

18 Regular Architecture

19 Regular Architecture is a sophisticated pattern matching tool to search for protein architectures.
20 The name is partially borrowed from Regular Expressions.

21 This package evolved from [pfql](#) to a more straightforward API and a more robust code written in
22 Typescript.

23 What RegArch is not

- 24 • RegArch is not an annotation tool.
- 25 • RegArch does not find patterns in protein sequences, it takes protein annotations as input.
- 26 • RegArch does not give partial matches

27 Specifications

- 28 • RegArch must be agnostic of annotation resources.
- 29 • RegArch must allow the use of annotations from multiple resources at the same time.
- 30 • RegArch must be able to deal with positional and non-positional pattern.

31 Building a pattern

32 `RegArch` uses JSON format to represent a pattern, in contrast with `RegExp` that uses strings.

33 `RegArch` patterns contain two sections: non-positional (`npos`) and positional (`pos`) and each of
34 them contains an array of unit patterns.

35 A unit pattern

36 Because `RegArch` is agnostic of the resource used to annotate the protein sequence, we describe
37 the name of the feature and the resource used. For example, to describe a `Secretin` domain from
38 PFAM:

```
39 {  
40   name: "Secretin",  
41   resource: "pfam"  
42 }
```

43 We can also include the number of times this domain should appear using `RegExp` notation with
44 curly brackets `{}`. For example, to describe a pattern to match proteins with 2 or more `Secretin`
45 domains:

```
46 {  
47   count: "{2,}",  
48   name: "Secretin",
```

```
49     resource: "pfam"
50 }
```

51 Combining unit patterns

52 Unit patterns must be part of either positional or non-positional sections of the `RegArch` pattern.
53 These sections are part of a JSON object with the key `patterns`. The following is a valid
54 `RegArch` pattern:

```
55 {
56   patterns: [
57     {
58       npos: [
59         {
60           name: "CheW",
61           resource: "pfam"
62         }
63       ],
64       pos: []
65     }
66   ]
67 }
```

68 This pattern matches all proteins that have 1 and only 1 CheW domain as defined in the PFAM
69 domain database (resource), anywhere in the sequence. The `pos` keyword is mandatory even if
70 no positional requirements are needed to describe this pattern.

71 Notice that the name of the feature and the resource are both conventions, but they must match
72 how the proteins have been annotated (see `RegArch` annotation standards below).

73 It is also simple to write a positional pattern:

```
74 {
75   patterns: [
76     {
77       npos: [],
78       pos: [
79         {
80           name: "CheW",
81           resource: "pfam"
82         }
83       ]
84     }
85   ]
86 }
```

87 This pattern matches the same proteins as the previous pattern: 1 and only 1 CheW anywhere in
88 the sequence. That is because `RegArch` starts to require positional matches from the position it
89 finds the first match.

90 To match proteins that start with a particular feature, we can rely on `RegArch` special features.
91 The symbols `^` and `$` are both `RegArch` features meaning *start* of the sequence and *end* of the
92 sequence, respectively. To only select proteins with 1 and only 1 `CheW` and no other feature
93 what so ever, we must make a positional pattern:

```
94 {  
95   patterns: [  
96     {  
97       npos: [],  
98       pos: [  
99         {  
100           name: "^",  
101           resource: "regarch"  
102         },  
103         {  
104           name: "CheW",  
105           resource: "pfam"  
106         },  
107         {  
108           name: "$",  
109           resource: "regarch"  
110         },  
111       ]  
112     }  
113   ]  
114 }
```

115 Notice that the pattern starts with the `^` from the resource `regarch` and ends with `$` from the
116 resource `regarch`.

117 **Complex combinations of patterns**

118 We also can combine positional and non-positional rules in the same pattern:

```
119 {  
120   patterns: [  
121     {  
122       npos: [  
123         {  
124           name: "HATPase_c",  
125           resource: "pfam"  
126         },  
127       ],  
128       pos: [  
129         {  
130           name: "CheW",  
131           resource: "pfam"  
132         },  
133         {  
134           name: "$",  
135           resource: "regarch"  
136         },  
137       ]  
138     }  
139   ]  
140 }
```

```
138     }
139   ]
140 }
```

141 This pattern matches proteins that end with CheW domain and an HATPase_c anywhere in the
142 sequence. Those are most likely histidine kinases from the chemotaxis pathway.

143 We can easily combine features from multiple resources. For example to match proteins that start
144 with a transmembrane region predicted by DAS and ends with the MCPsignal domain from
145 PFAM database:

```
146 {
147   patterns: [
148     {
149       npos: [],
150       pos: [
151         {
152           name: "^",
153           resource: "regarch"
154         },
155         {
156           name: "TM",
157           resource: "das"
158         },
159         {
160           name: "MCPsignal",
161           resource: "pfam"
162         },
163         {
164           name: "$",
165           resource: "regarch"
166         },
167       ]
168     }
169   ]
170 }
```

171 Notice that `TM` was just a convention we made. DAS only annotate transmembrane regions and
172 does not have a feature name for it. We called it as `TM` when parsing the DAS results and
173 formatted the protein annotation as required by `RegArch`.

174 **Logical AND and ORs**

175 `RegArch` allows combining patterns in AND and OR logical operations. All unit patterns in non-
176 positional arrays are interpreted as AND, meaning they all must match for the protein to be a
177 match to the pattern. For example, a pattern to match every protein that has a CheW domain
178 **AND** a `Response_reg` domain.

```
179 {
180   patterns: [
181     {
182       npos: [
```

```

183         {
184             name: "CheW",
185             resource: "pfam"
186         },
187         {
188             name: "Response_reg",
189             resource: "pfam"
190         },
191     ],
192     pos: []
193 }
194 ]
195 }

```

196 To combine non-positional patterns with OR logical operator, we need to add a pattern object in
197 the array. For example, to match proteins with CheW, Response_reg, **OR** both:

```

198 {
199     patterns: [
200         {
201             npos: [
202                 {
203                     name: "CheW",
204                     resource: "pfam"
205                 }
206             ],
207             pos: []
208         },
209         {
210             npos: [
211                 {
212                     name: "Response_reg",
213                     resource: "pfam"
214                 }
215             ],
216             pos: []
217         }
218     ]
219 }

```

220 **OR** operations are similar for positional patterns; we must repeat the pattern as a new element
221 for the array `patterns` and change only the part we want.

222 We can also build positional patterns that require that multiple unit patterns match the same
223 position. For that, `RegArch` positional patterns can be nested.

224 For example, to match proteins that have 1 occurrence of any PFAM domains except `Cache_1`
225 between two transmembranes. Thus, we must request that at the position between the two
226 transmembrane regions two unit patterns must match at the same time: any pfam domain AND a
227 `{0}` count for `Cache_1` domain:

```

228 {
229     patterns: [

```

```

230     {
231         npos: [],
232         pos: [
233             {
234                 name: 'TM',
235                 resource: 'das',
236             },
237             [
238                 {
239                     name: '.*',
240                     resource: '.*',
241                 },
242                 {
243                     count: '{0}',
244                     name: 'Cache_1',
245                     resource: 'pfam',
246                 },
247             ],
248             {
249                 name: 'TM',
250                 resource: 'das',
251             },
252         ]
253     }
254 ]
255 }

```

256 This flexible standard allows for simple and complex patterns to be built and can help to
257 compose protein sequence datasets efficiently.

258 **Formatting protein annotations**

259 As with the patterns, `RegArch` also uses JSON formatted objects to represent protein annotations.
260 The idea behind is that most bioinformatics study dealing with different types of information
261 work with nested JSON objects where different keys representing different information for a
262 single protein sequence. For this reason, `RegArch` considers the protein annotation as the value of
263 the object with key `pa`.

264 The following is a valid `RegArch` protein annotation:

```

265 {
266     "pa": {
267         "pfam": [
268             ["CheW", 171, 314, "...", 0.1, 2, 137, "...", 170, 315, "...", 113.1, 3e-
269 36, 4.3e-33, 0.94],
270             ["CheW", 2, 139, "...", 0.1, 4, 136, "...", 1, 141, "[.", 83.2, 4.9e-27, 7e-
271 24, 0.93]
272         ],
273         "smart": [
274             ["SM00260", 2, 137, 4.9e-17],
275             ["SM00260", 166, 311, 1.5e-31]
276         ]
277     }

```

278 }

279 The information from each resource must appear in an array under the key with the resource's
280 name. Each feature must appear as an array with the following format
281 ["name_of_the_feature", start, stop]. RegArch ignores other information in the rest of
282 the array.

283 **These names (feature and resources) are the names that must match with the ones used in**
284 **the patterns.**

285 **Scope of resources**

286 Positional patterns must make sure that the next unit pattern matches the next protein annotation.
287 However, because RegArch accepts annotation from multiple resources, it might be the case that
288 some of the annotations are irrelevant for this particular pattern. For example, if we annotate a
289 CheW protein with PFAM and SMART domains, we get something that looks like this:

```
290 {  
291   "pa": {  
292     "pfam": [{"CheW", 3, 136, "..", 4, 2, 135, "..", 2, 139, "..", 115.7, 1.3e-  
293 37, 6.6e-34, 0.98}],  
294     "smart": [{"SM00260", 1, 135, 2.9e-32]}  
295   }  
296 },  
297 }
```

298 If RegArch ignored the scope of resources, the following pattern would not match the above
299 annotation:

```
300 {  
301   patterns: [  
302     {  
303       npos: [],  
304       pos: [  
305         {  
306           name: "^",  
307           resource: "regarch"  
308         },  
309         {  
310           name: "CheW",  
311           resource: "pfam"  
312         },  
313         {  
314           name: "$",  
315           resource: "regarch"  
316         },  
317       ]  
318     }  
319   ]  
320 }
```

321 However, since there is only mention of the `pfam` resource, `RegArch` ignores the annotations of
322 all other resources.

323 We can make a more stringent search for CheW proteins from both resources by running two
324 `RegArch`, one for each resource, and requesting that both must be a match.

```
325 import { RegArch } from 'regarch'  
326  
327 const raPfam = new RegArch(patternUsingPfam)  
328 const raSmart = new RegArch(patternUsingSmart)  
329  
330 const matchPfam = raPfam.exec(listOfAnnotatedProteins)  
331 const matchSmart = raSmart.exec(listOfAnnotatedProteins)  
332  
333 const matchBoth = listOfAnnotatedProteins.filter((pa, i) => {  
334     return matchPfam[i] && matchSmart[i]  
335 })
```

336 **The API**

337 Much like `RegExp`, we instantiate a new instance of `RegArch` with a pattern. The only exposed
338 method in a `RegArch` instance is `exec()`, which takes an array with protein annotations as input.

339 Pretty much like this:

```
340 import { RegArch } from 'regarch'  
341  
342 const ra = new RegArch(pattern)  
343 const results = ra.exec(listOfAnnotatedProteins)
```

344 The `results` is a boolean array with `true` if the protein matches the pattern and `false` if it does
345 not.

346

347 Now we have everything we need to use `RegArch`.

348 Happy matching!

349