7 Pattern Matching

Using patterns to find things in data

Terms defined: Chain of Responsibility pattern, **Document Object Model**, **Open-Closed Principle**, **base class**, **child (in a tree)**, **coupling**, **depth-first**, **derived class**, **eager matching**, **greedy algorithm**, **node**, **polymorphism**, **query selector**, **regular expression**, **scope creep**, **test-driven development**

We have been globbing to match filenames against patterns since Chapter 2. This lesson will explore how that works by building a simple version of the **regular expressions** used to match text in everything from editor and shell commands to web scrapers. Our approach is inspired by Brian Kernighan's[1] entry in [Oram2007].

Regular expressions have inspired pattern matching for many other kinds of data, such as **query selectors** for HTML. They are easier to understand and implement than patterns for matching text, so we will start by looking at them.

7.1 How can we match query selectors?

Programs stores HTML pages in memory using a **document object model** or DOM. Each element in the page, such as a heading and or paragraph, is a **nodes**; the **children** of a node are the elements it contains (Figure 7.1).

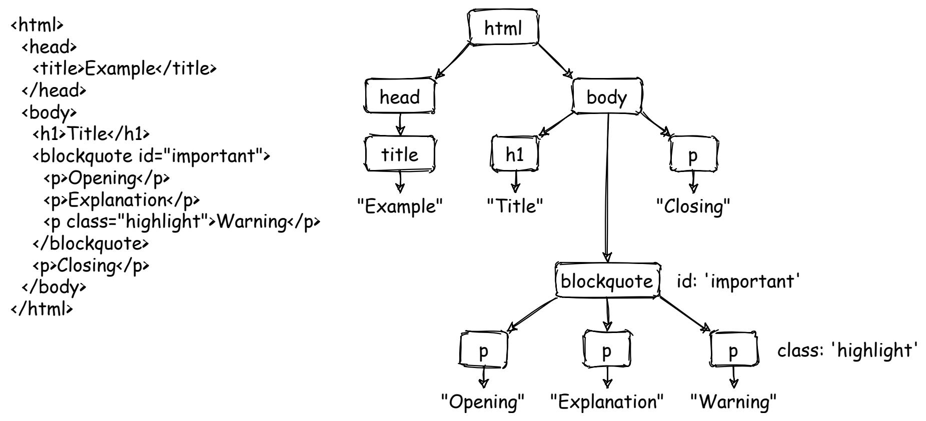
The first step is to define the patterns we want to support (Table 7.1).

According to this grammar, `blockquote#important p.highlight` is a highlighted paragraph inside the blockquote whose ID is `"important"`. To find elements in a page that match it, our `select` function breaks the query into pieces and uses `firstMatch` to search recursively down the document tree until all the selectors in the query string have matched or no matches have been found (Figure 7.2).

Meaning	Selector
Element with tag `"elt"`	`elt`
Element with `class="cls"`	`.cls`
Element with `id="ident"`	`#ident`
`child` element inside a `parent` element	`parent child`

Table 7.1: Supported patterns.

[1] https://en.wikipedia.org/wiki/Brian_Kernighan



```
<html>
  <head>
    <title>Example</title>
  </head>
  <body>
    <h1>Title</h1>
    <blockquote id="important">
      <p>Opening</p>
      <p>Explanation</p>
      <p class="highlight">Warning</p>
    </blockquote>
    <p>Closing</p>
  </body>
</html>
```

Figure 7.1: Representing an HTML document as a tree.

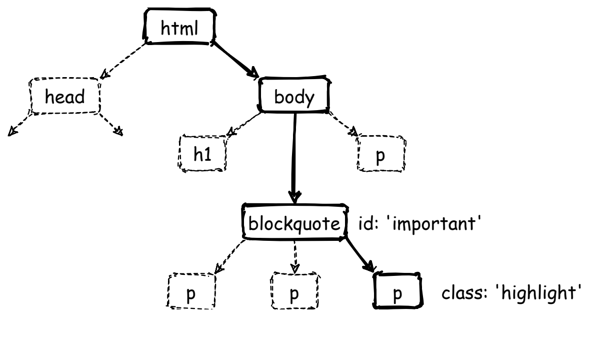


Figure 7.2: Matching a simple set of query selectors.

```
import assert from 'assert'

const select = (root, selector) => {
  const selectors = selector.split(' ').filter(s => s.length > 0)
  return firstMatch(root, selectors)
}

const firstMatch = (node, selectors) => {
  assert(selectors.length > 0,
    'Require selector(s)')

  // Not a tag.
  if (node.type !== 'tag') {
    return null
  }

  // This node matches.
  if (matchHere(node, selectors[0])) {
    // This is the last selector, so matching worked.
    if (selectors.length === 1) {
      return node
    }

    // Try to match remaining selectors.
    return firstChildMatch(node, selectors.slice(1))
  }

  // This node doesn't match, so try further down.
  return firstChildMatch(node, selectors)
}


export default select
```

Listing 7.1: simple-selectors.js

The `firstMatch` function handles three cases:

1. This node isn't an element, i.e., it is plain text or a comment. This can't match a selector, and these nodes don't have children, so the function returns `null` to indicate that matching has failed.

2. This node matches the current selector. If there aren't any selectors left then the whole pattern must have matched, so the function returns this node as the match. If there *are* more selectors, we try to match those that remain against this node's children and return whatever result that produces.

3. This node *doesn't* match the current selector, so we search the children one by one to see if there is a match further down.

This algorithm is called **depth-first search**: it explores one possible match to the end before considering any others. `firstMatch` relies on a helper function called `firstChildMatch`, which finds the first child of a node to match a set of selectors:

```
const firstChildMatch = (node, selectors) => {
  assert(node.type === 'tag',
    `Should only try to match first child of tags, not ${node.type}`)

  // First working match.
  for (const child of node.children) {
    const match = firstMatch(child, selectors)
    if (match) {
      return match
    }
  }

  // Nothing worked.
  return null
}
```

Listing 7.2: simple-selectors.js

and on the function `matchHere` which compares a node against a selector:

```
const matchHere = (node, selector) => {
  let name = null
  let id = null
  let cls = null
  if (selector.includes('#')) {
    [name, id] = selector.split('#')
  } else if (selector.includes('.')) {
    [name, cls] = selector.split('.')
  } else {
    name = selector
  }
  return (node.name === name) &&
    ((id === null) || (node.attribs.id === id)) &&
    ((cls === null) || (node.attribs.class === cls))
}
```

Listing 7.3: simple-selectors.js

This version of `matchHere` is simple but inefficient, since it breaks the selector into parts each time it is called rather than doing that once and re-using the results. We will build a more efficient version in the exercises, but let's try out the one we have. Our test cases are all in one piece of HTML:

```
const HTML = `<main>
  <p>text of first p</p>
  <p id="id-01">text of p#id-01</p>
  <p id="id-02">text of p#id-02</p>
  <p class="class-03">text of p.class-03</p>
  <div>
    <p>text of div / p</p>
    <p id="id-04">text of div / p#id-04</p>
    <p class="class-05">text of div / p.class-05</p>
    <p class="class-06">should not be found</p>
  </div>
  <div id="id-07">
    <p>text of div#id-07 / p</p>
    <p class="class-06">text of div#id-07 / p.class-06</p>
  </div>
</main>`
```

Listing 7.4: simple-selectors-test.js

The program contains a table of queries and the expected matches. The function `main` loops over it to report whether each test passes or fails:

```
const main = () => {
  const doc = htmlparser2.parseDOM(HTML)[0]
  const tests = [
    ['p', 'text of first p'],
    ['p#id-01', 'text of p#id-01'],
    ['p#id-02', 'text of p#id-02'],
    ['p.class-03', 'text of p.class-03'],
    ['div p', 'text of div / p'],
    ['div p#id-04', 'text of div / p#id-04'],
    ['div p.class-05', 'text of div / p.class-05'],
    ['div#id-07 p', 'text of div#id-07 / p'],
    ['div#id-07 p.class-06', 'text of div#id-07 / p.class-06']
  ]
  tests.forEach(([selector, expected]) => {
    const node = select(doc, selector)
    const actual = getText(node)
    const result = (actual === expected) ? 'pass' : 'fail'
    console.log(`"${selector}": ${result}`)
  })
}

main()
```

Listing 7.5: simple-selectors-test.js

`main` uses a helper function called `getText` to extract text from a node or return an error message if something has gone wrong:

```
const getText = (node) => {
  if (!node) {
    return 'MISSING NODE'
  }
  if (!('children' in node)) {
    return 'MISSING CHILDREN'
  }
  if (node.children.length !== 1) {
    return 'WRONG NUMBER OF CHILDREN'
  }
  if (node.children[0].type !== 'text') {
    return 'NOT TEXT'
  }
  return node.children[0].data
}
```

Listing 7.6: simple-selectors-test.js

When we run our program it produces this result:

```
"p": pass
"p#id-01": pass
"p#id-02": pass
"p.class-03": pass
"div p": pass
"div p#id-04": pass
"div p.class-05": pass
"div#id-07 p": pass
"div#id-07 p.class-06": pass
```

Listing 7.7: simple-selectors-test.out

We will rewrite these tests using Mocha[2] in the exercises.

Test then build

We actually wrote our test cases *before* implementing the code to match query selectors in order to give ourselves a goal to work toward. Doing this is called **test-driven development**, or TDD; while research doesn't support the claim that it makes programmers more productive [Fucci2016, Fucci2017], we find it helps prevent **scope creep** when writing lessons.

7.2 How can we implement a simple regular expression matcher?

Matching regular expressions against text relies on the same recursive strategy as matching query selectors against nodes in an HTML page. If the first element of the pattern matches where we are, we see if the rest of the pattern matches what's left; otherwise, we see if the the pattern will match further along. Our matcher will initially handle just the five cases shown in Table 7.2.
These five cases are a small subset of what JavaScript provides, but as Kernighan wrote, "This is quite a useful class; in my own experience of using regular expressions on a day-to-day basis, it easily accounts for 95 percent of all instances."

[2]https://mochajs.org/

Meaning	Character
Any literal character *c*	*c*
Any single character	.
Beginning of input	^
End of input	$
Zero or more of the previous character	*

Table 7.2: Pattern matching cases.

The top-level function that users call handles the special case of `^` at the start of a pattern matching the start of the target string being searched. It then tries the pattern against each successive substring of the target string until it finds a match or runs out of characters:

```
const match = (pattern, text) => {
  // '^' at start of pattern matches start of text.
  if (pattern[0] === '^') {
    return matchHere(pattern, 1, text, 0)
  }

  // Try all possible starting points for pattern.
  let iText = 0
  do {
    if (matchHere(pattern, 0, text, iText)) {
      return true
    }
    iText += 1
  } while (iText < text.length)

  // Nothing worked.
  return false
}
```

Listing 7.8: simple-regex.js

`matchHere` does the matching and recursing:

```
const matchHere = (pattern, iPattern, text, iText) => {
  // There is no more pattern to match.
  if (iPattern === pattern.length) {
    return true
  }

  // '$' at end of pattern matches end of text.
  if ((iPattern === (pattern.length - 1)) &&
      (pattern[iPattern] === '$') &&
      (iText === text.length)) {
    return true
  }

  // '*' following current character means match many.
  if (((pattern.length - iPattern) > 1) &&
      (pattern[iPattern + 1] === '*')) {
    while ((iText < text.length) && (text[iText] === pattern[iPattern])) {
      iText += 1
    }
    return matchHere(pattern, iPattern + 2, text, iText)
  }

  // Match a single character.
  if ((pattern[iPattern] === '.') ||
      (pattern[iPattern] === text[iText])) {
    return matchHere(pattern, iPattern + 1, text, iText + 1)
  }

  // Nothing worked.
  return false
}
```

Listing 7.9: simple-regex.js

Once again, we use a table of test cases and expected results to test it:

```
const main = () => {
  const tests = [
    ['a', 'a', true],
    ['b', 'a', false],
    ['a', 'ab', true],
    ['b', 'ab', true],
    ['ab', 'ba', false],
    ['^a', 'ab', true],
    ['^b', 'ab', false],
    ['a$', 'ab', false],
    ['a$', 'ba', true],
    ['a*', '', true],
    ['a*', 'baac', true],
    ['ab*c', 'ac', true],
    ['ab*c', 'abc', true],
    ['ab*c', 'abbbc', true],
    ['ab*c', 'abxc', false]
  ]
  tests.forEach(([regexp, text, expected]) => {
    const actual = match(regexp, text)
    const result = (actual === expected) ? 'pass' : 'fail'
    console.log(`"${regexp}" X "${text}": ${result}`)
  })
}

main()
```

Listing 7.10: simple-regex.js

```
"a" X "a": pass
"b" X "a": pass
"a" X "ab": pass
"b" X "ab": pass
"ab" X "ba": pass
"^a" X "ab": pass
"^b" X "ab": pass
"a$" X "ab": pass
"a$" X "ba": pass
"a*" X "": pass
"a*" X "baac": pass
"ab*c" X "ac": pass
"ab*c" X "abc": pass
"ab*c" X "abbbc": pass
"ab*c" X "abxc": pass
```

Listing 7.11: simple-regex.out

This program seems to work, but it actually contains an error that we will correct in the exercises. (Think about what happens if we match the pattern `/a*ab/` against the string `'aab'`.) Our design is also hard to extend: handling parentheses in patterns like `/a(bc)*d/` will require major changes. We need to explore a different approach.

7.3 How can we implement an extensible matcher?

Instead of packing all of our code into one long function, we can implement each kind of match as separate function. Doing this makes it much easier to add more matchers: we just define a function that we can mix in with calls to the ones we already have.

Rather than having these functions do the matching immediately, though, we will have each one return an object that knows how to match itself against some text. Doing this allows us to build a complex match once and re-use it many times. This is a common pattern in text processing: we may want to apply a regular expression to each line in a large set of files, so recycling the matchers will make our programs more efficient.

Each matching object has a method that takes the target string and the index to start matching at as inputs. Its output is the index to continue matching at or `undefined` indicating that matching failed. We can then combine these objects to match complex patterns (Figure 7.3).

The first step in implementing this is is to write test cases, which forces us to define the syntax we are going to support:

```
import Alt from './regex-alt.js'
import Any from './regex-any.js'
import End from './regex-end.js'
import Lit from './regex-lit.js'
import Seq from './regex-seq.js'
import Start from './regex-start.js'
```

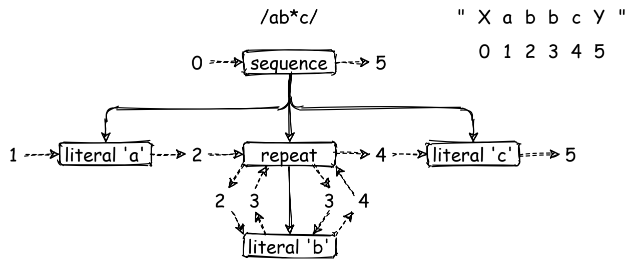


Figure 7.3: Using nested objects to match regular expressions.

```
const main = () => {
  const tests = [
    ['a', 'a', true, Lit('a')],
    ['b', 'a', false, Lit('b')],
    ['a', 'ab', true, Lit('a')],
    ['b', 'ab', true, Lit('b')],
    ['ab', 'ab', true, Seq(Lit('a'), Lit('b'))],
    ['ba', 'ab', false, Seq(Lit('b'), Lit('a'))],
    ['ab', 'ba', false, Lit('ab')],
    ['^a', 'ab', true, Seq(Start(), Lit('a'))],
    ['^b', 'ab', false, Seq(Start(), Lit('b'))],
    ['a$', 'ab', false, Seq(Lit('a'), End())],
    ['a$', 'ba', true, Seq(Lit('a'), End())],
    ['a*', '', true, Any('a')],
    ['a*', 'baac', true, Any('a')],
    ['ab*c', 'ac', true, Seq(Lit('a'), Any('b'), Lit('c'))],
    ['ab*c', 'abc', true, Seq(Lit('a'), Any('b'), Lit('c'))],
    ['ab*c', 'abbbc', true, Seq(Lit('a'), Any('b'), Lit('c'))],
    ['ab*c', 'abxc', false, Seq(Lit('a'), Any('b'), Lit('c'))],
    ['ab|cd', 'xaby', true, Alt(Lit('ab'), Lit('cd'))],
    ['ab|cd', 'acdc', true, Alt(Lit('ab'), Lit('cd'))],
    ['a(b|c)d', 'xabdy', true,
      Seq(Lit('a'), Alt(Lit('b'), Lit('c')), Lit('d'))],
    ['a(b|c)d', 'xabady', false,
      Seq(Lit('a'), Alt(Lit('b'), Lit('c')), Lit('d'))]
  ]
  tests.forEach(([pattern, text, expected, matcher]) => {
    const actual = matcher.match(text)
    const result = (actual === expected) ? 'pass' : 'fail'
    console.log(`"${pattern}" X "${text}": ${result}`)
  })
}

main()
```

Listing 7.12: regex-initial/regex-complete.js

Next, we define a **base class** that all matchers will inherit from. This class contains the `match` method that users will call so that we can start matching right away no matter what kind of matcher we have at the top level of our pattern.

```
class RegexBase {
  match (text) {
    for (let i = 0; i < text.length; i += 1) {
      if (this._match(text, i)) {
        return true
      }
    }
    return false
  }

  _match (text, start) {
    throw new Error('derived classes must override "_match"')
  }
}

export default RegexBase
```

Listing 7.13: regex-initial/regex-base.js

The base class also defines a `_match` method (with a leading underscore) that other classes will fill in with actual matching code. The base implementation of this method throws an exception so that if we forget to provide `_match` in a **derived class** our code will fail with a meaningful reminder.

One interface to call them all

Our design makes use of **polymorphism**, which literally means "having multiple forms". If a set of objects all have methods that can be called the same way, then those objects can be used interchangeably; putting it another way, a program can use them without knowing exactly what they are. Polymorphism reduces the **coupling** between different parts of our program, which in turn makes it easier for those programs to evolve.

We can now define empty versions of each matching class that all say "no match here" like this one for literal characters:

```
import RegexBase from './regex-base.js'

class RegexLit extends RegexBase {
  constructor (chars) {
    super()
    this.chars = chars
  }

  _match (text, start) {
    return undefined // FIXME
  }
}

export default (chars) => new RegexLit(chars)
```

Listing 7.14: regex-initial/regex-lit.js

Our tests now run, but most of them fail: "most" because we expect some tests not to match, so the test runner reports `true`.

```
"a" X "a": fail
"b" X "a": pass
"a" X "ab": fail
"b" X "ab": fail
"ab" X "ab": fail
"ba" X "ab": pass
"ab" X "ba": pass
"^a" X "ab": fail
"^b" X "ab": pass
"a$" X "ab": pass
"a$" X "ba": fail
"a*" X "": fail
"a*" X "baac": fail
"ab*c" X "ac": fail
"ab*c" X "abc": fail
"ab*c" X "abbbc": fail
"ab*c" X "abxc": pass
"ab|cd" X "xaby": fail
"ab|cd" X "acdc": fail
"a(b|c)d" X "xabdy": fail
"a(b|c)d" X "xabady": pass
```

Listing 7.15: regex-initial.out

This output tells us how much work we have left to do: when all of these tests pass, we're finished.

Let's implement a literal character string matcher first:

```
import RegexBase from './regex-base.js'

class RegexLit extends RegexBase {
  constructor (chars) {
    super()
    this.chars = chars
  }

  _match (text, start) {
    const nextIndex = start + this.chars.length
    if (nextIndex > text.length) {
      return undefined
    }
    if (text.slice(start, nextIndex) !== this.chars) {
      return undefined
    }
    return nextIndex
  }
}

export default (chars) => new RegexLit(chars)
```

Listing 7.16: regex-beginning/regex-lit.js

Some tests now pass, others still fail as expected:

```
"a" X "a": pass
"b" X "a": pass
"a" X "ab": pass
"b" X "ab": pass
"ab" X "ab": fail
"ba" X "ab": pass
"ab" X "ba": pass
"^a" X "ab": fail
"^b" X "ab": pass
"a$" X "ab": pass
"a$" X "ba": fail
"a*" X "": fail
"a*" X "baac": fail
"ab*c" X "ac": fail
"ab*c" X "abc": fail
"ab*c" X "abbbc": fail
"ab*c" X "abxc": pass
"ab|cd" X "xaby": fail
"ab|cd" X "acdc": fail
"a(b|c)d" X "xabdy": fail
"a(b|c)d" X "xabady": pass
```

Listing 7.17: regex-beginning.out

We will tackle `RegexSeq` next so that we can combine other matchers. This is why we have tests for `Seq(Lit('a'), Lit('b'))` and `Lit('ab')`: all children have to match in order without gaps.

But wait: suppose we have the pattern `/a*ab/`. This ought to match the text `"ab"`, but will it? The `/*/` is **greedy**: it matches as much as it can (which is also called **eager matching**). As a result, `/a*/` will match the leading `"a"`, leaving nothing for the literal `/a/` to match (Figure 7.4). Our current implementation doesn't give us a way to try other possible matches when this happens.

Let's re-think our design and have each matcher take its own arguments and a `rest` parameter containing the rest of the matchers (Figure 7.5). (We will provide a default of `null` in the creation function so we don't have to type `null` over and over again.) Each matcher will try each of its possibilities and then see if the rest will also match.

This design means we can get rid of `RegexSeq`, but it does make our tests a little harder to read:

```
import Alt from './regex-alt.js'
import Any from './regex-any.js'
import End from './regex-end.js'
import Lit from './regex-lit.js'
import Start from './regex-start.js'
```

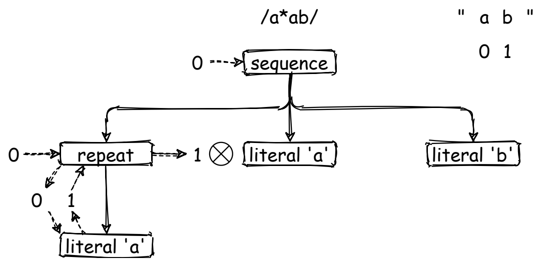


Figure 7.4: Why overly-greedy matching doesn't work.

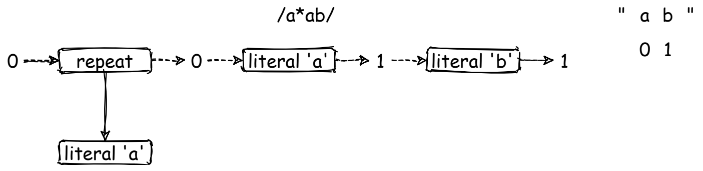


Figure 7.5: Using "rest" to match the remainder of a pattern.

```
const main = () => {
  const tests = [
    ['a', 'a', true, Lit('a')],
    ['b', 'a', false, Lit('b')],
    ['a', 'ab', true, Lit('a')],
    ['b', 'ab', true, Lit('b')],
    ['ab', 'ab', true, Lit('a', Lit('b'))],
    ['ba', 'ab', false, Lit('b', Lit('a'))],
    ['ab', 'ba', false, Lit('ab')],
    ['^a', 'ab', true, Start(Lit('a'))],
    ['^b', 'ab', false, Start(Lit('b'))],
    ['a$', 'ab', false, Lit('a', End())],
    ['a$', 'ba', true, Lit('a', End())],
    ['a*', '', true, Any(Lit('a'))],
    ['a*', 'baac', true, Any(Lit('a'))],
    ['ab*c', 'ac', true, Lit('a', Any(Lit('b'), Lit('c')))],
    ['ab*c', 'abc', true, Lit('a', Any(Lit('b'), Lit('c')))],
    ['ab*c', 'abbbc', true, Lit('a', Any(Lit('b'), Lit('c')))],
    ['ab*c', 'abxc', false, Lit('a', Any(Lit('b'), Lit('c')))],
    ['ab|cd', 'xaby', true, Alt(Lit('ab'), Lit('cd'))],
    ['ab|cd', 'acdc', true, Alt(Lit('ab'), Lit('cd'))],
    ['a(b|c)d', 'xabdy', true, Lit('a', Alt(Lit('b'), Lit('c'), Lit('d')))],
    ['a(b|c)d', 'xabady', false, Lit('a', Alt(Lit('b'), Lit('c'), Lit('d')))]
  ]
  tests.forEach(([pattern, text, expected, matcher]) => {
    const actual = matcher.match(text)
    const result = (actual === expected) ? 'pass' : 'fail'
    console.log(`"${pattern}" X "${text}": ${result}`)
  })
}

main()
```

Listing 7.18: regex-recursive/regex-complete.js

Here's how this works for matching a literal expression:

```
import RegexBase from './regex-base.js'

class RegexLit extends RegexBase {
  constructor (chars, rest) {
    super(rest)
    this.chars = chars
  }

  _match (text, start) {
    const nextIndex = start + this.chars.length
    if (nextIndex > text.length) {
      return undefined
    }
    if (text.slice(start, nextIndex) !== this.chars) {
      return undefined
    }
    if (this.rest === null) {
      return nextIndex
    }
    return this.rest._match(text, nextIndex)
  }
}

export default (chars, rest = null) => new RegexLit(chars, rest)
```

Listing 7.19: regex-recursive/regex-lit.js

The `_match` method checks whether all of the pattern matches the target text starting at the current location. If so, it checks whether the rest of the overall pattern matches what's left. Matching the start `/^/` and end `/$/` anchors is just as straightforward:

```
import RegexBase from './regex-base.js'

class RegexStart extends RegexBase {
  _match (text, start) {
    if (start !== 0) {
      return undefined
    }
    if (this.rest === null) {
      return 0
    }
    return this.rest._match(text, start)
  }
}

export default (rest = null) => new RegexStart(rest)
```

Listing 7.20: regex-recursive/regex-start.js

```
import RegexBase from './regex-base.js'

class RegexEnd extends RegexBase {
  _match (text, start) {
    if (start !== text.length) {
      return undefined
    }
    if (this.rest === null) {
      return text.length
    }
    return this.rest._match(text, start)
  }
}

export default (rest = null) => new RegexEnd(rest)
```

Listing 7.21: regex-recursive/regex-end.js

Matching either/or is done by trying the first pattern and the rest, and if that fails, trying the second pattern and the rest:

```
import RegexBase from './regex-base.js'

class RegexAlt extends RegexBase {
  constructor (left, right, rest) {
    super(rest)
    this.left = left
    this.right = right
  }

  _match (text, start) {
    for (const pat of [this.left, this.right]) {
      const afterPat = pat._match(text, start)
      if (afterPat !== undefined) {
        if (this.rest === null) {
          return afterPat
        }
        const afterRest = this.rest._match(text, afterPat)
        if (afterRest !== undefined) {
          return afterRest
        }
      }
    }
    return undefined
  }
}

const create = (left, right, rest = null) => {
  return new RegexAlt(left, right, rest)
}

export default create
```

Listing 7.22: regex-recursive/regex-alt.js

To match a repetition, we figure out the maximum number of matches that might be left, then count down until something succeeds. (We start with the maximum because matching is supposed to be greedy.) Each non-empty repetition matches at least one character, so the number of remaining characters is the maximum number of matches worth trying.

```
import RegexBase from './regex-base.js'

class RegexAny extends RegexBase {
  constructor (child, rest) {
    super(rest)
    this.child = child
  }

  _match (text, start) {
    const maxPossible = text.length - start
    for (let num = maxPossible; num >= 0; num -= 1) {
      const afterMany = this._matchMany(text, start, num)
      if (afterMany !== undefined) {
        return afterMany
      }
    }
    return undefined
  }

  _matchMany (text, start, num) {
    for (let i = 0; i < num; i += 1) {
      start = this.child._match(text, start)
      if (start === undefined) {
        return undefined
      }
    }
    if (this.rest !== null) {
      return this.rest._match(text, start)
    }
    return start
  }
}

const create = (child, rest = null) => {
  return new RegexAny(child, rest)
}
```

```
export default create
```

Listing 7.23: regex-recursive/regex-any.js

With these classes in place, our tests all pass:

```
"a" X "a": pass
"b" X "a": pass
"a" X "ab": pass
"b" X "ab": pass
"ab" X "ab": pass
"ba" X "ab": pass
"ab" X "ba": pass
"^a" X "ab": pass
"^b" X "ab": pass
"a$" X "ab": pass
"a$" X "ba": pass
"a*" X "": pass
"a*" X "baac": pass
"ab*c" X "ac": pass
"ab*c" X "abc": pass
"ab*c" X "abbbc": pass
"ab*c" X "abxc": pass
"ab|cd" X "xaby": pass
"ab|cd" X "acdc": pass
"a(b|c)d" X "xabdy": pass
"a(b|c)d" X "xabady": pass
```

Listing 7.24: regex-recursive.out

The most important thing about this design is how extensible it is: if we want to add other kinds of matching, all we have to do is add more classes. That extensibility comes from the lack of centralized decision-making, which in turn comes from our use of polymorphism and the **Chain of Responsibility** design pattern. Each component does its part and asks something else to handle the remaining work; so long as each component takes the same inputs, we can put them together however we want.

The Open-Closed Principle

The **Open-Closed Principle** states that software should be open for extension but closed for modification, i.e., that it should be possible to extend functionality without having to rewrite existing code. As we said in Chapter 3, this allows old code to use new code, but only if our design permits the kinds of extensions people are going to want to make. Since we can't anticipate everything, it is normal to have to revise a design the first two or three times we try to extend it. As [Brand1995] said of buildings, the things we make learn how to do things better as we use them.

7.4 Exercises

Split once

Modify the query selector code so that selectors like `div#id` and `div.class` are only split into pieces once rather than being re-split each time `matchHere` is called.

Find and fix the error

The first regular expression matcher contains an error: the pattern `'a*ab'` should match the string `'aab'` but doesn't. Figure out why it fails and fix it.

Unit tests

Rewrite the tests for selectors and regular expressions to use Mocha.

Find all with query selectors

Modify the query selector so that it returns *all* matches, not just the first one.

Select based on attributes

Modify the query selector to handle `[attribute="value"]` selectors, so that (for example) `div[align=center]` returns all `div` elements whose `align` attribute has the value `"center"`.

Child selectors

The expression `parent > child` selects all nodes of type `child` that are immediate children of nodes of type `parent`—for example, `div > p` selects all paragraphs that are immediate children of `div` elements. Modify `simple-selectors.js` to handle this kind of matching.

Find all with regular expressions

Modify the regular expression matcher to return *all* matches rather than just the first one.

Find one or more with regular expressions

Extend the regular expression matcher to support `+`, meaning "one or more".

Match sets of characters

Add a new regular expression matching class that matches any character from a set, so that `Charset('aeiou')` matches any lower-case vowel.

Make repetition more efficient

Rewrite `RegexAny` so that it does not repeatedly re-match text.

Lazy matching

The regular expressions we have seen so far are **eager**: they match as much as they can, as early as they can. An alternative is **lazy matching**, in which expressions match as little as they need to. For example, given the string `"ab"`, an eager match with the expression `/ab*/` will match both letters (because `/b*/` matches a 'b' if one is available) but a lazy match will only match the first letter (because `/b*/` can match no letters at all). Implement lazy matching for the `*` operator.

Optional matching

The `?` operator means "optional", so that `/a?/` matches either zero or one occurrences of the letter 'a'. Implement this operator.