2
Regexes

Expressões regulares são um formalismo que descreve linguagens regulares através de uma notação algébrica. A tabela 2.1 lista os operadores de expressões regulares.

e	L(e)	Operadores
ϕ	ϕ	linguagem vazia
ε	{""}	string vazia
a	{"a"}	símbolo do alfabeto
$e_1 \mid e_2$	$L(e_1) \cup L(e_2)$	alternativa
$e_1\, e_2$	$L(e_1)\, L(e_2)$	concatenação
e^*	$L(e)^*$	repetição

Tabela 2.1: Expressões regulares

A construção ϕ é utilizada apenas para permitir a definição da linguagem vazia.

Devido à sua forma declarativa, expressões regulares são amplamente utilizadas como entrada de ferramentas de casamento de padrões. Porém, expressões regulares puras são muito limitadas para descrever vários padrões na prática.

A ausência de um operador de complemento faz com que linguagens aparentemente simples sejam surpreendemente difíceis de serem definidas. Um exemplo típico é o padrão que reconhece comentários da linguagem C; outro exemplo comum são os identificadores de C que devem excluir palavras reservadas como `int` e `for`.

Outra limitação de expressões regulares é o seu foco apenas no reconhecimento de strings. Na maioria dos casos, não é suficiente saber se uma expressão casou ou não com a entrada, mas é necessário saber *como* cada subexpressão casou. Isto é, ao casar a expressão com a entrada, além do casamento responder a pergunta *"a expressão casou com a entrada?"*, o casamento deve ser capaz de responder *"com quais partes da entrada cada subexpressão casou?"*.

Assim, diversas linguagens de script possuem bibliotecas de casamento de padrões baseadas em expressões regulares extendidas – *regexes* – compostos da

combinação de operações de expressões regulares e de construções que focam em problemas específicos de casamento de padrões.

Neste capítulo, apresentamos um estudo dos regexes de Perl (WS00), Python (Lut06), Ruby (TFH09) e Lua (Ier06). Para fins de completude, também analisamos os regexes da biblioteca PCRE (Haz09), que é composta por um subconjunto de construções com a mesma sintaxe e semântica de Perl.

A partir da lista de regexes presente na "bíblia"de Perl (WS00), separamos as construções regexes em dois conjuntos. O primeiro conjunto consiste nas construções mais comuns entre as implementações de regexes listadas na tabela 2.2. São estas construções que estudamos com detalhes no decorrer desse capítulo. No capítulo 4, apresentamos a conversão de algumas construções desta lista para PEGs e, para as que não criamos conversões, justificamos a ausência de conversão. O segundo conjunto, listado na tabela 2.9, apresenta o restante das construções que completam a lista dos regexes. Nosso estudo aborda brevemente essas construções para fins de completude, mesmo que estas sejam pouco relevantes para o foco deste trabalho que é a conversão para PEGs. Na seção 2.9, justificamos porque não abordamos estas construções.

Construção	Sintaxe em Perl
Expressão independente	`(?>e)`
Âncoras iniciais	∧, `\A`
Âncoras finais	`$`, `\Z`, `\z`
Quantificadores gulosos	`e*`, `e+`, `e?`, `e{n}`, `e{n,}`, `e{n,m}`
Quantificadores preguiçosos	`e*?`, `e+?`, `e??`, `e{n}?`, `e{n,}?`, `e{n,m}?`
Quantificadores possessivos	`e*+`, `e++`, `e?+`, `e{n}+`, `e{n,}+`, `e{n,m}+`
Lookahead positivo	`(?=e)`
Lookahead negativo	`(?!e)`
Lookbehind positivo	`(?<=e)`
Lookbehind negativo	`(?<!e)`
Capturas	`(e)`
Backreferences	`\n`, onde `n` é um dígito
Parênteses sem capturas	`(?:e)`
Comentário	`(?#comentário)`
Classes de caracteres	`[...]`

Tabela 2.2: Construções mais comuns entre os regexes

As versões das bibliotecas abordadas nesse trabalho são: Perl 5.10.0; PCRE 8.02; Python 2.6.4; Ruby 1.8.7; Lua 5.1.

Ao longo desse capítulo utilizamos o termo *motor* para denotar uma implementação de casamento de padrão.

2.1 Expressão independente

Expressão independente é uma construção que permite o casamento de subexpressões de forma independente do casamento de expressões que as contém. Ao casar uma expressão independente, os trechos da entrada que foram consumidos por essa subexpressão não estão sujeitos a backtracking, mesmo se isso resultar na falha do casamento da expressão como um todo. Perl utiliza a sintaxe `(?>e)`, onde `e` é uma expressão, para denotar expressões independentes. Essa construção está presente nos regexes de Perl, PCRE e Ruby.

A principal vantagem de permitir que subexpressões casem de forma independente é possibilitar a otimização de expressões. Como exemplo, considere a expressão `\d+\b` onde `\d` casa com dígitos e `\b` casa com *fronteiras* de palavras. Quando casamos essa expressão com a entrada `"12345abc"`, a subexpressão `\d+` casa com a substring `"12345"` e o padrão `\b` falha ao casar entre `"5"` e `"a"`. Logo em seguida, o motor regex desfaz a última repetição de `\d+` e tenta casar `\b` entre `4` e `5` sendo que falha novamente. Como `\b` falhou, o motor regex continua tentando inutilmente todas as possibilidades até informar que a expressão não casa com a entrada.

Se reescrevermos a expressão anterior para `(?>\d+)\b` podemos evitar esses backtrackings inúteis. Como a subexpressão `(?>\d+)` é uma expressão independente, não há backtrackings para partes por ela consumidas, no caso `"12345"`. Isso evita que o motor regex tente casar sequências menores, que não levariam ao sucesso do casamento da expressão, por exemplo, casar `\b` entre `5` e `4`, entre `4` e `3` e assim por diante.

Uma expressão independente pode excluir casamentos que, sem ela, seriam possíveis. Como exemplo, considere a expressão `\d+\d?\d+` que casa com qualquer cadeia de dígitos de no mínimo dois dígitos. Se modificamos a subexpressão `\d?` para `(?>\d?)`, a expressão inteira passa a casar com cadeias que tenham no mínimo três digitos, excluindo as cadeias de tamanho dois que antes eram aceitas.

2.2 Quantificadores

Quantificadores são construções utilizadas para expressar repetições de uma determinada expressão. As implementações abordadas possuem dois tipos

de quantificadores, são eles: *gulosos* e *preguiçosos*. Perl e PCRE disponibilizam mais um tipo de quantificador, chamado *possessivo*.

Na tabela 2.3 apresentamos os quantificadores presentes em Perl, PCRE, Python e Ruby.

Guloso	Preguiçoso	Possessivo	Descrição
`{n,m}`	`{n,m}?`	`{n,m}+`	De `n` vezes até no máximo `m`
`{n,}`	`{n,}?`	`{n,}+`	Mínimo de `n` ocorrências
`{n}`	`{n}?`	`{n}+`	Exatamente `n` ocorrências
`*`	`*?`	`*+`	0 ou mais ocorrências
`+`	`+?`	`++`	1 ou mais ocorrências
`?`	`??`	`?+`	0 ou 1 ocorrência

Tabela 2.3: Quantificadores

Os quantificadores gulosos repetem a expressão o maior número de vezes possível. Esses quantificadores são *não-cegos*, isto é, a expressão será repetida de modo que a expressão inteira case. Como exemplo, considere a expressão `.*10` que casa com uma string que termina com a string `"10"`. Ao casar essa expressão com a entrada `"Julho de 2010"`, a subexpresssão `.*` casa com a substring `"Julho de 20"` para que o resto da entrada, `"10"`, case com a expressão `10`.

Os quantificadores preguiçosos repetem a expressão o menor número de vezes possível e, assim como os gulosos, também são *não-cegos*. Como exemplo, considere que seja necessário criar uma expressão que case com um elemento de HTML. Uma solução ingênua seria uma expressão com uma repetição gulosa como `<b>.*</b>`. Ao casarmos essa expressão com a string `"<b>primeiro</b> e <b>segundo</b>"`, a repetição `.*` casa com a substring `"primeiro</b> e <b>segundo"`, ao invés de casar com o resultado desejado, que é `"primeiro"`. Quantificadores preguiçosos permitem a criação de uma expressão adequada para essa tarefa. Na expressão `<b>.*?</b>`, a repetição `.*?` irá repetir o menor número de vezes possível de modo que o restante da expressão também case. Dessa forma, a subexpressão `.*?` casará apenas com a string `"primeiro"`.

Quantificadores possessivos repetem o maior número de vezes possível e são *cegos*, isto é, o maior número de repetições é realizado mesmo que isso resulte na falha do casamento da entrada como um todo. Por exemplo, a expressão `a*+a` em Perl não casa com nenhuma sequência de `a`'s pois, ao realizarmos o casamento, a subexpressão `a*+`, por ser possessiva, consumira todos os `a`'s.

A documentação de Perl define que a implementação de um quantificador possessivo não realiza backtracking para partes que foram consumi-

das. Contudo, a documentação não é muito clara quanto ao funcionamento da repetição. Podemos interpretar uma repetição como *"repita a expressão o máximo possível"* ou *"repita de forma a consumir mais caracteres"*. Os quantificadores possessivos de Perl misturam esses dois conceitos. Por exemplo, a casarmos a expressão `(ab|a)++b` com a string `"aaab"`, a subexpressão quantificada `(ab|a)++` consome todos os caracteres da entrada de modo que o restante do padrão, no caso `b`, não casa com nada, resultando na falha do casamento da expressão inteira. Porém, se alterarmos a ordem das alternativas, a expressão `(a|ab)++b` passa a aceitar a string `"aaab"`, pois essa alteração faz com que a expressão repita o mesmo número de vezes, mas sem consumir o maior número de caracteres.

Lua possui quantificadores com sintaxe e semântica um pouco diferente das outras bibliotecas. Em Lua, os quantificadores são aplicáveis apenas a classes de caracteres. Desse modo, expressões do tipo `a*` e `%w*` são válidas enquanto `(ab)*` não é. Lua tem suporte a repetições gulosas e preguiçosas. Na tabela 2.4 apresentamos os quantificadores de Lua.

Tipo	Quantificador	Descrição
guloso	`*`	0 ou mais ocorrências
guloso	`+`	1 ou mais ocorrências
guloso	`?`	0 ou 1 ocorrência
preguiçoso	`-`	0 ou mais ocorrências

Tabela 2.4: Quantificadores de Lua

2.3 Capturas

Capturas são construções que permitem obtermos trechos da entrada que casaram com uma determinada subexpressão.

Para realizarmos uma captura, utilizamos a sintaxe `(e)` onde `e` é uma expressão regex e, com isso, todo trecho da entrada que casar com esta expressão será capturado. Por exemplo, ao casarmos a expressão que reconhece url's `http://([a-z0-9.-_]+)` com a string `"http://www.lua.org"`, o motor regex reconhece a string inteira e retorna a captura da substring `"www.lua.org"`.

Cada captura é identificada por um índice numérico (que começa com o valor 1) atribuído de forma estática da esquerda para a direita, em ordem crescente. Como exemplo, considere a expressão `((a)(b))|(c)`; a captura da subexpressão `((a)(b))` possui índice 1, `(a)` possui índice 2, `(b)` possui índice 3 e, finalmente, `(c)` possui o índice 4. Ao quantificarmos um captura, como

`(e)*`, onde `e` é um padrão qualquer, o valor capturado será o resultante da última repetição de `e`.

Perl e Ruby armazenam as capturas em variáveis com a sintaxe `$n`, onde `n` é o índice da captura. Em Python, o casamento de uma expressão resulta em um objeto do tipo `MatchObject`. A partir desse objeto, podemos obter os valores capturados utilizando o método `m.group(n)`, passando como argumento o índice da captura. Em Lua, o casamento de uma expressão retorna todas as capturas na ordem em que foram definidas. Lua também possui *capturas de posição* que são denotadas por uma captura vazia `()`. Essa captura retorna a posição atual do casamento.

Para permitir capturas, as implementações precisam quebrar o casamento em partes, sendo que cada parte corresponde ao casamento de uma subexpressão. A idéia principal por trás dessa implementação é permitir que o motor regex saiba *como* cada subexpressão casou e, com isso, obter os valores que casaram com cada subexpressão. Como exemplo, considere a expressão regular `a*` que define a linguagem $\{\varepsilon$, a, aa, aaa, ...$\}$. Ao realizarmos o casamento da expressão `a*` sobre a string `"aa"`, uma implementação poderia casar tanto com a substring `"a"`, quanto com `"aa"` ou com a string vazia, já que todas pertencem a linguagem definida pela expressão `a*`. Para que o casamento ocorra de forma determinística, as implementações de repetição sempre repetem o maior número de vezes (guloso) ou o menor número de vezes (preguiçoso) de forma que não haja ambiguidade ao casar repetições.

Para permitir que os motores casem de forma determinística, a maioria das implementações de regexes são baseadas em backtracking. Com isso, o motor pode realizar a leitura da entrada diversas vezes antes de aceitá-la. Esse comportamento resulta, no pior caso, em casamentos com tempo exponencial (Cox07).

2.4 Backreferences

Backreferences são construções utilizadas para referenciarmos na própria expressão os valores capturados durante um casamento. Perl, PCRE, Python e Ruby utilizam a sintaxe `\n` para denotar backreferences, onde `n` corresponde ao índice da captura. Lua utiliza a sintaxe `%n`, onde `n` é o índice da captura referenciada.

Para exemplificar o uso de backreferences, considere a expressão `(["']).*?\1`, que reconhece citações dentro de um texto. Para reconhecermos aspas duplas ou aspas simples, que iniciam uma citação, utilizamos a subexpressão `(["'])` e capturamos o seu valor; logo em seguida, a repetição

preguiçosa `.*?` reconhece o texto da citação e, por fim, o backreference \1 reconhece o mesmo caracter capturado na subexpressão `(["'])`, indicando o fim da citação.

As documentações de Perl, PCRE e Ruby apresentam exemplos muito simples e não descrevem qual o comportamento de backreferences quando combinados com outras construções. Por exemplo, podemos usar backreferences dentro de classes de caracteres? Backreferences podem ser quantificados? É possível capturar backreferences? Ao testarmos essas combinações, constatamos que nenhuma das bibliotecas permite o uso de backreferences dentro de classes de caracteres, mas, apenas a documentação de Python explicita que essa combinação não possui semântica. Também constatamos que é possível quantificarmos backreferences, por exemplo, a expressão `(a)\1*` que casa com toda cadeia `"aaa"`. Em Lua, a quantificação de backreferences não é permitida, mas a documentação define claramente que só é possível quantificar classes de caracteres. Por fim, todas as bibliotecas permitem capturas de backreferences e, consequentemente, backreferences dessas capturas. Um exemplo que ilustra esse caso é a expressão `([a-z]+),(\1),\2` que casa com cadeias que possuem três palavras iguais separadas por vírgulas.

Backreferences são exemplos de construções que estendem o poder de reconhecimento de expressões regulares. Por exemplo, a expressão `(a+)(b+)\1\2` reconhece a linguagem $\{ a^i b^j a^i b^j \mid i, j > 0 \}$, que não pertence à classe de linguagens livres de contexto e, consequentemente, também não pertence à classe de linguagens regulares.

Implementações que disponibilizam backreferences podem levar tempo exponencial para executar o casamento da expressão com a entrada. Esse problema é incontornável, já que o problema 3-SAT pode ser reduzido para regexes com backreferences e, portanto, resolver esse casamento é um problema NP-Completo (Aho90).

2.5 Âncoras

Outra forma de estender expressões regulares é através de âncoras, construções que permitem determinar onde o padrão deve casar. Na literatura, âncoras são definidas como *assertivas de tamanho zero*: tamanho zero por serem construções que não consomem caracteres e assertiva por testarem uma propriedade do casamento.

A tabela 2.5 lista o conjunto de âncoras presente nos regexes abordados:

As âncoras de Perl e PCRE são idênticas. Já as âncoras de Ruby são um subconjunto das âncoras de Perl, com uma pequena diferença: em Perl, as

Âncora	Perl / PCRE	Python	Ruby	Lua
Início da entrada	∧, \A	∧, \A	\A	∧
Fim da entrada	$, \z	$, \Z	$, \z	$
Início de linha			∧	
Fim de linha			$	
Fim da entrada (antes de fim de linha opcional)	\Z		\Z	

Tabela 2.5: Âncoras

âncoras ∧ e $ casam, respectivamente, com início e fim da entrada, enquanto que, em Ruby, as âncoras ∧ e $, além de casarem com início e fim de entrada, também casam com início e fim da linha.

Em Python, as âncoras são semelhantes às de Perl, com exceção da âncora \Z. A âncora \z de Perl, que casa com o fim da entrada, possui a sintaxe \Z em Python. Essa diferença, apesar de ser apenas sintática, pode gerar confusão em usuários das duas linguagens, já que em Perl a sintaxe \Z já é utilizada para denotar outra âncora.

Diferente das outras implementações, Lua possui um conjunto de âncoras formado apenas pelos metacaracteres ∧ e $, que casam, respectivamente, com o início e final da entrada.

2.6 Lookahead

Lookaheads são construções que verificam se o trecho que sucede a posição atual do casamento casa ou não com um determinado padrão, sem consumir caracteres. Perl utiliza a sintaxe `(?=e)`, onde `e` é uma expressão regex, para denotar o tipo de lookahead, chamado *positivo*.

Como exemplo, podemos utilizar lookaheads para localizar nomes de arquivos que terminam com a extensão `".jpg"` usando a expressão `\w+(?=\.jpg)`, onde `\w` é um padrão que casa com qualquer caracter alfanumérico. Note que é necessário "escapar" o caracter "." já que este corresponde ao padrão que casa qualquer caracter. Também podemos localizar palavras em um texto que são sucedidas por uma virgula usando a expressão `\w+(?=,)`.

O outro tipo de lookahead, chamado *negativo*, utiliza a sintaxe `(?!e)`. Em um lookahead negativo, a expressão inteira casa apenas se a expressão de lookahead **não** casar a partir da posição atual do casamento. Um exemplo

simples é a expressão `\w+(?!.*,)` que verifica qual a última palavra em uma lista de palavras separadas por vírgulas.

Lookaheads, assim como as âncoras, são *assertivas de tamanho zero*, já que verificam uma determinada característica do casamento sem consumir caracteres. Note que âncoras de fim de entrada e fim de linha são casos particulares de lookaheads. Podemos expressar âncoras de fim de entrada através do lookahead negativo `(?!.)` e âncoras de fim de linha usando o lookahead positivo `(?=\n)`.

Algumas expressões apresentam comportamentos inesperados quando combinamos lookaheads com capturas. Expressões com capturas utilizadas em lookaheads positivos realizam suas capturas normalmente, porém em lookaheads negativos as capturas são ignoradas. Esse comportamento está presente em todas as bibliotecas que possuem lookahead (Perl, PCRE, Python e Ruby) mas nenhuma de suas documentações descreve esse comportamento.

As bibliotecas de Perl, PCRE e Python permitem a quantificação de lookaheads. Embora sejam possíveis, a quantificação de lookaheads não tem usos práticos, já que são construções que não consomem caracteres. Desse modo, em uma expressão `\w+(?=\n){10}`, podemos erroneamente deduzir que essa expressão casa com uma palavra seguida por 10 quebras de linhas. Porém, essa expressão casa com uma palavra seguida por uma quebra de linha pois, como lookaheads não consomem caracteres, todas as 10 repetições da subexpressão `(?=\n)` casam com o mesmo `\n`.

2.7 Lookbehind

Lookbehinds são construções que verificam se o trecho que precede a posição atual de casamento casa ou não com um determinado padrão, sem consumir caracteres. Essa construção, introduzida em Perl, também está presente nos regexes de PCRE e Python. As bibliotecas de Ruby e Lua não disponibilizam lookbehinds.

Lookbehinds positivos utilizam a sintaxe `(?<=e)` onde `e` é uma expressão regex usada na verificação dos caracteres que precedem a posição atual. Lookbehinds negativos utilizam a sintaxe `(?<!e)`. Assim como lookaheads e âncoras, lookbehinds também são *assertivas de tamanho zero*.

Âncoras iniciais são casos particulares de lookbehinds. Podemos construir uma expressão equivalente à âncora de início de entrada usando o lookbehind negativo `(?<!.)`, que casa apenas se nenhum caracter preceder a posição atual. É possível verificar um início de linha usando o lookbehind positivo `(?<=\n)`.

Lookbehinds de Perl e Python permitem apenas expressões que casam

com um texto de tamanho "fixo". As respectivas documentações não são claras quanto ao significado de "fixo" , mas constatamos através de testes que esta regra exclui expressões como `(?<=a*)bb`, pois `a*` é uma expressão que casa com trechos de tamanho variado. O mesmo vale para expressões como `a?`, `a+`, `a|bc`, `a{2,}` e `a{1,2}`. Esta regra também exclui o uso de expressões com backreferences pois considera que qualquer backreference é uma expressão que casa com trechos de tamanho variado. Sendo assim, expressões como `(abc)(?<=\1)` também não são permitidas. Expressões como `a|b`, que possuem alternativas que casam com trechos do mesmo tamanho e expressões quantificadas do tipo `a{3}` podem ser utilizadas.

Alguns expressões que não são permitidas em Perl e Python, como `(?<!books?)`, podem ser substituidas por `(?<!book)(?<!books)`, que é uma expressão válida, mas que possui a legibilidade prejudicada (Fri06).

Lookbehinds em PCRE permitem o uso de expressões compostas por alternativas que casam com trechos de tamanhos diferentes como, por exemplo, `(?<=book|books)`. Esse comportamento é inesperado pois, segundo a documentação de PCRE, a semântica de suas construções é igual à de Perl.

Visto que lookbehinds são construções que possuem restrições diferentes dependendo da implementação, buscamos outras APIs para fins de comparação. Lookbehinds do regex de Java (Fri06) permitem um subconjunto de expressões maior que o de PCRE, como `(?<=books?)` e `(?<=book|books)` mas expressões como `(?<=\w+)` não são permitidas. O regex da plataforma Microsoft's .NET permite lookbehinds com expressões que casam com trechos de tamanho variado. Essa implementação possui, porém, um potencial problema de eficiência caso lookbehinds sejam utilizados de forma imprudente. Por exemplo, quando uma expressão de tamanho variado é utilizada em lookbehinds, a ferramenta regex é forçada a verificar a expressão do lookbehind desde o início da entrada, o que significa que muito esforço pode ser desperdiçado se o casamento estiver próximo do fim de um texto muito grande (Fri06).

2.8 Classes de caracteres

Classes de caracteres são construções que definem um conjunto de caracteres que devem ser aceitos. Em Perl, PCRE, Python, Ruby e Lua podemos definir uma classe de caracteres através da sintaxe `[...]`. Uma classe pode ser definida através da enumeração dos caracteres que devem ser aceitos, por exemplo, a classe de caracteres `[abc_]` reconhece os caracteres `"a"`, `"b"`, `"c"` e *underline*. Também podemos definir uma classe de caracteres através de um intervalo, como exemplo, a classe `[a-z]` recohece todos os caracteres que

estão entre `"a"` e `"z"`.

Podemos definir uma classe de caracteres utilizando a sintaxe `[^classe]`, que casa com qualquer caracter **não** especificado pela classe. Como exemplo, a classe `[^0-9]` reconhece qualquer caracter que não seja um dígito.

Implementações de casamento de padrões geralmente disponibilizam classes de caracteres comuns com uma sintaxe mais concisa. Por exemplo, classes como `[0-9]` são representadas pela classe pré-definida `\d`. A tabela 2.6 apresenta as classes pré-definidas de Perl.

Sintaxe	Casa com
`\w`	caracteres alfanuméricos acrecido de "_"
`\W`	complemento de `\w`
`\s`	o caracter espaço
`\S`	o complemento de `\s`
`\d`	dígitos
`\D`	o complemento de `\d`

Tabela 2.6: Classes pré-definidas de Perl

A tabela 2.7 lista todas as classes de caracteres pré-definidas de Lua. Além das classes listadas, é possível definirmos o complemento de uma classe utilizando o caracter maiusculo da classe. Por exemplo, `%D` casa com o complemento de `%d`.

Sintaxe	Casa com
`%a`	as letras
`%c`	todos os caracteres de controle
`%d`	dígitos
`%l`	letras minúsculas
`%p`	todos os caracteres de pontuação
`%s`	espaços
`%u`	letras maiúsculas
`%w`	caracteres alfanuméricos
`%x`	caracteres hexadecimais
`%z`	o caracter com representação `0`

Tabela 2.7: Classes pré-definidas de Lua

Em Perl, PCRE, Python e Ruby também é possível utilizarmos as classes de caracteres de POSIX (IG04). A tabela 2.8 apresenta todas as classes de POSIX.

No capítulo 4 não mostramos como essas classes de caracteres são convertidas para PEGs já que sua conversão é direta para classes de caracteres de PEGs. No capítulo 5 apresentamos uma implementação prática do conversor de regexes para PEGs em que a conversão de classes de caracteres está implementada.

Sintaxe	Casa com
`[[:alpha:]]`	Caracteres alfabéticos, equivalente a classe `[a-zA-Z]`
`[[:alnum:]]`	Qualquer caracter alfanumérico
`[[:ascii:]]`	Qualquer caracter ASCII
`[[:blank:]]`	Espaço e tab
`[[:cntrl:]]`	Qualquer caracter de controle
`[[:digit:]]`	Qualquer digito, equivalente a `\d`
`[[:graph:]]`	Qualquer caracter que possui representação gráfica, excluindo espaço
`[[:lower:]]`	Qualquer caracter minúsculo
`[[:print:]]`	Qualquer caracter que possui representação gráfica, incluindo espaço
`[[:punct:]]`	Qualquer caracter que possui representação grática, excluindo alfanuméricos e "_"
`[[:space:]]`	Equivalente a classe `[ \t\r\n\v\f]`
`[[:upper:]]`	Qualquer caracter maiúsculo
`[[:word:]]`	Equivalente a `\w`
`[[:xdigit:]]`	Caracter hexadecimal

Tabela 2.8: Classes de POSIX

2.9 Outras construções

Nessa seção apresentamos o restante das construções que completam a lista dos regexes.

Essas construções não serão abordadas na conversão para PEGs por algumas serem dependentes da linguagem ou por as considerarmos pouco relevantes. Como critério de relevância, utilizamos o número de ocorrências destas construções em três livros que abordam regexes. O primeiro, *Mastering Regular Expressions* (Fri06), apresenta exemplos práticos de expressões regulares em diversas linguagens de programação. O segundo livro, chamado *Regular Expressions Cookbook* (GL09), é uma compilação de vários problemas reais que são solucionados com o uso de regexes. O terceiro livro, chamado *Programming Perl* (WS00), foi escrito pelo próprio autor da linguagem e aborda todas as construções regexes de Perl.

As construções discutidas nessa seção são listadas na tabela 2.9.

A construção de *Branch reset*, denotada pela sintaxe `(?|e)`, permite que capturas em alternativas diferentes sejam numeradas partir do mesmo índice. Como havíamos dito na seção 4.4, as capturas são identificadas através de um índice numérico atribuído da esquerda pra direita. Dessa forma, capturas em alternativas diferentes são numeradas com índices diferentes. O exemplo abaixo ilustra como os índices são atribuídos:

```
(a)  (   (b)|(c)   )  (d)
1    2    3   4       5
```

Nome	Construção	Descrição
Branch reset	`(?|e)`	Permite que a identificação de capturas em alternativas diferentes sejam numeradas a partir do mesmo índice.
Liga/Desliga modificadores	`(?pimsx-imsx:e)`	Habilita/Desabilita o modificador em um agrupamento específico.

Captura nomeada	`(?<nome>e)`
	`(?'nome'e)`
	`(?P<nome>e)`
		Captura que possui um nome como idenfiticador.

Código embutido	`(?{code})`	Permite a execução de um código Perl durante o casamento.
Expressão dinâmica	`(??{code})`	Executa um trecho de código Perl durante o casamento e o valor dessa computação é utilizada como uma expressão regex.
Recursão explícita	`(?n)`	Recursão para o parênteses de índice `n`.
	`(?-n)`	Recursão para o n-ésimo parênteses anterior.
	`(?+n)`	Recursão para o n-ésimo parênteses posterior.
	`(?R) (?0)`	Recursão para início da expressão.
	`(?&nome)`	Recursão para a parênteses com índice `nome`.
	`(?P>nome)`	Recursão para a parênteses com índice `nome`.
Expressão condicional	`(?(cond)`e_1 `|` e_2`)`	Se `cond` for verdadeiro então executa expressão e_1. Caso contrário, executa e_2.

Tabela 2.9: Outras construções

Podemos reescrever a expressão acima utilizando o Branch reset para numerar as capturas de índices 3 e 4 com o mesmo índice. O exemplo abaixo ilustra como essa construção modifica os índices das capturas:

```
(a)  (?| (b)|(c)   )  (d)
1        2    2       3
```

O uso dessa construção só modifica a forma de identificação das capturas. Além de estar presente apenas em Perl, essa construção não modifica o processo de casamento e nem permite a criação de padrões mais expressivos.

A construção `(?pimsx-imsx:e)` modifica o casamento da expressão `e` dependendo do modificador habilitado. O modificador `i` permite que uma dada subexpressão case no modo *case insensitive*. Por exemplo, a expressão `(?i:a)bc` reconhece as strings `"abc"` e `"Abc"`. Em Python, a ocorrência de um modificador em uma subexpressão modifica a expressão inteira. Desse modo, a expressão anterior, `(?i:a)bc`, casa toda a expressão `abc` em modo *case insensitive*. Em Perl, PCRE e Python, as âncoras $\wedge$ e `$`, que casam respectivamente com o início da entrada e fim de entrada,

podem ter seus comportamentos alterados caso o modificador de múltiplas linhas `m` for utilizado; dessa forma, essas âncoras também passam

a casar com o início de linha e fim de linha. A maioria dos exemplos encontrados na literatura utilizam modificadores de forma externa a expressão através da sintaxe `/expressão/modificadores`. Em (Fri06), (GL09) e (WS00) temos ao todo três exemplos de modificadores embutidos.

Captura nomeada é uma captura que utiliza um nome como identificador ao invés de um índice numérico. Apesar da diferença sintática, as capturas nomeadas são semanticamente equivalentes às capturas padrão.

A construção `(?{code})` permite a execução de uma expressão Perl durante o casamento da expressão. De forma análoga, construção `(??{code})` também executa um trecho de código Perl durante o casamento, porém o valor de retorno dessa expressão é utilizado como uma expressão regex. As construções de código embutido `(?{code})` e expressões dinâmicas `(??{code})` não serão abordadas por estarem em estado experimental (Tru).

A construção de recursão explícita permite que o casamento execute novamente uma determinada subexpressão. Permitir recursões em subexpressões resulta em padrões difíceis de prever o casamento. Para saber como uma expressão casou com a entrada bastava saber como cada subexpressão casou, porém, com recursões, é necessário saber em que *nível de recursão* cada subexpressão casou. Além disso, na literatura não há exemplos práticos que incentivem o uso dessa construção. Em (Fri06), (GL09) e (WS00) não temos nenhum exemplo que utiliza essa construção.

Uma expressão condicional `(?(cond)`$\mathtt{e}_1$ | $\mathtt{e}_2$`)` executa a expressão $\mathtt{e}_1$ se `cond` for verdadeira. Caso `cond` sejá falsa, então $\mathtt{e}_2$ é executado. Expressão condicional é outro exemplo de construção que é muito específica. Além de estar presente apenas no regex de Perl, há diversas restrições para a condição, entre elas, permitir código embutido `(?{code})` e recursões explicitas `(?n)` que não são abordadas nesse trabalho. Em (Fri06), (GL09) e (WS00) temos ao todo seis exemplos que utilizam essa construção.