Idee: (Pseudocode)

`x::t` bedeutet `x` hat Typ `t`.
`x :: a -> b` bedeutet `x` ist Funktion mit Eingabetyp `a` und Ausgabetyp `b`.
Wir definieren weiter den Typ `Parser = {String} -> {String}`.
Ein Parser ist also eine Funktion von einer Menge von Strings zu einer Menge von Strings. Später werden die folgenden Funktionen auf Indizes von Matches (startIndex, endIndex) laufen, jedoch ist zur Veranschaulichung dies besser.

Definition: `readChar`
`readChar :: Char -> Parser`
`readChar(c) =` $S \to \{w_1, \cdots w_n \mid w \in S \wedge w_0 = c\}$

`readChar(c)` gibt also einen Parser(Funktion) zurück, der den char `c` konsumiert.

Beispiel: `readChar`
Mit $f_{\texttt{a}} :=$ `readChar(a)` haben wir

$$f_{\texttt{a}}(\{\texttt{ab, aac, xb, b}\}) = \{\texttt{b, ac}\}$$

Definition: `concatenate`
`concatenate :: (Parser, Parser) -> Parser`
`concatenate(p1,p2) = p2` $\circ$ `p1`

Beispiel: `concatenate`
Mit $f_{\texttt{a}}$ wie oben und $f_{\texttt{b}}$ analog definieren wir $f_{\texttt{ab}} := \texttt{concatenate}(f_{\texttt{a}}, f_{\texttt{b}}) = f_{\texttt{b}} \circ f_{\texttt{a}}$.
Dann haben wir beispielsweise

$$f_{\texttt{ab}}(\{\texttt{abc, aa, bb, xbz}\}) = f_{\texttt{b}}(f_{\texttt{a}}(\{\texttt{abc, aa, bb, xbz}\})) = f_{\texttt{b}}(\{\texttt{bc, a}\}) = \{\texttt{c}\}$$

Definition: `alternate`
`alternate :: (Parser, Parser) -> Parser`
`alternate(p1,p2) =` $S \to \texttt{p1}(S) \cup \texttt{p2}(S)$

Beispiel: `alternate`
Mit $f_{\texttt{a}}, f_{\texttt{b}}$ wie oben definieren wir $f_{\texttt{a|b}} := \texttt{alternate}(f_{\texttt{a}}, f_{\texttt{b}})$. Dann haben wir beispielsweise:

$$\begin{aligned} f_{\texttt{a|b}}(\{\texttt{abc, by, xbz}\}) &= f_{\texttt{a}}(\{\texttt{abc, by, xbz}\}) \cup f_{\texttt{b}}(\{\texttt{abc, by, xbz}\}) \\ &= \{\texttt{bc}\} \cup \{\texttt{y}\} = \{\texttt{bc, y}\} \end{aligned}$$

Definition: `iterate`
`iterate :: Parser -> Parser`
`iterate(p) =` $S \to \bigcup_{i \in \mathbb{N}_0} \texttt{p}^i(S) \quad = \quad S \to S \cup \texttt{p}(S) \cup \texttt{p}(\texttt{p}(S)) \cup \cdots$

Beispiel: `iterate`
Mit $f_{\texttt{a}}$ wie oben definieren wir $f_{\texttt{a*}} := \texttt{iterate}(f_{\texttt{a}})$. Dann haben wir beispielsweise:

$$\begin{aligned} &f_{\texttt{a*}}(\{\texttt{aaaz, axa, edc}\}) \\ &= \{\texttt{aaaz, axa, edc}\} \cup f_{\texttt{a}}(\{\texttt{aaaz, axa, edc}\}) \cup f_{\texttt{a}}(f_{\texttt{a}}(\{\texttt{aaaz, axa, edc}\})) \cup \cdots \\ &= \{\texttt{aaaz, axa, edc}\} \cup \{\texttt{aaz, xa}\} \cup \{\texttt{az}\} \cup \{\texttt{z}\} \cup \varnothing \cup \varnothing \cup \cdots \\ &= \{\texttt{aaaz, axa, edc, aaz, xa, az, z}\} \end{aligned}$$

In dieser Weise lassen sich Induktiv Parser zusammenbasteln, z.B. mit dem Thompsonvisitor.

```
concatenate(
    iterate(
        alternate(
            readChar('a'),
            concatenate(
                readChar('b'),
                readChar('b')
            )
        )
    ),
    readChar('b')
)
```

Dies entspricht der Regex `(a | bb)*b`.

Um jetzt aber vernünftig matchen zu können arbeiten wir nicht mehr mit Strings sondern Indizes auf einem globalen gemeinsamen Eingabestring.

Wir haben also beispielsweise den Eingabestring `bbaaabbb` und Paare von start und endIndex als Matches der obigen Regex wie folgt:

$(0,1)$ für `|b|baaabbb`, wo also der Kleene-Stern 0 mal gematcht wird und dann das `b`.

$(0,6)$ für `|bbaaab|bb`	$(0,8)$ für `|bbaaabbb|`	$(1,2)$ für `b|b|aaabbb`
$(1,6)$ für `b|baaab|bb`	$(1,8)$ für `b|baaabbb|`	$(2,6)$ für `bb|aaab|bb` $\cdots$

Die Definitionen sehen dann wie folgt aus:
Für ein gegebenes Eingabewort $w = w_0 w_1 \cdots w_n$ haben wir

```
type Match = (Int, Int)        type Parser = {Match} -> {Match}
```

`readChar(c)` = $S \to \{(i, j+1) \mid (i,j) \in S \wedge w_j = c\}$

Die restlichen Funktionen behalten ihre Definition.

Wenn wir also einen Parser `p` aus diesen Funktionen zusammengebaut haben, so können wir unsere Startmenge von Matches durch $S := \{(i,i) \mid i \in [0, |w|-1]\}$ definieren, wobei w das Eingabewort ist. Dies symbolisiert alle möglichen Startposition von Matches im gegebenen String. So können wir alle möglichen Matches parallel behandeln. Weiter haben wir also eine Funktion `runParser :: (Parser, String) -> {Match}` welche die Startmenge S erzeugt und diese dem Parser übergibt.

Beispiel
Regex ist `(a|b)c*`. Eingabewort $w =$ `xabccx` , der Parser lässt sich bauen durch

```
p = concatenate(
    alternate(readChar('a'), readChar('b')),
    iterate(readChar('c'))
)
```

Wir setzen zur Veranschaulichung aber Zwischenfunktionen, also
$f_x :=$ `readChar(`x`)` für $x \in \{\mathtt{a,b,c}\}$, $f_{\mathtt{a|b}} :=$ `alternate(`$f_\mathtt{a}, f_\mathtt{b}$`)`, $f_{\mathtt{c*}} :=$ `iterate(`$f_\mathtt{c}$`)` und
$p :=$ `concatenate(`$f_{\mathtt{a|b}}$`, ` $f_{\mathtt{c*}}$`)`.

Wir nehmen also an das von überall Zugriff auf w ist, also w in Java z.B. ein `public static` der `Match`-Klasse ist. Der Aufruf `runParser(`p`, ` w`)` läuft dann wie folgt ab:

$$
\begin{aligned}
\texttt{runParser}(p,\ w) &= p(S) \\
&= f_{\texttt{c*}}(f_{\texttt{a|b}}(S)) \\
&= f_{\texttt{c*}}(f_{\texttt{a}}(S) \cup f_{\texttt{b}}(S)) \\
&= f_{\texttt{c*}}(\{(j, k+1) \mid (j,k) \in S \wedge w_k = \texttt{a}\} \cup f_{\texttt{b}}(S)) \\
&= f_{\texttt{c*}}(\{(1,2)\} \cup f_{\texttt{b}}(S)) \\
&= f_{\texttt{c*}}(\{(1,2)\} \cup \{(j, k+1) \mid (j,k) \in S \wedge w_k = \texttt{b}\}) \\
&= f_{\texttt{c*}}(\{(1,2)\} \cup \{(2,3)\}) \\
&= f_{\texttt{c*}}(\{(1,2),(2,3)\}) \\
&= \bigcup_{i \in \mathbb{N}_0} f_{\texttt{c}}^i(\{(1,2),(2,3)\}) \\
&= \{(1,2),(2,3)\} \cup \bigcup_{i \in \mathbb{N}} f_{\texttt{c}}^i(\{(1,2),(2,3)\}) \\
&= \{(1,2),(2,3)\} \cup \bigcup_{i \in \mathbb{N}} \{(j, k+i) \mid (j,k) \in \{(1,2),(2,3)\} \wedge \forall l \in [1,i] : w_{k+l} = \texttt{c}\} \\
&= \{(1,2),(2,3)\} \cup \{(2,4)\} \cup \{(2,5)\} \cup \varnothing \cup \cdots \\
&= \{(1,2),(2,3),(2,4),(2,5)\}
\end{aligned}
$$

Das Ergebnis entspricht den Matches `x|a|bccx, xa|b|ccx, xa|bc|cx, xa|bcc|x`.

Grundsätzlich haben wir also eine Regex-Matching Funktion rekursiv über den Aufbau unserer Regexe definiert. In einem würde es sich auch wie folgt schreiben lassen:

Sei Σ ein Alphabet, und $\mathcal{R}_\Sigma$ die Menge aller Regexe über Σ.
Sei ferner $r \in \mathcal{R}_\Sigma$, $w \in \Sigma^*$ und $S := \{(i,i) \mid i \in [0, |w|-1]\}$. Dann definiere:

$$
\texttt{ext}(r, S) := \begin{cases} \{(i, j+1) \mid (i,j) \in S \wedge w_j = c\} & \text{falls } r = c \in \Sigma \\ \texttt{ext}(r_2, \texttt{ext}(r_1, S)) & \text{falls } r = (r_1 r_2) \text{ für } r1, r2 \in \mathcal{R}_\Sigma \\ \texttt{ext}(r1, S) \cup \texttt{ext}(r2, S) & \text{falls } r = (r1 \mid r2) \text{ für } r1, r2 \in \mathcal{R}_\Sigma \\ \texttt{iter}(t, S) & \text{falls } r = t^* \text{ für } t \in \mathcal{R}_\Sigma \end{cases}
$$

$$
\texttt{iter}(t, S) := \begin{cases} \varnothing & \text{falls } S = \varnothing \\ S \cup \texttt{iter}(t, \texttt{ext}(t, S)) & \text{sonst} \end{cases}
$$

$$
\texttt{runParser}(r, w) := \texttt{ext}(r, S)
$$

Um Anforderungen wie z.B die Priorität in Alternationen zu erfüllen,
müsste man der Klasse `Match` Informationen hinzufügen. Man kann bspw. also ein Tripel (startIndex, endIndex, priority) anstatt dem obigen Paar benutzen.

Dies erlaubt einem bei der Rekusion an jeder Stelle einer Alternation die Priorität der Matches entsprechend anzupassen.

Wenn wir 0 als höchste, ∞ als geringste Priorität betrachten, definieren wir

$$
\texttt{decrPrio}(S) := \{(i, j, p+1) \mid (i, j, p) \in S\}
$$

Dann können wir die 3. Zeile der Definition von `ext` ersetzen durch:

$$
\texttt{ext}(r, S) = \texttt{ext}(r1, S) \cup \texttt{decrPrio(ext(r2, S))} \quad \text{falls } r = (r1 \mid r2) \text{ für } r1, r2 \in \mathcal{R}_\Sigma
$$

Damit haben wir dann neben StartPosition und Länge ein neues Kriterium, nach welchem wir die Ergebnismenge von Matches des Parsers ordnen können. Die restlichen Funktionen entsprechend anzupassen sollte kein Problem darstellen.