

Line 16 builds a struct from a simple dictionary of french and English words. Line 17 makes the replacement of the words of the dictionary within a string. Note that `jour` is not replaced because is does not occur at the word boundary.

```
struct WordsReplace{
    let dict:[String:String]
    let dictRE:Regex<Substring>

    init(_ dict:[String:String]){
        self.dict = dict
        let reS = dict.keys.joined(separator: "|")
        dictRE = try! Regex(#"\b(?:\#(reS))\b"#)
    }

     func replacing(in str: String)->String {
        return str.replacing(dictRE, with: {dict[String($0.output)]!})
    }
}

let fr2en = WordsReplace(["jour":"day", "monde":"world", "joyeux":"happy", "triste":"sad"])
fr2en.replacing(in:"bonjour le monde joyeux et rarement triste")
```

```
// => "bonjour le world happy et rarement sad"
```

5. Use Cases

This section presents some compelling uses of Swift regexes. First a simple example of parsing Roman numerals using either a regex literal or a `RegexBuilder` expression. Then a more elaborated example of a tokenizer that combines literal regexes within a `RegexBuilder` expression. Finally we show how dynamic regexes can be created from a JSON file for building the core of an ELIZA-like chatbot. The complete source file of these examples are available on the companion web site.

5.1. Parse a Roman Numeral

To illustrate the use of Swift regex in a real-world scenario, we will now demonstrate a regular expression that can be used to convert a Roman numeral string into its decimal equivalent. Roman numerals use letters to represent numbers: `M`:1000, `D`:500, `C`:100, `L`:50, `X`:10, `V`:5 and `I`:1. Up to three *letters* can appear following another to add their value to the previous one. If a lower-valued unit appears before a higher-valued one, it is deducted.

5.1.1. With a regex literal

Roman numerals in the range [0,4000) can be matched using the following regex *muli-line* literal which applies four optional regexes on the string. The first regex (line 2) matches one to three `M`, while regexes for hundred, tens and units follow the pattern of line 5 replacing `I`, `V`, `X` by `X`, `L`, `C` and by `C`, `D` and `M` respectively. This pattern is an alternative between:

- an `I` followed by an `X`, a `V` or up to 2 other `I`;
- a `V` followed by unto three `I`.

The regexes on lines 3 to 5 could have been also written in the form `(I|II|III|IV|V|VI|VII|VIII|IX)?` but this would entail more backtracking, unless the regex compiler is very clever. We prefer writing the tree-based form which seems very clear anyway.

```
1  let romanRE = #/
2      (M{1,3})?                 # thousands
3      (C(?:M|D|C{,2})|DC{,3})?  # hundreds
4      (X(?:C|L|X{,2})|LX{,3})?  # tens
5      (I(?:X|V|I{,2})|VI{,3})?  # units
6  /#
```

These spans of letters correspond to values to be added to determine the overall value:

```
1  let romanVals = [
2      "C": 100, "CC": 200, "CCC": 300, "CD": 400, "CM": 900,
3      "D": 500, "DC": 600, "DCC": 700, "DCCC": 800,
4      "I": 1, "II": 2, "III": 3, "IV": 4, "IX": 9,
5      "L": 50, "LX": 60, "LXX": 70, "LXXX": 80,
6      "M": 1000, "MM": 2000, "MMM": 3000,
7      "V": 5, "VI": 6, "VII": 7, "VIII": 8,
8      "X": 10, "XC": 90, "XL": 40, "XX": 20, "XXX": 30
9  ]
```

With these definitions, the value of a string corresponding to a roman numeral can be obtained with the following function that matches the whole string (line 2). The values of each captured string that appears to the result (lines 5-8) are then added. Note the use of the optional captured groups that must be unwrapped done here with the `if let` construct; this value is a substring transformed to string by string interpolation is used as a key for the `romanVals` dictionary; as indexing a dictionary can in principle also return nil, the result of the indexing must also be unwrapped. When the whole string cannot be matched, the string does not correspond to a valid roman numeral (lines 11,12).

```
func parseRomanRE(_ s:String)->Int? {
    if let m = s.wholeMatch(of: romanRE){
        var res = 0
        if let v = m.output.1 {res += romanVals["\(v)"]!}
        if let v = m.output.2 {res += romanVals["\(v)"]!}
        if let v = m.output.3 {res += romanVals["\(v)"]!}
        if let v = m.output.4 {res += romanVals["\(v)"]!}
        return res
    }
    return nil
}
```

Note that it is not possible to use an integer index for the captures because the result of the match is a tuple. So we deal with them separately.

5.1.2. With a RegexBuilder Expression

We now illustrate how this approach to Roman numeral parsing can be implemented using a `RegexBuilder`. As the regex for units, tens and hundreds follow the same pattern, we use a function to define a pattern parametrized by strings for the unit, the five and the ten. It returns (line 7) the integer (already unwrapped in the transformation) corresponding to the parsed string.

```
func makeRB(_ i: String,_ v:String, _ x:String)->Capture<(Substring, Int)>{
    return Capture {
                ChoiceOf {
                    Regex {i ; ChoiceOf { x; v ; Repeat(...2) { i }}}
                    Regex { v; Repeat(...3) { i }}
                }
            } transform:{romanVals["\($0)"]!}
}
```

`makeRB` is used to define the regex for all components.

```
let romanRB = Regex {
    Optionally {Capture { Repeat(1...3){"M"} } transform: {str in 1000*str.count}}
    Optionally {makeRB("C","D","M")}
    Optionally {makeRB("X","L","C")}
    Optionally {makeRB("I","V","X")}
}
```

This function can be used to parse a string and return the corresponding value by adding the returned value by each optional regex when it exists, 0 otherwise (line 4). Compare this with `parseRomanRE` above.

```
func parseRomanRB(_ s:String)->Int? {
    if let m = s.wholeMatch(of: romanRB){
        let out = m.output
        return (out.1 ?? 0) + (out.2 ?? 0) + (out.3 ?? 0) + (out.4 ?? 0)
    }
    return nil
}
```

5.2. Developing a Tokenizer

We will now demonstrate how to create a tokenizer that classifies substrings using regular expressions. This is typically the first stage of compiling, but it can also be used in other text-processing applications. This tokenizer is a Swift implementation of an example from the Python documentation, but in this case, we combine literal regexes with a RegexBuilder expression to get the best of both worlds.

5.2.1. Defining a Token

A `Token` is a structure (lines 13-19) with many alternatives (`kinds`) defined by an `enum` with associated values (lines 1-11) to classify each span of text. In some cases (`ID`, `KEYWORD`, `OP`), the text span is kept with the token; if it is a `NUMBER`, it is converted to a numeric value. The line and column positions of the start of the token (line 15) are also saved, which is useful for error messages or for languages that take indentation into account. Line 16 specifies the format for displaying a `Token`: the `kind` followed by line and column numbers within square brackets.

```
1  enum kinds {
2      case NUMBER (Double)
3      case ASSIGN
4      case END
5      case ID (Substring)
6      case KEYWORD (Substring)
7      case OP (Substring)
8      case SKIP
9      case NEWLINE
10     case MISMATCH
11 }
12
13 struct Token:CustomStringConvertible {
14     let kind: kinds
15     let line,column: Int
16     public var description:String {
17         "\(kind) [\(line),\(column)]"
18     }
19 }
```

5.2.2. Creating a Pattern

The association of a portion between a matched substring by a `regex` and a `Token` can be defined using the following Swift construct.

```
1  Capture {regex} transform:{Token(kind:…, line:…, column:… )}
```

To simplify the notation, we define `pat`, a function to create such associations. It has two parameters: a regex and a closure to define the kind of token to create. `lineNumber` is a global variable (line 1) maintained by the tokenizing process and `colPos` is function (lines 5-8) giving the starting position of the matched substring in the subject. `pat` (lines 10-15) returns a `Capture` with the matched substring and the *transformed* `Token`. Since the closure will be executed after the `pat` function returns, it must be annotated with `@escaping` (more information about this annotation).

```
1  var lineNumber:Int
2
3  // find the position as an integer of a substring within its original/base string
4  // columns are numbered from 1
5  func colPos(_ s:Substring) -> Int {
6      let base = s.base
7      return base.distance(from: base.startIndex, to: s.startIndex)+1
8  }
9
10 func pat(_ regex:Regex<Substring>,
11           kindClosure: @escaping (Substring)->kinds) -> Capture<(Substring,Token)>{
12     return Capture {regex} transform:{
13         Token(kind: kindClosure($0), line:lineNumber, column: colPos($0))
14     }
15 }
```

5.2.3. Using the Patterns

We can now construct a RegexBuilder expression for each token type, with each line being a call to `pat` with a literal regex as first parameter and a trailing closure returning the appropriate kind of the token depending on the matched substring `s`. Line 2 demonstrates the conversion of the string into a numeric value. The substring matched by line 5 can either be an `ID` or a `KEYWORD`, depending on whether it appears in the set of predefined keywords (line 14). An underscore is given as a parameter to the closure when the value of the matched string is not needed. If none of the first seven patterns match, it returns a `MISMATCH`.

```
1  let tokenSpecifications =  ChoiceOf {
2      pat(/\d+(?:\.\d*)?/){s in .NUMBER(Double(s)!)}  //  .1  number
3      pat(/:=/)           {_ in .ASSIGN}              //  .2  assignment
4      pat(/;/)            {_ in .END}                 //  .3  end of statement
5      pat(/[A-Za-z]\w*/)  {s in                       //  .4  identifier or keyword
6                             return keywords.contains(String(s)) ? .KEYWORD(s) : .ID(s)}
7      pat(/[+\-*\/]/)     {s in .OP(s)}               //  .5  arithmetic operator
8      // ignored values
9      pat(/[ \t]+/)       {_ in .SKIP}                //  .6  spaces
10     pat(/$/)            {_ in .NEWLINE}             //  .7  end of line
11     pat(/./)            {_ in .MISMATCH}            //  .8  error
12 }
13
14 let keywords:Set = ["IF", "THEN", "ENDIF", "FOR", "NEXT", "GOSUB", "RETURN"]
```

These pairs of regex and token kinds could be extended to include other tokenization features, such as end-of-line comments, literal strings, parentheses, brackets, braces, etc.

The inferred type of `tokenSpecifications` is

```
1  ChoiceOf<(Substring, Token?, Token?, Token?, Token?, Token?, Token?, Token?, Token?)>
```

a tuple containing the matched substring, followed by eight optional `Token`S, only one of which is non-nil because the `ChoiceOf` stops as soon as it finds a match, so properly ordering the `pat` calls is important. The access to the components of the resulting tuple is done with a number between 1 and 8, as indicated in the comments following each `pat` call. As contrarily to Python, Swift does not give access to the *last match number*, all possibilities will have to be checked.

5.2.4. Defining the Tokenizer

A tokenizer is typically invoked by a parsing routine that iteratively handles each token using a method that generates a new token on every invocation. This pattern is comparable to the `IteratorProtocol` which requires the definition of a `next()` method.

Here is a Swift `Tokenizer` structure taking a *program* string (line 5) to build a list of numbered lines kept as a list of pairs. It stores the current line in a property and updates it using the `static nextLine` method (lines 37-42). This method creates a new string from the first element of `lines` and removes it from the list. This ensures that the column numbers are relative to the beginning of the current line rather than of the entire program. The line number is included in the returned value.

The `next()` function (lines 12-35) first checks whether the current line is empty. If it is not, it retrieves the subsequent line. Otherwise, it returns `nil`. At line 20, the beginning of the line is identified, the outcome of the match is stored, and the matched substring is erased from the start of the line. If the result is `MISMATCH` (line 23), an error message is displayed, indicating the incorrect substring (in this case, a single character) along with the line and column numbers. The function `next()` is then called recursively (line 21) to search for a real token. If the result is either `NEWLINE` or `SKIP` (line 27), the function ignores it by calling `next()`. In all other instances (line 30), the *transformed* token is returned (line 23).

```
1  struct Tokenizer:IteratorProtocol {
2      let program:String
3      var lines:[(Int,Substring)]
```

```
4       var line:Substring  // current line
5
6       init(_ program: String){
7           self.program = program
8           self.lines = Array(zip(1...,program.split(separator:"\n")))
9           (lineNumber,line) = Tokenizer.nextLine(&self.lines)
10      }
11
12      mutating func next()->Token? {
13          while line.isEmpty {
14              if !lines.isEmpty {
15                  (lineNumber,line) = Tokenizer.nextLine(&self.lines)
16              } else {
17                  return nil
18              }
19          }
20          if let m = line.prefixMatch(of: tokenSpecifications) {
21              let out = m.output
22              line = line.dropFirst(out.0.count) // remove matched substring
23              if let _ = out.8 { // error message for mismatch and search next token
24                  print("\(out.0) unexpected at line \(lineNumber), column: \(colPos(out.0))")
25                  return next()
26              }
27              if let _ = out.6 ?? out.7 { // ignore newline, skip
28                  return next()
29              }
30              if let t = out.1 ?? out.2 ?? out.3 ?? out.4 ?? out.5 ?? out.6 {
31                  return t
32              }
33          }
34          return nil // should never happen
35      }
36
37      static func nextLine( _ lines:inout [(Int,Substring)])->(Int,Substring) {
38          // create new string for setting column numbers relative to this line
39          let (number,substr) = lines.removeFirst()
40          let str = String(substr)
41          return (number, str[str.startIndex..<str.endIndex])
42      }
43  }
```

5.2.5. Running the Tokenizer

The code below demonstrates how to use this tokenizer. First an instance of a `Tokenizer` is created (line 8) with the *program statements*. The tokenizer is used on lines 9-11 to print each returned tokens, but it could be seamlessly integrated into a more complex program. Lines 14-23 show an excerpt of the output.

```
1   let statements = #"""
2   IF quantity THEN
3       total := total + price * quantity;
4       tax := price * 0.05; #
5   ENDIF;
6   """#
7
8   var tokenizer = Tokenizer(statements)
9   while let t=tokenizer.next() {
10      print(t)
```

```
}

/*   output
KEYWORD("IF") [1,1]
ID("quantity") [1,4]
KEYWORD("THEN") [1,13]
ID("total") [2,5]
ASSIGN [2,11]
...
END [3,24]
# unexpected at line 3, column: 26
KEYWORD("ENDIF") [4,1]
END [4,6]
*/
```

5.3. Eliza-like Chatbot

Eliza in 1966 was one of the first programs that allowed a *conversation* between a human and a computer. It works by detecting patterns with placeholders in the user input, for which predetermined responses, usually questions, are already defined. Although the original implementation did not use regular expressions , Eliza-like programs have since been developed, which make extensive use of regexes. The source code gives a Swift implementation of a JavaScript version that illustrates run-time regexes created from patterns given in a JSON file (`elizadata.json`).

The heart of the matching process used by this program is explained in this section, which is data-driven by a script organized as shown in the following code block. The most important part of the data for the script are embedded lists associated with a keyword.

When a keyword is detected in the user input, the system selects a question associated with the first pattern that matches the input. In a question. A pattern is a string containing words and `*` used as a capture group similar to a `(.*?)` regex. In the associated questions, these groups are referred to by number in parentheses. For example, given the input `Often I remember coming to my house`, Eliza will produce the question `Do you often think of coming to your house ?` by matching *pattern-1* in which the star corresponds to *coming to my house* in which `my` is replaced by `your`.

```
[keyword, priority, [   | ["remember", 5, [
 [pattern-1,            |   ["* i remember *", [
   [question-11,        |       "Do you often think of (2) ?",
    question-12,        |       "Does thinking of (2) bring anything else to mind ?",
    ...]],              |       ...]],
 [pattern-2,            |   ["* do you remember *", [
   [question-21,        |       "Did you think I would forget (2) ?",
    question-22,        |       "Why do you think I should recall (2) now ?",
    ...]],              |       ...]],
 ...]                   |   ...
]                       |   ]
```

When a pattern contains an ampersand, it refers to a synonym defined elsewhere. For example the pattern in line 1 of the next code block, corresponds to the `/(.*?) i \b(desire|want|need)\b (.*?)/` regex. As this alternative corresponds to capture group, we see why the last part of the string is referred to by `(3)`. These regexes are generated when the JSON file is read when the program is launched.

```
["* i @desire *",["What would it mean to you if you got (3) ?",
                  "Why do you want (3) ?",
                   ...]],

let synons = {"desire":["want","need"],"sad":["unhappy","depressed","sick"]}
```

5.3.1. Eliza Class (`Eliza.swift`)

The properties of the `Eliza` class save the information from the `elizadata.json` file. Most of its properties are straightforward conversions from the JSON format. But the `keywords` property transformation is more complex because each list of lists shown above si transformed into a named tuple of the following type:

```
typealias Keyword = (priority:Int,
                     keywordRE:Regex<Substring>,
                     matchers:[(String)->String?])
```

`priority` is the JSON value and `keywordRE` a regex that matches the keyword string within two word boundaries. The `matchers` array consists of several functions that accept a string as input and try to match it against a specific pattern. Upon a successful match, the function returns a question with the placeholders replaced by substrings from the user's input.

The `makeMatcher()` method, an attribute of the Eliza class, demonstrates how to generate a matcher function from a pattern string, a list of question strings, and a dictionary of synonyms. The `Questions` class encapsulates a list of question strings and returns one of them sequentially on each invocation of `nextQuestion()`. Once all questions have been exhausted, it resumes at the beginning.

Lines 4-10 build a run-time regex from the pattern string: if it contains an ampersand, it expands the following word to a regex containing it and the alternatives from the `synons` dictionary; each star is replaced by a regex that matches any substring.

Lines 12–24 define a closure that matches the user input and returns a single question with the appropriate captured groups substituted. *Post-replacements* (e.g., `my` by `your`) are applied to the groups (line 19).

The conversion of a pattern into a regular expression occurs just once when the `makeMatcher()` function is invoked during startup. It does not happen again each time the resulting closure is used.

In the closure, `userInput` is compared to the `patternRE` regular expression (line 12). If a match occurs, a specific query text is chosen. Within this selected query, the corresponding substrings extracted from the "userInput" are substituted for the numeric placeholders enclosed in parentheses. It is important to note that the selection of the match output component (line 17) uses an index, as the regex on line 10 has type `Regex<(Substring,Substring)>`. On the other hand, the component is obtained by subscripting on line 18, since patternRE (line 10) is a run-time regex of type `Regex<AnyRegexOutput>`. Finally (line 21), the function replaces sequences of one or more spaces with a single space. If the input does not match the pattern regex (line 23), the function returns `nil`.

```
func makeMatcher(pattern:String,questions: Questions,synons:[Substring:[String]])
            ->((String)->String?) {
    // create a regex from the pattern
    var pattern = pattern
    if let m = pattern.firstMatch(of: /@(\w+)/) { // expand synonyms
        let synonsREs = m.1 + "|" + synons[m.1]!.joined(separator:"|")
        pattern.replace(m.0,with:#"\b(\#(synonsREs))\b"#)
    }
    pattern.replace(/\s*\*\s*/,with:#"(.*?)"#) // deal with *
    let patternRE = (try! Regex(pattern)).ignoresCase() // create regex
    return {userInput in        // returned closure
        if let m = userInput.wholeMatch(of: patternRE){
            var question = questions.nextQuestion()
            question.replace(/\((\d)\)/){i in
                let groupNo=Int("\(i.output.1)")!
                // replace (i) in the response by the ith capture of the input
                // to which the post-replacements are applied
                let replacement = String(m.output[groupNo].substring!)
                return posts.replacing(in: replacement)
            }
            return question.replacing(/\s+/,with:" ")
        }
        return nil
    }
}
```

5.3.2. Running the dialog

Within the `eliza-chat.swift` file, `makeQuestion(..)` (lines 1-14) is the main function that returns a question based on the user's text. To accomplish this, it breaks the text down at periods, processing each segment individually before recombining them. The appropriate keyword structure is found by `getKeyword()` (lines 16-18), which scans through the list of keywords looking for the first match to the `keywordRE` pattern.

When a particular keyword is encountered (line 6), the `reply` function (lines 20-31) is invoked. On line 22, the selected matcher function is applied to the input to generate the response. If the response mentions another keyword (line 23), the `reply` function is invoked recursively for that keyword (line 24). Otherwise, the response is returned directly. If no matching keywords were found, then nil is returned. However, this should never happen, as there should always be at least one pattern that matches unless there is a bug. However, this issue has arisen during development a few times.

Line 10 executes when no keyword is found in the input. Otherwise, line 12 performs post-processing on the full question.

```
1  func makeQuestion(_ text:String) -> String {
2      var question = ""
3      for part in unify(text).split(separator: "."){ // separate input in parts
4          let part = eliza.pres.replacing(in: String(part)) // preprocess the part
5          if !part.isEmpty, let keyword = getKeyword(part) {
6              question += reply(String(part),keyword)!+" "
7          }
8      }
9      if question.isEmpty {
10         return reply("x",eliza.keywords.first(where: {"xnone".contains($0.keywordRE)})!)!
11     } else {
12         return applyPostTrans(question) // apply post translation
13     }
14 }
15
16 func getKeyword(_ text:String)-> Keyword? {
17     return eliza.keywords.first(where: {text.contains($0.keywordRE)})
18 }
19
20 func reply(_ part:String,_ keyword:Keyword)->String? {
21     for matcher in keyword.matchers {
22         if let rpl = matcher(String(part)) {
23             if let m = rpl.wholeMatch(of: /goto (\w+)/){
24                 return reply(part,getKeyword(String(m.output.1))!)
25             } else {
26                 return rpl
27             }
28         }
29     }
30     return nil
31 }
```

5.3.3. Sample dialog

Here is a small dialog with Eliza with only two statements from the patient.

```
Is something troubling you ?
User: Often,I remember returning to my house.
Eliza: Do you often think of returning to your house ?
User: I need some help.
Eliza: Why do you want some help ?
This was a good session, wasn't it -- but time is over now.  Goodbye.
```

The source code shows a more complete script similar to the one in the paper of Weizenbaum (1966).

6. Conclusion

This document has described some original aspects of Swift regular expressions and provided illustrative instances of their use. Although it does not aim to be comprehensive, it should offer enough understanding for users to further investigate the Swift API. It also showcased three full-fledged examples: translating Roman numerals, breaking down a string into tokens and a pattern-matching based chat box. These examples demonstrate the cutting-edge features of Swift's regex abilities.

I hope that this text will be just as helpful to the reader as it was for me while writing it.

7. Appendix

7.1. Further reading

- **Links about Swift regexes:**
  - The *Swift-evolution proposals* 350, 351, 354, 355 and 357 although the implementation differs in details.
  - Videos presented at the *Apple Worldwide Developers Conference 2022* (WWDC 2022):
    - Meet Swift Regex that introduces the formalism
    - Swift Regex: Beyond the basics video illustrates more advanced parts (WWDC notes).
  - An online Swift regex tester is useful for testing some ideas, but it does not allow the use of Foundation parsers.
  - Other useful introductions:
    - Regular expressions in Swift: improve your text validations by using the new Regex APIs
    - Swift Regex Deep Dive
- **Links about regexes in general**
  - Syntax for regex literals using the ICU regular expressions or other programming languages namely Python.
  - A comprehensive website about regular expressions in many programming languages, but not Swift.

7.2. Swift API

- `Regex` structure
- RegexBuilder Framework
- `RegexComponent` protocol
- Regex related functions are distributed across many types; the most often used functions are described in the Bidirectional Collection protocol. As it can be difficult to get *authoritative* information, this browsable subset of the Regex API can be useful.
- Regular expressions used before Swift 5.7 : NSRegularExpression (they can also be used)

7.3. Source Files

Source files for the examples in this document

- RegexInSwift/RegexInSwift/main.swift
- RegexInSwift/RegexInSwift/Roman.swift (Section 5.1)
- RegexInSwift/RegexInSwift/Tokenizer.swift (Section 5.2)
- Eliza-Like Chatbot (Section 5.3)
  - RegexInSwift/RegexInSwift/eliza-chat.swift
  - RegexInSwift/RegexInSwift/eliza.swift
  - RegexInSwift/RegexInSwift/elizadata.json
- RegexInSwift.playground

7.4. Showing Matches

In the `main.swift` source file, we have defined the function `showMatches(of patternS:String, in subject:String)` which highlights with up arrows the matched characters within a string by a regular expression given as a `String`. It takes for granted that the subject is a single line. Here are a few examples of calls.

```
1 showMatches(of: "abc|def|ghi", in: "labc ghk ghi def ccc")
```

```
// output
Matching /abc|def|ghi/ 3 times
> labc ghk ghi def ccc
>  ↑↑↑     ↑↑↑ ↑↑↑

showMatches(of: "\".*\"",        in:#"abc "abcf" "gh"i"#)
// output
Matching /".*"/ once
> abc "abcf" "gh"i
>     ↑↑↑↑↑↑↑↑↑↑

showMatches(of:#"".*?""#,        in:#"abc "abcf" "gh"i"#)
// output
Matching /".*?"/ twice
> abc "abcf" "gh"i
>     ↑↑↑↑↑↑ ↑↑↑↑
```

It is also possible to call this overloaded function by giving it a `RegexComponent`, but in this case the string corresponding to the regular expression cannot be printed. This is the core function called by the preceding one.

```
" (2+(3+4)())+( abc (1+2) ".matches(of:BalancedParentheses()).map{"\($0.output)"})
// output
>  (2+(3+4)())+( abc (1+2)
>  ↑↑↑↑↑↑↑↑↑↑       ↑↑↑↑↑
```

We found these functions useful for learning and debugging purposes.

7.5. Showing Capture Groups

It can be useful for debugging to get the list of substrings matched by capture groups of a matching result, but the standard way that Swift prints substrings is difficult to interpret. The following function can be used to get a list of strings for the capture groups in the result match of a run-time regex.

```
func getGroups(_ output:Regex<AnyRegexOutput>.Match)->[String]{
    return (0 ..< output.count).map{"\(output[$0].substring!)"}
}
```

getGroups can also be used for a typed regex match result by converting it by calling the `getGroups(Regex.Match(...))` . The `Regex.match` constructor creates *type erased Match object*.

7.6. Swift Regex Cheat Sheets

7.6.1. Methods

Operation	Method
Check for a occurrence	*String* `.contains(` *Regex* `)->Bool` *String* `.starts(with:` *Regex* `)->Bool`
Find a match	*String* `.firstMatch(of:` *Regex* `)->Regex.Match?` *String* `.wholeMatch(of:` *Regex* `)->Regex.Match?` *String* `.prefixMatch(of:` *Regex* `)->Regex.Match?` *Regex* `.firstMatch(in:` *String* `)->Regex.Match?` *Regex* `.wholeMatch(in:` *String* `)->Regex.Match?` *Regex* `.prefixMatch(in:` *String* `)->Regex.Match?`
Find all matches	*String* `.matches(of:` *Regex* `)->[Regex.Match]`
Replace a match	*String* `.replacing(` *Regex* `, with:` *String* `)` *String* `.replacing(` *Regex* `){` *Closure* `}`
Change a string with a match	*String* `.replace(` *Regex* `, with:` *String* `)` *String* `.replace(` *Regex* `){` *Closure* `}`
Find ranges of a match	*String* `.firstRange(of:` *Regex* `)->Range?` *String* `.ranges(of:` *Regex* `)->[Range]`
Split a string with a regex	*String* `.split(separator:` *Regex* `)->[String]`
Get a string after removing a prefix	*String* `.trimmingPrefix(` *Regex* `)->[String]`
Remove prefix of string	*String* `.trimPrefix(` *Regex* `)->Void`

7.6.2. Frequently Used Metacharacters

Uppercase letters are the set inverse of the corresponding lowercase.

Escape sequence	Meaning	CharacterClass
`.`	any character	`any`
`\b` `\B`	word boundary	
`\d` `\D`	digit	`digit`
`\k<name>`	back reference a named capture	
`\p{..}` `\P{}`	any character with a unicode property	
`\s` `\S`	any white space character	`whiteSpace`
`\w` `\W`	any word character	`word`
`[..]`	any character in the set	`anyOf(...)`
`^`	beginning of line	
`$`	end of line	

`\N`	back reference a capture group by number `N`	
`\`	quote one of the following characters `\ * ? + [ ( ) { } ^ $ *`	

7.6.3. Operators in Literal and Keyword in RegexBuilder

Operator	Description	RegexBuilder
`|`	alternation	`ChoiceOf`
`*` `*?` `*+`	match 0 or more times (eager, reluctant, possessive)	`ZeroOrMore`
`+` `+?` `++`	match 1 or more times (eager, reluctant, possessive)	`OneOrMore`
`?`	match 0 or 1	`Optionally`
`( )`	capture group	`Capture`
`(?: )`	non-capture group	`One` *often omitted*
`(?<name>)`	named capture group	`Capture(as:...)`
`(?=...)`	match position before pattern	`Lookahead`
`(?!...)`	match position not before pattern	`NegativeLookahead`
`(?>...)`	create a Local (atomic) group without capture	`Local`
`{m,n}`	repeat previous match between m and n times, `m` is 0 and `n` is +∞ by default	`Repeat`

7.6.4. Access to the properties of the Match Object (`m`)

Literal regex, RegexBuilder expression or typed run-time regex: a Tuple

Operator	Description	Type
`m.output`	Substring matched	`Substring`
`m.range`	Interval of string indices spanning the match	`Range<String.Index>`
`m.count`	Length of match	`Int`
`m.N`	*N*th capture group	`Substring`
`m.name`	Named capture group	`Substring`

Untyped run-time regex (`Regex<AnyRegexOutput>.Match`) : an Array

Operator	Description	Type
`m.output[N].substring`	*N*th group	`Substring`
`m.output[name]?.substring`	regex with named capture group	`Substring`
`m.output.count`	number of capture groups	`Int`