# 4
# Convertendo regexes em PEGs

Neste capítulo apresentamos um estudo das conversões de regexes em PEGs. Como ponto de partida, introduzimos uma função que converte expressões regulares puras em PEGs chamada *continuation-based conversion* (OIdM10). Logo em seguida, apresentamos extensões dessa função que permitem a conversão de algumas construções regexes em PEGs.

Uma PEG sintaticamente similar a uma expressão regular em geral não reconhece a linguagem definida pela expressão. Por exemplo, a expressão regular `(a|ab)c` define a linguagem {`"ac"`, `"abc"`}, mas a PEG `(a/ab)c` não reconhece a string `"abc"`. Isso ocorre porque, após o casamento do padrão `a` com a subcadeia `"a"`, a PEG tenta casar o padrão `c` com o resto da entrada, `"bc"`. Assim que esse casamento falha, não há backtracking para a segunda alternativa, no caso `ab`.

O exemplo acima é um caso particular da estrutura $(\mathtt{p}_1 \,|\, \mathtt{p}_2)\,\mathtt{p}_3$. Em PEGs, se $\mathtt{p}_1$ casar e $\mathtt{p}_3$ falhar, não haverá backtracking para a alternativa $\mathtt{p}_2$. Em implementações de regexes, este backtracking sempre ocorre. Para construir PEGs equivalentes a expressões regulares com essa estrutura, é necessário concatenar o padrão $\mathtt{p}_3$ no fim de cada alternativa, resultando uma expressão na forma $(\mathtt{p}_1\,\mathtt{p}_3 \,|\, \mathtt{p}_2\,\mathtt{p}_3)$. Ao distribuirmos a expressão $\mathtt{p}_3$ entre as alternativas, solucionamos nosso problema que é construir PEGs equivalentes as expressões regulares. Isso ocorre porque a PEG $(\mathtt{p}_1 \,/\, \mathtt{p}_2)\,\mathtt{p}_3$ não reconhece a mesma linguagem que a expressão $(\mathtt{p}_1 \,|\, \mathtt{p}_2)\,\mathtt{p}_3$ define, porém, a expressão $(\mathtt{p}_1\,\mathtt{p}_3 \,|\, \mathtt{p}_2\,\mathtt{p}_3)$ é equivalente a PEG $(\mathtt{p}_1\,\mathtt{p}_3 \,/\, \mathtt{p}_2\,\mathtt{p}_3)$

Para convertermos expressões regulares em PEGs, a função continuation-based conversion utiliza uma continuação explícita para guiar o processo de conversão. Como exemplo, voltamos a expressão `(a|ab)c` que é uma concatenação de duas subexpressões: `(a|ab)` e `c`. Podemos visualizar a segunda subexpressão, `c`, como a continuação da primeira subexpressão. Essa continuação basicamente define *o que falta casar*. Ao concatenamos essa continuação para todas as alternativas da primeira subexpressão, obtemos a PEG `ac/abc`, que reconhece corretamente a linguagem {`"ac"`, `"abc"`}.

## 4.1 Continuation-based conversion

O continuation-based conversion é uma função $\Pi(\mathtt{e}, \mathtt{k})$ que recebe uma expressão regular `e` e uma PEG `k`, e retorna uma expressão de parsing. A PEG `k` denota uma *continuação*, isto é, define o que é necessário casar após o casamento de `e`. Como efeito-colateral, a função $\Pi$ constrói uma PEG que, ao final do processo de conversão, será equivalente a expressão regular `e`.

A função $\Pi$ é definida por 4 casos, correspondentes à estrutura da expressão `e`:

$$\Pi(\varepsilon, \mathtt{k}) = \mathtt{k} \tag{4-1}$$

$$\Pi(\mathtt{c}, \mathtt{k}) = \mathtt{c}\,\mathtt{k} \tag{4-2}$$

$$\Pi(\,\mathtt{p}_1\,\mathtt{p}_2,\,\mathtt{k}\,) = \Pi(\,\mathtt{p}_1,\,\Pi(\,\mathtt{p}_2,\,\mathtt{k}\,)\,) \tag{4-3}$$

$$\Pi(\,\mathtt{p}_1 \mid \mathtt{p}_2,\,\mathtt{k}\,) = \Pi(\,\mathtt{p}_1,\,\mathtt{k}\,) \,/\, \Pi(\,\mathtt{p}_2,\,\mathtt{k}\,) \tag{4-4}$$

O caso 4-1 descreve a conversão da expressão regular $\varepsilon$ que casa com a string vazia. Essa conversão resulta em uma expressão composta apenas pela continuação.

O caso 4-2 descreve a conversão de um simples caracter `c`. Como resultado, temos a expressão $\mathtt{c}\,\mathtt{k}$, que casa `c` e, logo em seguida, a continuação `k`.

O caso 4-3 converte concatenações. Para converter uma concatenação $\mathtt{p}_1\,\mathtt{p}_2$, primeiro convertemos a segunda subexpressão usando a continuação original. Essa conversão é denotada por $\Pi(\,\mathtt{p}_2,\,\mathtt{k}\,)$. Logo em seguinda, convertemos a subexpressão $\mathtt{p}_1$ utilizando o resultado de $\Pi(\,\mathtt{p}_2,\,\mathtt{k}\,)$ como continuação.

O caso 4-4 mostra como converter alternativas. Para isso, basta convertermos cada alternativa usando a continuação original `k`. Este passo consiste, basicamente, em distribuir a continuação original entre as alternativas. É esta distribuição da continuação que soluciona o nosso problema original, que é converter expressões na forma $(\mathtt{p}_1 \mid \mathtt{p}_2)\,\mathtt{p}_3$ em $(\mathtt{p}_1\,\mathtt{p}_3 \mid \mathtt{p}_2\,\mathtt{p}_3)$.

A conversão de repetições $\mathtt{e}^*$ é um pouco mais complexa. A intuição para a conversão surge da seguinte igualdade de repetições:

$$\mathtt{e}^* \;=\; \mathtt{e}\;\mathtt{e}^* \mid \varepsilon$$

A partir dessa igualdade, podemos expandir $\Pi(\mathtt{e}^*, \mathtt{k})$ da seguinte forma:

$$\begin{aligned}
\Pi(\ \mathtt{e}^*,\ \mathtt{k}\ ) &= \Pi(\ \mathtt{e}\,\mathtt{e}^* \mid \varepsilon,\ \mathtt{k}\ ) && \text{(igualdade)}\\
&= \Pi(\ \mathtt{e}\,\mathtt{e}^*,\ \mathtt{k}\ ) \,/\, \Pi(\ \varepsilon,\ \mathtt{k}\ ) && \text{(caso 4-4)}\\
&= \Pi(\ \mathtt{e}\,\mathtt{e}^*,\ \mathtt{k}\ ) \,/\, \mathtt{k} && \text{(caso 4-1)}\\
&= \Pi(\ \mathtt{e},\ \Pi(\ \mathtt{e}^*,\ \mathtt{k}\ )\ ) \,/\, \mathtt{k} && \text{(caso 4-3)}
\end{aligned}$$

Se substituirmos $\Pi(\mathtt{e}^*,\ \mathtt{k})$ por um não-terminal `A`, obtemos a equação abaixo:

$$\mathtt{A} = \Pi(\ \mathtt{e},\ \mathtt{A}\ ) \,/\, \mathtt{k}$$

Desse modo, a conversão de repetições adiciona um novo não-terminal `A` e uma nova regra na PEG resultante. A conversão de repetições é a seguinte:

$$\begin{aligned}
\Pi(\ \mathtt{e}^*,\ \mathtt{k}\ ) &= \mathtt{A} \\
\mathtt{A} &\leftarrow \Pi(\ \mathtt{e},\ \mathtt{A}\ ) \,/\, \mathtt{k}
\end{aligned} \tag{4-5}$$

Abaixo mostramos um exemplo que converte a expressão `(ba|a)*a`. Note que a conversão começa com a continuação `""`. A continuação vazia significa que, assim que a expressão inteira casar, não há nada mais a ser casado.

$$\begin{aligned}
\Pi(\ \mathtt{(ba|a)*\,a},\ \mathtt{""}\ ) &= \Pi(\ \mathtt{(ba|a)*},\ \Pi(\ \mathtt{a},\ \mathtt{""}\ )\ ) \\
&= \Pi(\ \mathtt{(ba|a)*},\ \mathtt{a}\ ) \\
&= \mathtt{A} \\
\mathtt{A} &\leftarrow \mathtt{b\,a\,A} \,/\, \mathtt{a\,A} \,/\, \mathtt{a}
\end{aligned}$$

Note que `baA/aA` é o resultado de $\Pi(\ \mathtt{(ba|a)}, \mathtt{A}\ )$.

A prova formal da corretude do continuation-based conversion foi apresentada por Sérgio Medeiros em sua tese de doutorado (Med10). A idéia básica desta prova consiste em mostrar que se a expressão regular $\mathtt{e}_k$ é equivalente a PEG $\mathtt{p}_k$, denotado por $\mathtt{e}_k \sim \mathtt{p}_k$, então a concatenação de uma expressão `e` com $\mathtt{e}_k$ é equivalente a conversão de `e` usando $\mathtt{p}_k$ como continuação: $\mathtt{e}_k \sim \mathtt{p}_k \Rightarrow \mathtt{e}\,\mathtt{e}_k \sim \Pi(\mathtt{e}, \mathtt{p}_k)$.

Abaixo, mostramos exemplos de conversão de expressões que estão presentes no livro *Mastering Regular Expressions* (Fri06). Nosso primeiro

exemplo mostra a conversão de uma expressão regular muito comum que identifica os campos `From`, `Subject` e `Date` de cabeçalhos de e-mail:

$$\Pi(\texttt{(From|Subject|Date):}, \texttt{""}) \;=\; \texttt{"From:"} \,/\, \texttt{"Subject:"} \,/\, \texttt{"Date:"}$$

Neste exemplo, a PEG resultante possui o caracter `":"` distribuido entre as alternativas, já que este corresponde a continuação dessas alternativas. Note que a PEG possui o mesmo número de alternativas que a expressão regular original. Porém, há casos de expressões que, ao convertermos, resultam em PEGs com mais alternativas que a expressão original e, consequentemente, estas PEGs são maiores que a expressão regular. Por exemplo, considere a expressão abaixo que casa com formas diferentes de grafias do nome `Jeffrey`:

$$\Pi(\texttt{(Geo|Je)ff(re|er)y}, \texttt{""}) \;=\; \texttt{"Geoff"}\ (\texttt{"rey"} \,/\, \texttt{"ery"}) \,/\, \texttt{"Jeff"}\ (\texttt{"rey"} \,/\, \texttt{"ery"})$$

Neste exemplo, a PEG resultante possui uma alternativa a mais que a expressão regular. Na seção seguinte, analisamos o tamanho da PEG resultante em relação a expressão regular e mostramos alguns casos em que temos o aumento exponencial do número de alternativas.

## 4.2 Tamanho da PEG resultante

Ao convertermos expressões compostas por sequências de alternativas, o algoritmo Π pode aumentar exponencialmente o tamanho das PEGs resultantes. Por exemplo, ao convertermos a expressão `(a|b)(c|d)`, composta por uma sequência de duas alternativas, o algoritmo Π constrói a PEG `a(c/d)/b(c/d)`, que possui uma alternativa a mais que a expressão original. Se convertermos uma expressão composta por uma sequência de 3 alternativas, como `(a|b)(c|d)(e|f)`, o algoritmo Π constrói uma PEG que possui 7 alternativas, no caso, `a(c(e/f)/d(e/f))/b(c(e/f)/d(e/f))`.

De modo geral, a conversão de expressões na forma `(a|b)(X)`, onde `X` é uma expressão regular qualquer, resulta em PEGs na forma `a(X)/b(X)`. Ao distribuirmos `X` entre as alternativas, consequentemente construimos uma PEG que possui o dobro de alternativas de `X`, acrescido de mais uma alternativa. Logo, o número total de alternativas da PEG resultante é `2f(X) + 1`, onde `f(X)` é o número de alternativas da expressão `X`. Por fim, é fácil ver que a conversão de expressões com sequências de `n` alternativas resulta em uma PEG com $2^n-1$ alternativas.

Uma maneira de evitarmos a explosão exponencial de alternativas é utilizando a conversão abaixo, que é equivalente ao caso 4-4:

$$\begin{aligned} \Pi(\mathtt{p}_1 \,|\, \mathtt{p}_2, \mathtt{k}) &= \Pi(\mathtt{p}_1, \mathtt{A}) \,/\, \Pi(\mathtt{p}_2, \mathtt{A}) \\ \mathtt{A} &\leftarrow \mathtt{k} \end{aligned}$$

Ao convertermos expressões compostas por sequências de alternativas, usando a conversão acima, mantemos o mesmo número de alternativas que a expressão original. Assim, evitamos o crescimento exponencial da PEG resultante. Abaixo, mostramos o resultado da conversão da expressão `(a|b)(c|d)(e|f)` usando essa estratégia:

$$\begin{aligned} \Pi(\texttt{(a|b)(c|d)(e|f)}, \texttt{""}) &= \mathtt{a\,A} \,/\, \mathtt{b\,A} \\ \mathtt{A} &\leftarrow \mathtt{c\,B} \,/\, \mathtt{d\,B} \\ \mathtt{B} &\leftarrow \mathtt{e\,C} \,/\, \mathtt{f\,C} \\ \mathtt{C} &\leftarrow \texttt{""} \end{aligned}$$

Considere a expressão, que mencionamos na seção anterior, que casa com formas diferentes de grafias do nome `Jeffrey` (Fri06). Ao convertermos essa expressão usando a nova forma de converter alternativas obtemos:

$$\begin{aligned} \Pi(\texttt{ (Geo|Je)ff(re|er)y}, \texttt{""}) &= \texttt{"Geo" A / "Je" A} \\ \mathtt{A} &\leftarrow \texttt{"ff" ("re" B / "er" B)} \\ \mathtt{B} &\leftarrow \texttt{"y"} \end{aligned}$$

Contudo, com exceção de alguns exemplos incomuns, como a expressão Perl de 82 linhas para validar endereços de e-mail (War02), a maioria dos usos práticos de regexes utilizam expressões com poucas alternativas. A expressão que possui o maior número de alternativas no livro *Mastering Regular Expressions* (Fri06), por exemplo, contém 13 alternativas, no caso, `com|edu|gov|int|mil|net|org|biz|info|name|museum|coop|aero|[a-z][a-z]`. Como esta expressão não é composta por sequências de alternativas, a conversão desta resulta em uma PEG com o mesmo número de alternativas. Todos os outros exemplos desse livro possuem no máximo 4 alternativas, sendo que, nenhum deles são sequências de alternativas.

Além disso, o maior número de alternativas na PEG resultante não im-

pacta na eficiência dessa PEG em relação as expressões regexes. Basicamente, este impacto não ocorre pois a PEG resultante não executa mais backtrackings que a expressão original. Na verdade, a PEG gerada executa o mesmo número de backtrackings que a expressão original, pois a função Π apenas converte os backtrackings globais de regexes em backtrackings locais de PEGs.

A figura 4.1 mostra como o motor regex executa os backtrackings da expressão `(a|b)(c|d)`. Neste exemplo, cada estado, com exceção do estado final, possui dois caminhos que levam ao estado seguinte. Como os backtrackings são globais, multiplicamos os caminhos que cada estado possui para o estado seguinte e obtemos o máximo de 4 caminhos.

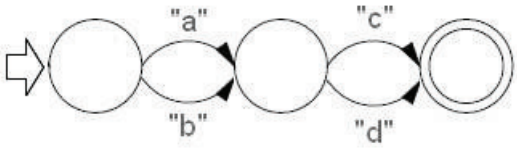


Figura 4.1: Backtracking global

A figura 4.2 exibe a PEG criada a partir da conversão da expressão `(a|b)(c|d)`. Como o algoritmo Π distribui a continuação por todas as alternativas, os mesmos caminhos que existem na expressão regular são construidos de forma explicita. Note que temos, ao todo, 4 caminhos equivalentes aos da expressão regular original.

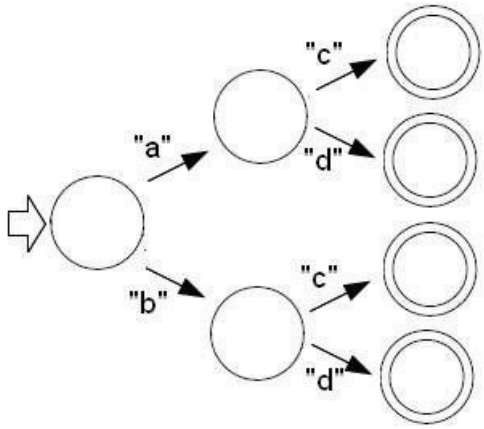


Figura 4.2: Backtracking local

## 4.3 Regexes → PEGs

Nesta seção apresentamos extensões da função Π que permitem a conversão de alguns regexes em PEGs.

A primeira extensão da função Π permite convertermos expressões independentes. Expressões independentes casam de forma independente da expressão mais externa. A idéia básica da conversão de expressões independentes é construir PEGs que não permitem backtracking para partes que já foram consumidas. Note que este é o comportamento original do backtracking em PEGs. Para garantir esse comportamento, a conversão de uma expressão independente separa a expressão da continuação original. Para isso, a conversão de uma expressão independente utiliza a continuação vazia ao invés da continuação original. Por fim, concatenamos com a continuação original, como descrito no caso abaixo:

$$\Pi(\ \texttt{(?>p)},\ \texttt{k}\ ) = \Pi(\ \texttt{p},\ \texttt{""}\ )\,\texttt{k} \tag{4-6}$$

O caso 4-5, apresentado na seção anterior, mostrou como converter repetições gulosas. Outras repetições gulosas, como `p+` e `p?`, são açúcares sintáticos para as respectivas expressões $\texttt{p}\texttt{p}^*$ e `p|""`. Dessa forma, a conversão apenas substitui a expressão quantificada pela expressão desaçucarada. Os casos abaixo descrevem essas conversões:

$$\Pi(\ \texttt{p+},\ \texttt{k}\ ) = \Pi(\ \texttt{p}\,\texttt{p}^*,\ \texttt{k}\ ) \tag{4-7}$$

$$\Pi(\ \texttt{p?},\ \texttt{k}\ ) = \Pi(\ \texttt{p}\,|\,\texttt{""}\ ,\ \texttt{k}\ ) \tag{4-8}$$

A repetição `p{n}` consiste basicamente em `n` concatenações da expressão `p`. O caso abaixo descreve como é a conversão dessa construção. Como notação, usaremos `p...p` pra representar `n` concatenações de `p`.

$$\Pi(\ \texttt{p\{n\}},\ \texttt{k}\ ) = \Pi(\ \texttt{p...p},\ \texttt{k}\ ) \tag{4-9}$$

A expressão `p{n,}` é composta por `n` concatenações da expressão `p` acrecida de uma concatenação do padrão $\texttt{p}^*$ no final. O caso abaixo apresenta a conversão de `p{n,}`:

$$\Pi(\ \texttt{p\{n,\}},\ \texttt{k}\ ) = \Pi(\ \texttt{p...p}\texttt{p}^*,\ \texttt{k}\ ) \tag{4-10}$$

O padrão `p{n,m}` é equivalente a `n` concatenações do padrão `p` seguido de `m` padrões `p?`. Como exemplo, `p{2,4}` é equivalente a `ppp?p?`. O caso 4-11 apresenta a conversão de `p{n,m}`:

$$\Pi(\ \texttt{p\{n,m\}},\ \texttt{k}\ ) = \Pi(\ \texttt{p...pp?...p?},\ \texttt{k}\ ) \tag{4-11}$$

É possível otimizarmos a conversão acima da seguinte forma. Considere a sequência `p?p?p?`. Podemos substituir esta sequência pela expressão equivalente `p(p(p|"")|"")|""` e, ao convertermos diretamente essa expressão, teremos uma PEG ligeiramente mais eficiente. Abaixo, mostramos os pas-

sos que mostram a equivalência da expressão `p?p?p?` com a expressão `p(p(p|"")|"")|""`.

$$
\begin{aligned}
\texttt{p?p?p?} &= \texttt{(p|"")(p|"")(p|"")} && \text{(igualdade)}\\
&= \texttt{(p|"")(pp|p|"")} && \text{(distributiva)}\\
&= \texttt{ppp|pp|p|""} && \text{(distributiva)}\\
&= \texttt{p(pp|p|"")|""} && \text{(evidência)}\\
&= \texttt{p(p(p|"")|"")|""} && \text{(evidência)}
\end{aligned}
$$

A conversão de uma expressão preguiçosa é semelhante à conversão de expressão gulosa, porém com a ordem trocada das alternativas.

$$
\begin{aligned}
\Pi(\ \texttt{p*?},\ \texttt{k}\ ) &= \texttt{A} \\
\texttt{A} &\leftarrow \texttt{k} \,/\, \Pi(\ \texttt{p},\ \texttt{A}\ )
\end{aligned}
\tag{4-12}
$$

Na conversão de quantificadores preguiçosos, a PEG gerada tenta casar apenas a continuação `k`; caso falhe, a PEG tenta casar o resultado da conversão da expressão `p` e executa a repetição.

Para convertermos expressões possessivas é preciso definir o significado de *possessivo*. Uma maneira de definir a semântica precisa de uma repetição possessiva é considerando-a um caso particular de expressão independente, já que ambas não permitem backtracking para partes que foram consumidas. A partir dessa definição, a conversão de quantificadores possessivos é trivial:

$$\Pi(\ \texttt{p*+},\ \texttt{k}\ ) = \Pi(\ \texttt{p*},\ \texttt{""}\ )\ \texttt{k} \tag{4-13}$$

Para convertermos lookaheads, utilizamos o predicado sintático `&` para que a PEG equivalente ao lookahead não consuma caracteres. A conversão de lookaheads utiliza a PEG vazia como continuação para separar a expressão da continuação original. Essa estratégia foi utilizada na conversão da expressões independentes pois, assim como elas, lookaheads casam de forma independente do restante da expressão. O caso abaixo apresenta a conversão de lookaheads positivos para PEGs.

$$\Pi(\ \texttt{(?=p)},\ \texttt{k}\ ) = \texttt{\&}\,\Pi(\ \texttt{p},\ \texttt{""}\ )\,\texttt{k} \tag{4-14}$$

A conversão de lookahead negativo é análoga à conversão de lookahead positivo, a diferença está na utilização do predicado de negação (`!`) ao invés do predicado de lookahead.

$$\Pi(\ \texttt{(?!\,p)},\ \texttt{k}\ ) = \texttt{!}\,\Pi(\ \texttt{p},\ \texttt{""}\ )\,\texttt{k} \tag{4-15}$$

Para converter âncora de fim de entrada `$`, utilizamos a PEG `(!.)`, que verifica se não há nenhum caracter em seguida, concatenado com a continuação original `k`. Em geral, a continuação é vazia já que esta âncora deve ser posicionada no final da expressão. Porém, os regexes permitem padrões em que a âncora está no interior da expressão, como `a$b`, que obviamente não casam com nenhuma entrada. Dessa forma, a concatenação da continuação original no final da PEG gerada mapeia estes casos para PEGs que não reconhecem nenhuma entrada. Abaixo mostramos como é a conversão dessa âncora:

$$\Pi(\ \texttt{\$},\ \texttt{k}\ ) = \texttt{!.}\ \texttt{k} \tag{4-16}$$

A conversão de âncora de fim de linha `\z` produz a PEG `(&"\n" / !. )`, onde a primeira alternativa verifica se há um fim de linha logo em seguida; se falhar, a segunda alternativa verifica se não há nenhum caracter em seguida, que sinaliza o fim da entrada. Por fim, basta concatenamos a continuação `k`:

$$\Pi(\ \texttt{\textbackslash z},\ \texttt{k}\ ) = \texttt{(\&"\textbackslash n" / !. )}\ \texttt{k} \tag{4-17}$$

Perl, PCRE e Ruby possuem mais uma âncora (`\Z`) que casa com fim de entrada porém pode ter um fim de linha opcional antes do fim da entrada. O caso 4-18 descreve a conversão da âncora `\Z`:

$$\Pi(\ \texttt{\textbackslash Z},\ \texttt{k}\ ) = \texttt{\&("\textbackslash n"? !.)}\ \texttt{k} \tag{4-18}$$

Embora seja muito fácil convertermos âncoras de fim de linha e fim de entrada para PEGs, as âncoras de início necessitam de extensões do formalismo de PEGs pois não temos como verificar os caracteres que já foram consumidos, por exemplo, `"\n"` no caso de âncoras que casam com início de linha. Na seção 4.7 abordamos com mais detalhes as âncoras de início e suas características que dificultam a conversão para PEGs.

Agrupamento simples é uma construção que possui conversão trivial para PEGs. A função Π já executa a conversão da expressão de modo que a PEG construida mantém o agrupamento correto da expressão. O caso abaixo descreve a conversão de agrupamento simples:

$$\Pi(\ \texttt{(?:p)},\ \texttt{k}\ ) = \Pi(\ \texttt{p},\ \texttt{k}\ ) \tag{4-19}$$

Por fim, apresentamos a conversão de construções que permitem comentários dentro da expressão. Como os comentários não modificam o casamento, a conversão apenas os ignoram. Veja abaixo:

$$\Pi(\ \texttt{(?\#comment)},\ \texttt{k}\ ) = \texttt{k} \tag{4-20}$$

Como forma de vermos a aplicação prática dessas conversões, vamos mostrar a conversão de algumas expressões frequentemente utilizadas no desenvolvimento de software. Abaixo, mostramos exemplos de conversão de expressões que estão presentes no livro *Regular Expressions Cookbook* (GL09). Como primeiro exemplo, considere a expressão `<p>.*?</p>` que casa com parágrafos em HTML.

$$\begin{aligned} \Pi(\texttt{<p>.*?</p>}, \texttt{""}) &= \texttt{"<p>"}\ \texttt{A} \\ \texttt{A} &\leftarrow \texttt{"</p>"}\ /\ \texttt{.}\ \texttt{A} \end{aligned}$$

Neste exemplo, note que a PEG casa o início do parágrafo `<p>` seguido do não-terminal `A`. Este não-terminal, por sua vez, tenta casar o fim do parágrafo e, caso falhe, tenta a alternativa seguinte, que consome um caracter (`.`) e tenta casar `A` novamente. O tamanho da PEG resultante é igual ao da conversão das repetições gulosas, isto é, a conversão dessas repetições acrescenta na PEG um novo não-terminal e uma alternativa.

O exemplo seguinte mostra a conversão da expressão `^(https?|ftp)://.+$`, que valida protocolos de transmissão de arquivos. Na conversão abaixo, usamos a segunda conversão de alternativas que mostramos anteriormente.

$$\begin{aligned} \Pi(\texttt{\^{}(https?|ftp)://.+\$}, \texttt{""}) &= \texttt{"http"}\ (\texttt{"s"}\ \texttt{A}\ /\ \texttt{A})\ /\ \texttt{"ftp"}\ \texttt{A} \\ \texttt{A} &\leftarrow \texttt{"://"}\ \texttt{.}\ \texttt{B} \\ \texttt{B} &\leftarrow \texttt{.}\ \texttt{B}\ /\ \texttt{!.} \end{aligned}$$

Neste exemplo, considere que a conversão da âncora $\wedge$, que casa com início da entrada, não produz nenhuma alteração na PEG resultante, já que PEGs casam em modo ancorado por natureza [1].

O próximo exemplo mostra a conversão da expressão `(?=.{1,5}$).*` que utiliza lookahead positivo para verificar se o tamanho da string está entre 1 e 5.

$$\begin{aligned} \Pi(\texttt{(?=.\{1,5\}\$).*}, \texttt{""}) &= \texttt{\&(.(.(.(.(./" ")/" ")/" ")/" ")!.)}\ \texttt{A} \\ \texttt{A} &\leftarrow \texttt{.}\ \texttt{A}\ /\ \texttt{" "} \end{aligned}$$

[1] Falaremos mais a respeito da âncora $\wedge$ na seção 4.7

Note que a conversão de lookahead positivo é trivial, já que em PEGs temos o predicado que possui exatamente o mesmo comportamento. Neste exemplo, como a expressão é quantificada numericamente, de 1 a 5 repetições, nos convertermos este quantificador como uma sequência de alternativas.

Existem diversas otimizações que podem ser feitas na PEG resultante da conversão de Π. No caso 4-5, mostramos que a conversão de `*` resulta em uma PEG que repete através de chamadas a um não-terminal. A conversão de `*` é feita dessa maneira para preservar o comportamento não-cego das repetições de regexes. Porém, dependendo da expressão, é possível convertermos `*` em repetições cegas de PEGs e, dessa forma, otimizar o casamento da PEG.

Para formalizarmos essa otimização, primeiro precisamos definir um conceito auxiliar que será útil na nossa formalização.

Dizemos que uma PEG $\mathtt{p}_1$ **não interfere** com uma PEG $\mathtt{p}_2$ quando $\nexists$ `x`, `y` tal que $\mathtt{p}_1$ casa com `x` e $\mathtt{p}_2$ casa com `xy`.

Dado este conceito, podemos formalizar a otimização das repetições da seguinte forma:

$$\text{Se } \Pi(\mathtt{p}, \varepsilon) \text{ não interfere com } \mathtt{k} \;\rightarrow\; \Pi(\mathtt{p*}, \mathtt{k}) = \Pi(\mathtt{p}, \varepsilon)\mathtt{*}\,\mathtt{k}$$

A idéia básica dessa otimização é identificar que a PEG resultante da conversão de $\Pi(\mathtt{p}, \varepsilon)$ pode não consumir o trecho da entrada que `k` consome inicialmente. Assim, temos a garantia que a repetição de $\Pi(\mathtt{p}, \varepsilon)$ não irá consumir mais do que deveria e, com isso, podemos repeti-la de forma cega.

No capítulo 5, mostramos uma implementação de uma versão simplificada desta otimização. Durante a conversão da árvore sintática da expressão em uma PEG, identificamos as repetições gulosas e obtemos um conjunto de caracteres iniciais que a continuação `k` pode casar. De posse dessa informação, verificamos se o padrão que é repetido casa com estes caracteres iniciais; se não casar, convertemos a repetição regex diretamente para uma repetição cega de PEG; se casar, significa que não podemos fazer esta otimização. Ainda no capítulo 5 mostramos como implementar esta otimização e os impactos desta no desempenho da PEG resultante.

## 4.4 Capturas

O formalismo original de expressões regulares e PEGs não definem capturas. Como PEGs não possuem operadores que obtêm valores que foram

reconhecidos, não é possível a conversão de capturas para PEGs.

Como capturas são construções indispensáveis em qualquer aplicação prática de regexes, nesta seção apresentamos uma alternativa de conversão de capturas cujo resultado é uma PEG de LPeg.

Capturas são operações ortogonais ao casamento, isto é, a presença de capturas em uma expressão não modifica a forma como a expressão casa. A implementação de Perl disponibiliza capturas que nos permitem obter trechos da entrada que casaram com uma dada subexpressão. Durante todo o texto, utilizamos o termo captura *simples* para denotar esse tipo de captura. Esta captura é a mais comum entre as implementações de regexes, estando presente também em PCRE, Python, Ruby e Lua. A implementação de Lua disponibiliza outra captura, chamada captura de *posição*, que obtém a posição da entrada que ocorreu o casamento. Uma característica das implementações de regexes é que as capturas são estáticas, isto é, cada captura presente na

expressão corresponde a um valor capturado. Com isso podemos saber, de antemão, quantas capturas serão produzidas ao casar uma dada expressão.

LPeg também possui capturas equivalentes ao que chamamos de capturas simples dos regexes. Além disso, LPeg também possui outros tipos de capturas como, por exemplo, capturas de posição (análogas às de Lua) e capturas constantes (que sempre produzem um determinado valor). Diferente dos regexes, em LPeg, as capturas são dinâmicas. Isso significa que LPeg permite que uma captura produza um número variado de valores dependendo do trecho casado. Por exemplo, considere a PEG `{a}+`. O casamento desta PEG com a entrada `"aaa"` resulta nas três capturas `"a"`, `"a"` e `"a"`.

A conversão de capturas apresenta um problema interessante. Como a função Π converte uma expressão explicitando sua continuação, a conversão pode quebrar uma captura. Para ilustrar este problema, considere a expressão `{a(b|c)}d`. Nessa expressão, utilizamos os metacaracteres `"{"` e `"}"` para demilitar, respectivamente, início de captura e fim de captura, enquanto que a expressão `(b|c)` é apenas um agrupamento que não produz capturas. Abaixo, mostramos passo a passo a conversão dessa expressão:

$$
\begin{aligned}
\Pi(\ \{\texttt{a(b | c)}\}\texttt{d}\ ,\ \texttt{""}\ ) &= \Pi(\ \{\texttt{a(b | c)}\},\ \Pi(\ \texttt{d},\ \texttt{""}\ )\ ) \\
&= \Pi(\ \{\texttt{a(b | c)}\},\ \texttt{d}\ ) \\
&= \Pi(\ \{\texttt{a(b | c)},\ \Pi(\ \},\texttt{d})\ ) \\
&= \Pi(\ \{\texttt{a(b | c)},\ \}\texttt{d}\ ) \\
&= \Pi(\ \{\texttt{a},\ \Pi(\texttt{(b | c)},\ \}\texttt{d})\ ) \\
&= \Pi(\ \{\texttt{a},\ \Pi(\texttt{b},\ \}\texttt{d})\ /\ \Pi(\texttt{c},\ \}\texttt{d})\ ) \\
&= \Pi(\ \{\texttt{a},\ \texttt{b}\}\texttt{d}\ /\ \texttt{c}\}\texttt{d}\ ) \\
&= \{\texttt{a(}\ \texttt{b}\}\texttt{d}\ /\ \texttt{c}\}\texttt{d)}
\end{aligned}
$$

Ao convertermos a expressão {a(b | c)}d, a função Π distribui os delimitadores de final de capturas entre as alternativas. Assim, a PEG resultante {a( b}d / c}d ) possui os delimitadores de capturas estaticamente desbalanceados. Dessa forma, as capturas claramente não podem ser convertidas diretamente para as capturas simples de LPeg.

Para resolver esse problema, a conversão de capturas que propomos é dividida em três etapas. A primeira etapa converte o operador () nos delimitadores "{$_{\texttt{i}}$" e "}". O delimitador "{$_{\texttt{i}}$" define o início de uma captura, onde i é o identificador da captura. O delimitador "}" define o fim da última captura aberta.

Em LPeg, o delimitador "{$_{\texttt{i}}$" é representado pelo padrão {$_c$"open", i}{}. Este padrão é composto pela captura constante {$_c$"open", i} que casa com a string vazia e produz a string "open" e o identificador da captura i. Este padrão é seguido pela captura de posição {}, que casa com a string vazia e retorna a posição da entrada onde ocorreu o casamento [2]. De forma análoga, o delimitador "}" é representado em LPeg pelo padrão {$_c$"close"}{}. O padrão {$_c$"close"} casa com a string vazia e produz a string "close" enquanto que o padrão {} retorna a posição do casamento [3].

O caso abaixo ilustra a conversão do operador () nos delimitadores propostos.

$$\Pi(\ \texttt{(p)},\ \texttt{k}\ ) = \Pi(\ \{_{\texttt{i}}\texttt{p}\},\ \texttt{k}\ ) \tag{4-21}$$

Após a conversão da expressão, realizamos o casamento da PEG resultante sobre a entrada. Este casamento resulta em uma lista de valores capturados pelos delimitadores.

[2]Na sintaxe original de LPeg, "{$_i$" é traduzido para `lpeg.Cc("open", i) * lpeg.Cp()`
[3]Na sintaxe original de LPeg, "}" é traduzido para `lpeg.Cc("close") * lpeg.Cp()`

Como exemplo, considere a conversão da expressão `{a(b|c)}d` que resulta na PEG `{`$_1$`a( b}d / c}d )`. Ao casarmos esta PEG com a entrada `"abd"`, teremos como resultado a lista de valores [`"open"`, `1`, `1`, `"close"`, `2`]. Os três primeiros valores desta lista, `"open"`, 1 e 1, foram gerados pela PEG equivalente ao delimitador `"{`$_1$`"`. A string `"open"` sinaliza que começou uma nova captura; o primeiro valor 1 representa o identificador desta captura e o segundo valor 1 representa a posição da entrada em que a captura começou. Os dois últimos valores dessa lista, `"close"` e 2, foram produzidos pela PEG equivalente ao delimitador `"}"`. A string `"close"` sinaliza o fim da captura recentemente aberta e o valor 2 é a posição em que esta captura terminou.

A segunda etapa da nossa conversão é feita após o casamento da PEG com a entrada. Esta etapa consiste em iterar sobre a lista de capturas resultante do casamento e construir uma *tabela de índices* composta pelas colunas *posição inicial* e *posição final*. Para cada sequência [`"open"`, `i`, `p`] preenchemos a coluna *posição inicial* com o valor `p` na linha `i`. Para cada sequência [`"close"`, `p`] preenchemos a coluna *posição final* com o valor `p` na linha correspondente a última captura aberta.

Para ilustrar como essas capturas funcionam em expressões que possuem capturas aninhadas, considere a PEG abaixo, resultante da conversão de `(a(b)*(c))`:

```
S  ←  {1 a A
A  ←  {2 b} A  /  {3c}  }
```

Ao casarmos a PEG acima com a entrada `"abbc"` teremos a lista de valores [`"open"`, `1`, `1`, `"open"`, `2`, `2`, `"close"`, `3`, `"open"`, `2`, `3`, `"close"`, `4`, `"open"`, `3`, `4`, `"close"`, `5` `"close"`, `5`].

| i | posição inicial | posição final |
|---|---|---|
| 1 | 1 | 5 |
| 2 | 3 | 4 |
| 3 | 4 | 5 |

Tabela 4.1: Tabela de índices

É importante frisar que a forma como construímos a tabela de índices utiliza a ordem das linhas como a ordem das capturas.

Por fim, a terceira etapa extrai as subcadeias definidas pela tabela de índices. Se considerarmos a tabela de índices do exemplo anterior teremos

como resultado a tabela de capturas ilustrada pela tabela abaixo.

| i | valor capturado |
|---|---|
| 1 | `"abbc"` |
| 2 | `"b"` |
| 3 | `"c"` |

Tabela 4.2: Tabela de capturas

Ao realizarmos essas 3 etapas, obtemos exatamente as mesmas capturas definidas na expressão original.

## 4.5 Backreferences

Backreferences são construções intimamente relacionadas com as capturas. Para permitirmos backreferences, é necessário que haja capturas na expressão. Além disso, diferente das capturas, backreferences modificam o casamento de uma expressão, ou seja, dependendo dos valores capturados, backreferences casam de forma diferente. Por exemplo, a expressão `(\w*)\s\1` casa com duas palavras repetidas separadas por um espaço, porém dependendo do valor capturado por `(\w*)`, a subexpressão `\1` casa de forma diferente.

Assim, da mesma forma que capturas não podem ser convertidas para PEGs, backreferences também não podem, por serem construções dependentes das capturas.

Uma alternativa estudada foi converter backreferences de modo a obter os valores da tabela de índices provenientes da conversão de capturas, em tempo de casamento, verificando se o valor capturado é igual ao trecho que deveria casar com o backreference. Essa estratégia é semelhante à utilizada pelo módulo `re` (Ier09), que compõe a biblioteca LPeg. Porém, para adaptarmos essa estratégia em nossa conversão, teríamos que disponibilizar a tabela de índices das capturas em tempo de casamento, ao invés de construí-la após o casamento. Essa pequena diferença resulta em uma implementação ineficiente, já que construir a tabela de índices em tempo de casamento requer a utilização da função `lpeg.Cmt` (*runtime capture*) ao casarmos os delimitadores `"{`$_{\texttt{i}}$`"` e `"}"`. Em expressões com muitas capturas, esse impacto é visível no tempo de casamento, pois a função `lpeg.Cmt` é executada diversas vezes.

A solução que estudamos resolvia backreferences através de código Lua ao invés de PEGs, o que distancia do foco deste trabalho, que é estudar conversões formais para PEGs. Dessa forma, não apresentamos nenhuma conversão de backreferences neste trabalho.

## 4.6 Lookbehind

Lookbehinds e backreferences, em particular, possuem uma característica em comum. Ambas as construções dependem de trechos que antecedem a posição atual do casamento para decidirem o resultado do casamento. Dessa forma, a conversão de lookbehinds para PEGs não é trivial, pois PEGs não possuem operadores que nos permitem obter (ou recordar) os trechos que antecedem a posição atual do casamento.

A conversão de lookbehinds para PEGs ainda é um problema em aberto. Neste trabalho não conseguimos produzir nenhuma conversão de lookbehinds em PEGs nem provar que é impossível converter lookbehinds em PEGs.

Para poder converter lookbehinds em PEGs, propomos uma extensão para PEGs. Esta extensão consiste em um novo predicado sintático chamado *back*, denotado por `<p`. Este possível predicado consiste em retroceder um caracter na entrada e casar a PEG `p`. Assim como outros predicados, este não consome caracteres.

Como exemplo, considere a expressão regex `(?<=ola)`. Com o predicado *back* podemos construir a PEG `(<(<(<o)l)a)`, que é equivalente a essa expressão. Para ilustrar como funciona este predicado sintático, os passos abaixo exibem o casamento dessa PEG com a entrada `"ola "`. Neste exemplo, sublinhamos o caracter que está na posição atual do casamento. Considere que, inicialmente, o casamento está na posição `"ola_"`:

$$\texttt{(<(<(<o)l)a)} \rightsquigarrow \texttt{"ola\_"} \quad (1)$$
$$\texttt{(<(<o)l)a} \rightsquigarrow \texttt{"ol\underline{a} "} \quad (2)$$
$$\texttt{(<o)l} \rightsquigarrow \texttt{"o\underline{l}a "} \quad (3)$$
$$\texttt{o} \rightsquigarrow \texttt{"\underline{o}la "} \quad (4)$$

Neste exemplo, utilizamos a notação $\texttt{p} \rightsquigarrow \texttt{s}$ para representar o casamento da expressão `p` com a entrada `s`. Em (1) apenas ilustramos o estado inicial antes de casarmos a PEG com a entrada `"ola_"`. Para que a expressão inteira case com a entrada, a PEG retrocede um caracter da entrada e tenta casar a expressão de parsing `(<(<o)l)a`, ilustramos esse passo em (2). De forma análoga, para que a expressão `(<(<o)l)a` case com a entrada `"ol`$\underline{\texttt{a}}$`"`, a PEG retrocede mais um caracter e tenta casar a expressão `(<o)l` com a entrada `"o`$\underline{\texttt{l}}$`a "`, ilustramos esse passo em (3). Finalmente, em (4), a PEG retrocede mais um caracter e tenta casar o caracter `"o"`; se esse casamento suceder, a PEG volta para a posição sinalizada no caso (3) e tenta casar o restante da

expressão, no caso o caracter `"l"`; se essa verificação também suceder, a PEG volta para a posição sinalizada no caso (2) e tenta casar com o caracter `"a"`. Ao final desse processo, se `"a"` casar então a PEG inteira casou com a entrada.

Com este possível predicado *back*, que retrocede apenas um caracter da entrada, podemos construir PEGs equivalentes aos lookbehinds de Perl. Como os lookbehinds de Perl permitem apenas expressões de tamanho "fixo", é possível sabermos quantos caracteres devemos retroceder na entrada. Assim, é possível criarmos uma PEG que retrocede este mesmo número de caracteres.

Este predicado também permite criarmos PEGs equivalentes a algumas construções, como `\b`, que casa com limites de palavras [4] (*boundaries*). A âncora `\b` casa com uma posição que não é precedida por `\W` e é seguida por um caracter `\w`. Com o predicado *back* podemos construir a PEG `!(<\W)&\w` que possui este comportamento. Além disso, `\b` também casa com uma posição que é precedida por um caracter que casa com `\w` e que não é sucedida por um `\W`. A PEG que possui este comportamento é `<\w!\W`. O predicado *back* também permite criarmos PEGs equivalentes a algumas âncoras, como as que casam com início de entrada e de linha. Na próxima seção tratamos com mais detalhes estas âncoras.

Embora a semântica do predicado sintático proposto seja simples, não estudamos as implicações dessa extensão em PEGs. Um estudo mais profundo deve ser feito para verificar quais propriedades de PEGs são preservadas e quais são alteradas. Por exemplo, não sabemos se a adição desse predicado sintático permitiria que PEGs reconheçam mais classes de linguagens. Com isso, não sabemos se as PELs (*Parsing Expression Languages*), isto é, as linguagens reconhecidas pelas PEGs, continuariam mantendo as propriedades de serem fechadas sob as operações de união, interseção e complemento (For04).

Outro aspecto interessante a ser estudado é se PEGs com o predicado *back* poderiam reconhecer qualquer entrada em tempo linear (For02).

Uma das limitações deste predicado é permitir que determinados casamentos entrem em loop infinito. Por exemplo, ao casarmos a PEG `S` ← `a (<S)` com a entrada `"a"`, a PEG casa com o caracter `"a"` e, logo em seguida, retrocede um caracter e repete este casamento.

O estudo mais detalhado das implicações do predicado *back* nas propriedades de PEGs se distancia do escopo deste trabalho e deixamos como sugestão para um possível trabalho futuro.

[4] Perl define "palavra" como um caracter que casa com `\w`.

## 4.7 Âncoras de início

A busca de um padrão na entrada é feita pelo motor regex de forma externa à expressão. Basicamente, o que o motor faz é tentar casar a expressão com o início da entrada; se falhar, o motor avança um caracter da entrada e executa o casamento novamente. Esse processo se repete até o motor encontrar a primeira ocorrência do padrão ou até chegar no fim da entrada. Dessa forma, a implementação de âncoras que casam com o início da entrada é trivial nos regexes; basta o motor tentar casar o padrão apenas uma vez.

PEGs, ao contrário de regexes, reconhecem um padrão em modo ancorado. Implementações de PEGs, como LPeg, também reconhecem o padrão em modo ancorado, assim como descrito no formalismo original. Se a busca de um padrão for desejada, é muito fácil construimos uma PEG que procura a primeira ocorrência do padrão na entrada. Por exemplo, dada uma PEG `p` podemos reescrevê-la na forma `S` $\leftarrow$ `p / . S` e, com isso, buscar o padrão `p` na entrada.

Como podemos ver, regexes e PEGs se diferenciam quanto a busca de um padrão. Enquanto os regexes fazem a busca de forma externa à expressão, em PEGs a busca pode ser explicitamente descrita pela PEG.

A maioria dos usos práticos da âncora que casa com início da entrada consiste em expressões que possuem esta âncora apenas no início da expressão, como em `^p` e `^(`$p_1$`|`$p_2$`|...|`$p_n$`)`. Nos livros *Mastering Regular Expressions* (Fri06) e *Regular Expressions Cookbook* (GL09), por exemplo, todos os usos práticos que contêm estas âncoras as utilizam no início da expressão. Em expressões neste formato, é fácil verificarmos antes da conversão se é necessário realizar a busca da PEG resultante; se houver âncora no início da expressão, basta não realizar a busca da PEG. Porém, regexes permitem expressões em que a âncora pode estar no interior da expressão, como `a|^b|c`. Como a busca é feita externamente a métodos de conversão apresentados neste trabalho, ao convertermos uma expressão que contém âncoras no seu interior, não podemos simplesmente decidir se fazemos busca ou não da PEG resultante pois esta não preservaria a semântica da expressão.

Uma alternativa que possibilita a conversão de âncoras em posição ambitrária consiste em utilizar o predicado sintático *back* (`<`) proposto na seção anterior. Com esse predicado podemos criar a PEG `!(<.)`, que casa apenas se a posição corrente **não** for precedida por nenhum caracter, ou seja, que casa com o início da entrada. Com isso, podemos converter todos os casos de âncoras de início, inclusive expressões como `a|^b|c`, que possuem âncoras no interior da expressão.

Âncoras que casam com início de linha também são casos particulares de lookbehinds. Para permitirmos a conversão destas âncoras, podemos utilizar novamente o predicado *back*. Por exemplo, para casarmos a string `"hello"` no início de uma linha, podemos construir a PEG `(<"\n" / !(<.))"hello"`, que casa com `"hello"` apenas se for precedido por um fim de linha ou se estiver no início da entrada.