Pontifícia Universidade Católica
do Rio de Janeiro

Marcelo Oikawa

Conversão de regexes para Parsing Expression Grammars

PUC-Rio - Certificação Digital Nº 0812596/CA

Dissertação de Mestrado

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós–graduação em Informática do Departamento de Informática da PUC–Rio

Orientador: Prof. Roberto Ierusalimschy

Rio de Janeiro
Agosto de 2010

Pontifícia Universidade Católica
do Rio de Janeiro

Marcelo Oikawa

Conversão de regexes para Parsing Expression Grammars

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós–graduação em Informática do Departamento de Informática do Centro Técnico Científico da PUC–Rio. Aprovada pela Comissão Examinadora abaixo assinada.

Prof. Roberto Ierusalimschy
Orientador
Departamento de Informática — PUC–Rio

Prof. Luiz Henrique de Figueiredo
IMPA

Prof. Fabio Mascarenhas de Queiroz
Departamento de Informática — PUC-Rio

Prof. José Eugenio Leal
Coordenador Setorial do Centro Técnico Científico — PUC–Rio

Rio de Janeiro, 25 de Agosto de 2010

PUC-Rio - Certificação Digital Nº 0812596/CA

Todos os direitos reservados. É proibida a reprodução total ou parcial do trabalho sem autorização da universidade, do autor e do orientador.

Marcelo Oikawa

Graduou-se em Ciência da Computação pela Universidade Federal de Viçosa (Viçosa - Minas Gerais).

PUC-Rio - Certificação Digital Nº 0812596/CA

Ficha Catalográfica

Oikawa, Marcelo

Conversão de regexes para Parsing Expression Grammars / Marcelo Oikawa; orientador: Roberto Ierusalimschy. — 2010

v., 71 f: il. ; 29,7 cm

1. Dissertação (Mestrado em Informática) - Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2010

Inclui bibliografia

1. Informática – Teses. 2. Expressões regulares. 3; Regexes; 4. Gramáticas de expressões de parsing; 5. Reconhecimento de linguagens. I. Ierusalimschy, Roberto. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Agradecimentos

PUC-Rio - Certificação Digital Nº 0812596/CA

Ao meu orientador, Professor Roberto Ierusalimschy, por toda confiança, dedicação e paciência, além do imenso aprendizado que obtive durante o tempo em que trabalhamos juntos.

A minha grande amiga e 'mãe' Ana Lúcia de Moura por todas as conversas que me confortaram nas horas difíceis.

Aos amigos Thiago Valente e Vinícius Lopes por todo o apoio que me deram desde que cheguei ao Rio.

Aos grandes companheiros do LabLua, Sérgio Madeiros, Fábio Mascarenhas e Chico Sant'Anna, pelas várias horas de conversa e descontração.

A minha família pelo amor incondicional que sempre me fez ter forças para superar qualquer obstáculo.

Ao tratante do Lourival que prometeu uma garrafa de Whisky há um ano atrás e até hoje nada.

Meus sinceros agradecimentos a todos que um dia disseram 'Japa, você é bom e vai conseguir'.

Resumo

Oikawa, Marcelo; Ierusalimschy, Roberto. **Conversão de regexes para Parsing Expression Grammars**. Rio de Janeiro, 2010. 71p. Dissertação de Mestrado — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Expressões regulares são um formalismo utilizado para descrever linguagens regulares e compõem a base de diversas bibliotecas de casamento de padrão. No entanto, existem determinados padrões úteis que são complexos ou impossíveis de serem descritos com expressões regulares puras. Devido a essas limitações, linguagens de script modernas disponibilizam bibliotecas de casamento de padrões baseadas em *regexes*, isto é, extensões de expressões regulares compostas, principalmente, por construções ad-hoc que focam em problemas específicos. Apesar de serem muito úteis na prática, os regexes possuem implementações complexas e distantes do formalismo original de expressões regulares. Parsing Expression Grammars (PEG) são uma alternativa formal para reconhecer padrões e possuem mais expressividade que expressões regulares sem necessitar de contruções ad-hoc. O objetivo deste trabalho é estudar formas de conversão de regexes para PEGs. Para isso, estudamos as implementações atuais de regexes e mostramos a conversão de algumas construções para PEGs. Por fim, apresentamos uma implementação da conversão de regexes para PEGs para a linguagem Lua.

PUC-Rio - Certificação Digital Nº 0812596/CA

Palavras–chave

Expressões regulares; Regexes; Gramáticas de expressões de parsing; Reconhecimento de linguagens.

Abstract

Oikawa, Marcelo; Ierusalimschy, Roberto(Advisor). **Converting regexes to PEGs**. Rio de Janeiro, 2010. 71p. M.Sc Dissertation — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Regular expressions are a formalism used to describe regular languages and form the basis of several pattern-matching libraries. However, many interesting patterns either are difficult to describe or cannot be described by pure regular expressions. Because of these limitations, modern scripting languages have pattern matching libraries based on regexes, ie, extensions of regular expressions mainly composed by a set of ad-hoc constructions that focus on specific problems. Although very useful in practice, these implementations are complex and distant from the original formalism of regular expressions. Parsing Expression Grammars (PEG) are a formal alternative to recognize patterns and it is much more expressive than pure regular expressions and does not need use ad-hoc constructions. The goal of this work is to study the convertion of regexes to PEGs. To accomplish this task, we studied the current implementations of regexes and show how to convert some constructions to PEGs. Finally, we present an implementation that convert regexes to PEGs for the Lua language.

PUC-Rio - Certificação Digital Nº 0812596/CA

Keywords

Regular expressions; Regexes; Parsing Expression Grammars; Theory of Parsing.

Sumário

PUC-Rio - Certificação Digital Nº 0812596/CA

Lista de figuras

PUC-Rio - Certificação Digital Nº 0812596/CA

Lista de tabelas

PUC-Rio - Certificação Digital Nº 0812596/CA

PUC-Rio - Certificação Digital Nº 0812596/CA

Quando nasci, um anjo torto
desses que vivem na sombra
disse: Vai, Carlos! ser gauche na vida.

As casas espiam os homens
que correm atrás de mulheres.
A tarde talvez fosse azul,
não houvesse tantos desejos.

O bonde passa cheio de pernas:
pernas brancas pretas amarelas.
Para que tanta perna, meu Deus, pergunta
meu coração.
Porém meus olhos
não perguntam nada.

O homem atrás do bigode
é serio, simples e forte.
Quase não conversa.
Tem poucos, raros amigos
o homem atrás dos óculos e do bigode.

Meu Deus, por que me abandonaste
se sabias que eu não era Deus
se sabias que eu era fraco.

Mundo mundo vasto mundo,
se eu me chamasse Raimundo
seria uma rima, não seria uma solução.
Mundo mundo vasto mundo,
mais vasto é meu coração.

Eu não devia te dizer
mas essa lua
mas esse conhaque
botam a gente comovido como o diabo.

Carlos Drummond de Andrade

1
Introdução

Expressões regulares são um formalismo utilizado para descrever linguagens regulares (HMRU00). Atualmente, diversas linguagens de programação utilizam expressões regulares em suas implementações de casamento de padrão. Porém, expressões regulares puras são muito limitadas para criar alguns padrões práticos e, para contornar essas limitações, muitas implementações de casamento de padrão são baseadas em expressões regulares extendidas – chamados *regexes* – que consistem na combinação de expressões regulares com construções que focam em problemas específicos de casamento de padrão.

No final dos anos 90, os regexes de Perl (WS00) foram amplamente utilizados devido ao poder expressivo resultante do seu conjunto extenso de contruções (Fri06). Posteriormente, outras linguagens de script adotaram os regexes de Perl como base de suas implementações de casamento de padrão, por exemplo, Python (Lut06) e Ruby (TFH09), que possuem implementações compostas por um subconjunto dos regexes de Perl com diferenças sutis na semântica de algumas construções. Em Lua (Ier06), a biblioteca de casamento de padrão é baseada em expressões regulares simplificadas, de modo que só são permitidas alternativas e repetições de classes de caracteres.

Embora auxiliem a construção de padrões expressivos, os regexes têm implementações complexas e distantes do formalismo original de expressões regulares. Devido à ausência de uma base formal, é impossível prever a complexidade de determinados padrões, além de ser difícil inferir a semântica de algumas combinações entre as construções.

Ford (For04) apresentou um formalismo para reconhecimento de linguagens chamado Parsing Expression Grammars (PEG). Uma PEG possui uma sintaxe semelhante à de uma CFG na notação BNF com operações de expressões regulares; porém, ao invés de definir uma linguagem como as CFGs, uma PEG define um reconhecedor *top-down* para essa linguagem (For04).

A motivação para o uso de PEGs como alternativa para gramáticas livres de contexto se deve ao seu poder de expressão. Por exemplo, PEGs podem reconhecer todas as linguagens livres de contexto *LR(k)* determinísticas e algumas linguagens que não são livres de contexto (For04). Além disso,

PUC-Rio - Certificação Digital Nº 0812596/CA

PEGs possuem um modelo formal que descreve claramente a semântica de qualquer gramática. Recentemente, PEGs foram propostas como alternativa para casamento de padrões (Ier09), já que PEGs são mais expressivas que expressões regulares e possuem operadores similares aos de expressões regulares e regexes, como lookaheads e quantificadores possessivos.

A ausência de formalismo dos regexes e a expressividade de PEGs nos motivou a investigar correspondências entre regexes e PEGs. Um dos benefícios de converter regexes em PEGs é o melhor entendimento da semântica dos regexes, baseado no modelo formal de PEGs. Outro resultado interessante é a possibilidade de executar regexes sobre implementações de PEGs e oferecer uma base teórica para possíveis otimizações.

O objetivo deste trabalho é estudar a conversão de regexes para PEGs. Um dos resultados deste trabalho foi a criação de uma função baseada em continuação que converte expressões regulares puras em PEGs (OIdM10).

Para convertermos regexes em PEGs é necessário conhecer a semântica de cada construção regex. Para isso, apresentamos um estudo das construções regexes de Perl (WS00), PCRE (Haz09) (Perl Compatible Regular Expressions), Python (Fou10), Ruby (TFH09) e Lua (IdFC06), analisando a semântica das principais construções, como capturas, lookahead, repetição possessiva e expressões independentes. Este estudo também lista as construções que não conseguimos converter para PEGs, por exemplo, lookbehinds e backreferences, e aponta as características destas construções que dificultam a sua conversão para PEGs.

Esse trabalho também apresenta a implementação de um conversor de regexes para PEGs chamado *Lua Regex*. Para converter regexes em PEGs, o Lua Regex implementa a função baseada em continuação e suas extensões apresentadas neste trabalho. Esse conversor aceita como entrada expressões compostas por um subconjunto dos regexes de Perl e os regexes de Lua. A conversão resulta em uma gramática de LPeg (Ier09), que é uma implementação de PEGs para a linguagem Lua. A idéia básica desse conversor é permitir que desenvolvedores continuem utilizando a sintaxe de regexes, que é amplamente conhecida, deixando transparente que a sua execução seja sobre uma implementação de PEGs, no caso LPeg.

Este trabalho está organizado da seguinte forma: no capítulo 2 apresentamos um estudo detalhado dos regexes de Perl, PCRE, Python, Ruby e Lua baseado nas suas respectivas documentações. Para enriquecer esse estudo, fizemos diversos testes que visavam comprovar (ou confrontar) o que foi definido em cada documentação. O resultado desse estudo nos permite definir quais são as construções mais interessantes para a conversão e quais não serão

PUC-Rio - Certificação Digital Nº 0812596/CA

abordadas por serem pouco utilizadas na prática. No capítulo 3 descrevemos o formalismo de PEGs, suas vantagens em relação a expressões regulares e uma breve introdução a uma implementação de PEGs para a linguagem Lua, chamada LPeg (Ier08). No capítulo 4 introduzimos uma função baseada em continuação que converte expressões regulares em PEGs, e extensões dessa função que convertem um subconjunto de regexes. No capítulo 5 apresentamos o *Lua Regex*, uma implementação prática do continuation-based conversion para a linguagem Lua. Finalmente, no capítulo 6 discutimos as conclusões deste trabalho e resumimos suas contribuições.

PUC-Rio - Certificação Digital Nº 0812596/CA

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

PUC-Rio - Certificação Digital Nº 0812596/CA

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.

PUC-Rio - Certificação Digital Nº 0812596/CA

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

PUC-Rio - Certificação Digital Nº 0812596/CA

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-

PUC-Rio - Certificação Digital Nº 0812596/CA

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

PUC-Rio - Certificação Digital Nº 0812596/CA

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

PUC-Rio - Certificação Digital Nº 0812596/CA

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

PUC-Rio - Certificação Digital Nº 0812596/CA

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

PUC-Rio - Certificação Digital Nº 0812596/CA

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

PUC-Rio - Certificação Digital Nº 0812596/CA

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

PUC-Rio - Certificação Digital Nº 0812596/CA

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.

PUC-Rio - Certificação Digital Nº 0812596/CA

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
```

PUC-Rio - Certificação Digital Nº 0812596/CA

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

PUC-Rio - Certificação Digital Nº 0812596/CA

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.

PUC-Rio - Certificação Digital Nº 0812596/CA

3
Parsing Expression Grammars

Parsing Expression Grammars (PEG) são um formalismo que descreve reconhecedores de linguagens (For04). PEGs são uma alternativa para gramáticas livres de contexto (CFGs) e expressões regulares, e nos permite reconhecer diversas classes de linguagens. Por exemplo, PEGs reconhecem todas as linguagens livres de contexto *LR(k)* determinísticas e algumas linguagens sensíveis a contexto (For04).

A principal característica de PEGs é o *determinismo*. Isto é, dado uma PEG, esta casa apenas de uma forma com uma determinada entrada. O determinismo de PEGs é resultado de duas outras características fundamentais: falha e escolha ordenada. Dizemos que um padrão falha quando este não casa com um dado prefixo da entrada. Escolha ordenada é uma operação que define prioridade em uma sequência de alternativas através do operador barra (/). Em escolhas ordenadas, uma alternativa só será testada se a alternativa anterior tiver falhado.

Uma PEG é uma gramática G composta por uma 4-tupla (V_n, V_t, R, e_S) onde V_n é um conjunto finito de não-terminais; V_t é um conjunto finito de terminais sendo que $V_n \cap V_t = \emptyset$; R é um conjunto de regras $\texttt{A} \leftarrow \texttt{e}$ onde $\texttt{A} \in V_n$ e `e` é uma *expressão de parsing*; e e_S é a expressão de parsing usada como ponto de partida.

Os operadores que constroem expressões de parsing são listados na tabela abaixo.

Em PEGs, literais devem ser delimitados por aspas simples ou aspas duplas. Colchetes definem classes de caracteres semelhantes as classes de caracteres dos regexes. Estas podem conter intervalos como `[0-9]` e `[a-zA-Z]` ou uma lista de caracteres como `[abc123_]`. O operador ".", assim como nos regexes, casa com qualquer caracter.

A concatenação $\texttt{e}_1\,\texttt{e}_2$ tenta casar $\texttt{e}_2$ após o casamento de $\texttt{e}_1$; caso $\texttt{e}_1$ ou $\texttt{e}_2$ falhe, toda a concatenação falha.

A escolha ordenada define prioridade em uma lista de alternativas. Dessa forma, em um padrão $\texttt{e}_1/\texttt{e}_2$, o padrão $\texttt{e}_2$ só é testado se $\texttt{e}_1$ falhar. Se $\texttt{e}_1$ casar, $\texttt{e}_2$ nunca será testado.

PUC-Rio - Certificação Digital Nº 0812596/CA

Operadores	Descrição
`' '`	literal
`" "`	literal
`[...]`	classe de caracteres
`.`	qualquer caracter
`e?`	opcional
`e*`	zero-ou-mais
`e+`	um-ou-mais
`&e`	predicado de lookahead
`!e`	predicado de negação
$\texttt{e}_1\ \texttt{e}_2$	concatenação
$\texttt{e}_1\ /\ \texttt{e}_2$	escolha ordenada

Tabela 3.1: Operadores de PEGs

Uma consequência interessante da escolha ordenada é o backtracking restrito. Isso significa que PEGs permitem apenas backtrackings *locais*, isto é, uma PEG faz somente backtrackings para escolher uma opção de uma escolha ordenada. Assim que uma opção casar com a entrada, esta não pode ser desfeita mesmo que os padrões seguites falhem. Para ilustrar os backtrackings locais de PEGs, considere o exemplo abaixo:

$$\texttt{S} \leftarrow \texttt{A B}$$
$$\texttt{A} \leftarrow \texttt{p}_1\ /\ \texttt{p}_2\ /\ \ldots\ /\texttt{p}_n$$

Para casar `S` com a entrada, é necessário casar a expressão `AB`. Para casar `A`, tentamos casar a primeira alternativa $\texttt{p}_1$; se $\texttt{p}_1$ falhar, a PEG retrocede para a posição inicial da entrada (backtracking local) e tenta casar $\texttt{p}_2$ e assim por diante. Assim que um padrão $\texttt{p}_i$ casar, a PEG considera que `A` casou com um prefixo da entrada e tenta casar `B`. Caso `B` falhe, não haverá backtracking para `A`.

Os operadores `*`, `+` e `?` são repetições semelhantes às repetições de regexes, mas são cegas, isto é, em uma expressão $\texttt{e}_1\texttt{*}\ \texttt{e}_2$ o padrão $\texttt{e}_1$ repete o máximo de vezes possível (guloso) sem se importar se o restante da expressão, no caso $\texttt{e}_2$, irá casar (cego). O comportamento cego das repetições é uma consequência das escolhas ordenadas. Basicamente, a repetição `e*` é um açúcar sintático para a PEG abaixo:

$$\texttt{A} \leftarrow \texttt{e A / ""}$$

PUC-Rio - Certificação Digital Nº 0812596/CA

Para casar `A`, a PEG tenta casar o padrão `e` repetidas vezes e, assim que o casamento de uma expressão `e` falhar, a PEG tenta casar com a segunda opção `""`, que casa com o string vazia. Assim que a repetição do padrão `e` falhar, a segunda opção `""` irá casar com a entrada já que a string vazia casa com qualquer prefixo da entrada. Como os backtrackings de PEGs são locais, uma vez que o padrão `A` casa, os caracteres consumidos não estarão sujeitos a backtrackings, o que caracteriza uma repetição cega. De forma análoga, a repetição `+` também é cega por ser apenas um açúcar sintático para a expressão `e e*`.

A operação `?` é um açúcar sintático para a expressão `e/""` e, da mesma forma, é cega por dar prioridade ao casamento da expressão `e`.

PEGs possuem dois predicados sintáticos que não consomem porções da entrada. O predicado de negação, denotado por `!`, define que uma expressão `!e` casa somente se o padrão `e` falhar. Já o predicado de lookahead, denotado por `&`, define que uma expressão `&e` casa somente se `e` casar com a entrada. O predicado de lookahead `&e` é apenas um açúcar sintático para a PEG `!!e`. A PEG abaixo ilustra um exemplo prático que utiliza o predicado de negação para casar comentários de C:

```
C  ←  "/*" (!"*/".)* "*/"
```

Nesse exemplo, padrão `"/*"` casa com o início de um comentário. Logo após, a expressão `(!"*/".)*` consome todos os caracteres que **não** casam com o fim do comentário, isto é, caracteres que compõem o conteúdo do comentário. Assim que essa repetição falhar, a PEG tenta casar o padrão `"*/"` que delimita o fim do comentário.

Os predicados sintáticos são operadores que aumentam o poder de expressão das PEGs. Além de proporcionar um poderoso mecanismo de lookahead, PEGs não precisam de um operador específico para casar com o fim de entrada (denotado por `$` em regexes). O padrão `!.` casa apenas se . (que casa qualquer caracter) falhar; esta falha ocorre apenas no fim da entrada, quando não há mais caracteres a serem casados.

O predicado de negação também permite a construção de uma operação similar ao de complemento. Por exemplo, a PEG abaixo casa todas as palavras minúsculas excluindo as palavras reservadas (`"int"`, `"float"`, etc):

PUC-Rio - Certificação Digital Nº 0812596/CA

```
Identifier ← !Reserved [a-z]+
  Reserved ← ("int"/ "double"/ ... / "float") ![a-z]
```

Assim como BNF, uma PEG é um conjunto de uma ou mais regras. Abaixo temos a PEG completa que descreve sintaxe de PEGs:

```
     pattern ← grammar / simplepatt
     grammar ← (nonterminal "←" sp simplepatt)+
  simplepatt ← alternative ("/" sp alternative)*
 alternative ← ([!&]? sp suffix)+
      suffix ← primary ([*+?] sp)?
     primary ← "(" sp pattern ")" sp /
               "." sp /
               literal /
               charclass /
               nonterminal !"←"
     literal ← ['] (!['] .)* ['] sp
   charclass ← "["(!"]"(. "-". / .))* "]" sp
 nonterminal ← [a-zA-Z]+ sp
          sp ← [ \t\n]*
```

Uma PEG é composta por uma gramática (`grammar`) ou por um padrão simples (`simplepatt`). Gramáticas são compostas por um conjunto de regras na forma `nonterminal` ← `simplepatt`. Um padrão simples é uma sequência de alternativas separadas por barra (`/`) que correspônde a operação de escolha ordenada. Cada alternativa é uma sequência de `suffix` prefixados opcionalmente por predicados sintáticos e cada `suffix` é uma expressão seguida opcionamente por um quantificador. Por fim, uma expressão pode ser um literal demilitado por aspas simples ou aspas duplas, pode ser um ponto (`.`) que casa com qualquer caracter, uma referência para um não-terminal ou uma expressão agrupada por parênteses.

Abaixo, mostramos um exemplo simplificado de uma PEG que casa com elementos XML:

PUC-Rio - Certificação Digital Nº 0812596/CA

```
Elemento ← StartTag Value EndTag
StartTag ← "<"Name ">"
  EndTag ← "</" Name ">"
   Value ← Elemento+ / [a-zA-Z0-9_ ]+ / ""
    Name ← [a-zA-Z]+
```

Neste exemplo, um `Elemento` é composto pelo início de uma Tag (`StartTag`), um valor (`Value`) e pelo delimitador que indica o fim da Tag (`EndTag`). Tanto `StartTag` e `EndTag` possuem um nome que, basicamente, é composto por uma sequência de um ou mais caracteres alfabéticos. Cada valor pode ser uma sequência de elementos, ser um texto ou ser a string vazia (Tag sem valor).

PEGs casam em modo ancorado, isto é, PEGs não fazem busca do padrão na entrada. Porém, podemos contruir facilmente uma PEG que executa a busca de um dado padrão na entrada. Para ilustrar esse comportamento, considere a PEG abaixo, que reconhece expressões aritméticas em modo ancorado:

```
     Exp ← Factor (FactorOp Factor)*
  Factor ← Term (TermOp Term)*
    Term ← "-"? Number
FactorOp ← [+-]
  TermOp ← [*/]
  Number ← [0-9]+
```

Essa PEG reconhece a string `"2*3+4/4-1"`, porém falha ao casarmos com a string `"expressão 2*3+4/4-1"`. Porém, podemos reescrever esta PEG para buscar uma expressão aritmética na entrada. A PEG abaixo ilustra a gramática modificada:

```
S ← Exp / . S
```

`Exp` ← ... (igual à original)

Esta PEG tenta casar a expressão `Exp`; se essa alternativa falhar, a PEG consome um caracter da entrada e tenta casar a PEG inteira novamente. Dessa forma, a PEG consome os caracteres da entrada, um a um, buscando a primeira

PUC-Rio - Certificação Digital Nº 0812596/CA

ocorrência de uma expressão aritmética.

3.1 LPeg - PEGs em Lua

LPeg é uma implementação de PEGs focada em casamento de padrões (Ier09) e está disponível como um módulo Lua (Ier08). LPeg é uma biblioteca inspirada no casamento de padrão de SNOBOL4 (Gri71), onde os padrões são valores de primeira classe. Embora deixe a criação de padrões mais prolixa, essa abordagem possui diversas vantagens como, por exemplo, facilitar a inserção de comentários entre os padrões, armazenar padrões em variáveis auxiliares, reutilizar padrões e testar cada padrão independentemente.

A implementação de PEGs introduzida por Ford, chamada Packrat (For02), utiliza uma versão alterada do algoritmo que reconhece qualquer gramática *Generalized Top-Down Parsing Language* (GTDPL), um formalismo proposto por Aho & Ullman (AU72). Essa alteração consiste em usar *avaliação preguiçosa* presente em linguagens funcionais modernas como Haskell (Hut07) com o intuito de aumentar a eficiência do casamento. A implementação de LPeg, ao invés de adotar a estratégia de Packrat, utiliza uma *máquina virtual de parsing* onde cada PEG corresponde a um programa dessa máquina (MI08). A vantagem de usar essa abordagem é que a máquina de parsing possui um modelo de desempenho bem definido e uma implementação simples.

Nessa subseção vamos apresentar apenas as operações de LPeg fundamentais para o entendimento deste trabalho. Vamos descrever apenas algumas capturas que LPeg disponibiliza, são elas: capturas simples, capturas de posição e capturas constantes. A documentação completa de LPeg pode ser consultada online (Ier08).

Para explicar as capturas de LPeg, vamos utilizar a sintaxe de PEGs acrescida pelos metacaracteres {} que denotam início e fim de uma captura. Basicamente, uma captura simples consiste em obter o trecho que casou com uma determinada expressão. Utilizamos a sintaxe {`e`} para denotar a captura simples do padrão `e`.

Uma captura de posição consiste em obter a posição atual do casamento. Para denotar uma captura de posição, utilizamos a sintaxe {} (captura vazia).

Capturas constantes são capturas que produzem valores determinados. Para denotar essas capturas, utilizamos a sintaxe {$_c$"v_1", "v_2",..., "v_n"}, onde "v_1", "v_2" e "v_n" são os valores produzidos pela captura.

Como exemplo de captura simples, considere a PEG `"#"{[A-F0-9]+}` que casa com literais do sistema hexadecimal. O casamento dessa PEG com a string `"#0000FF"` resulta na captura do valor `"0000FF"`.

PUC-Rio - Certificação Digital Nº 0812596/CA

Para sabermos as posições de início e fim do valor hexadecimal, podemos utilizar capturas de posição. Se modificarmos o exemplo anterior para `"#"{}{[A-F0-9]+}{}` e casarmos com a mesma entrada, `"#0000FF"`, obtemos a lista de valores [`2, "0000FF", 8`], onde 2 é a posição capturada pela primeira captura de posição, {}, `"0000FF"` é o valor capturado pela expressão {`[A-F0-9]+`} e 8 é o valor capturado pela segunda captura de posição.

Para ilustrar um uso prático de capturas constantes, considere a PEG abaixo, que casa com expressão que possuem parênteses balanceados:

```
  Parenthesis  ←  "("(NonParenthesis / Parenthesis)* ")"
NonParenthesis ←  ![()] .
```

Podemos modificar a PEG acima de modo que, para cada parenteses `"("` casado, utilizamos uma captura constante que produz a string `"open"` seguida por uma captura de posição. De forma análoga, para cada parenteses `")"` criamos uma captura constante da string `"close"` seguida da captura de posição. Abaixo mostramos a PEG modificada para produzir essas capturas:

```
  Parenthesis  ←  Open (NonParenthesis / Parenthesis)* Close
         Open  ←  {c"open"} {} "("
        Close  ←  {c"close"} {} ")"
NonParenthesis ←  ![()] .
```

Ao casarmos essa PEG com uma expressão que possui parênteses balanceados teremos, como resultado, uma lista de valores onde cada `"open"` e `"close"` é seguido pela posição em que casaram na entrada. Por exemplo, o casamento desta PEG com a string `"(()())"` resulta na lista de valores [`"open", 1, "open", 2, "close", 3, "open", 4, "close", 5, "close", 6`].

PUC-Rio - Certificação Digital Nº 0812596/CA

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"`}.

PUC-Rio - Certificação Digital Nº 0812596/CA

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 Π constrói uma PEG que, ao final do processo de conversão, será equivalente a expressão regular `e`.

A função Π 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 ε 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:

PUC-Rio - Certificação Digital Nº 0812596/CA

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

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

PUC-Rio - Certificação Digital Nº 0812596/CA

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.

PUC-Rio - Certificação Digital Nº 0812596/CA

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-

PUC-Rio - Certificação Digital Nº 0812596/CA

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.

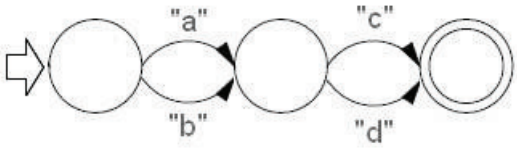

Figura 4.1: Backtracking global

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.

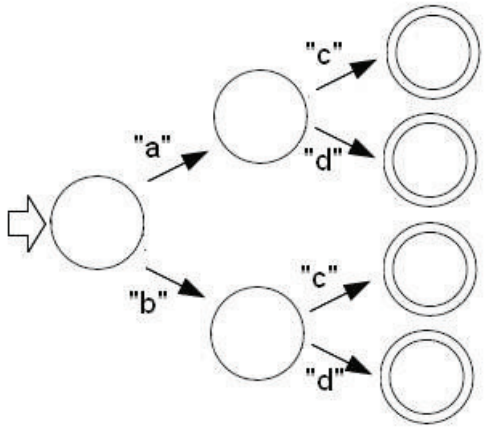

Figura 4.2: Backtracking local

4.3 Regexes → PEGs

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

PUC-Rio - Certificação Digital Nº 0812596/CA

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-

PUC-Rio - Certificação Digital Nº 0812596/CA

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.

PUC-Rio - Certificação Digital Nº 0812596/CA

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

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:

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

PUC-Rio - Certificação Digital Nº 0812596/CA

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

PUC-Rio - Certificação Digital Nº 0812596/CA

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

PUC-Rio - Certificação Digital Nº 0812596/CA

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:

PUC-Rio - Certificação Digital Nº 0812596/CA

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

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()`

PUC-Rio - Certificação Digital Nº 0812596/CA

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

PUC-Rio - Certificação Digital Nº 0812596/CA

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.

PUC-Rio - Certificação Digital Nº 0812596/CA

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

PUC-Rio - Certificação Digital Nº 0812596/CA

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`.

PUC-Rio - Certificação Digital Nº 0812596/CA

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.

PUC-Rio - Certificação Digital Nº 0812596/CA

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.

PUC-Rio - Certificação Digital Nº 0812596/CA

5
Lua Regex

Neste capítulo apresentamos a implementação de um conversor de regexes para PEGs, chamado *Lua Regex*. Dada uma expressão regex, este conversor utiliza a função Π e as extensões apresentadas neste trabalho para construir uma PEG equivalente. Esta biblioteca aceita como entrada expressões compostas por um subconjunto de regexes de Perl (WS00) e expressões regexes de Lua (IdFC06) e as converte em uma gramática de LPeg (Ier08), que é uma implementação de PEGs para Lua.

O Lua Regex é composto pelos módulos `regex` e `lregex`. Basicamente, a diferença entre esses dois módulos está na sintaxe das expressões que recebem como entrada. O módulo `regex` recebe como entrada regexes de Perl, enquanto que o módulo `lregex` recebe os regexes de Lua.

A principal contribuição do Lua Regex é permitir o casamento de regexes sobre um modelo formal proveniente de PEGs, ao invés de executar o casamento de forma ad-hoc como em outras bibliotecas de regexes.

5.1
Módulo regex

Dado uma expressão regex, este módulo reconhece esta expressão e constrói sua respectiva árvore sintática. De posse desta árvore, geramos uma PEG equivalente usando a função de conversão apresentada no capítulo 4.

A PEG abaixo descreve a sintaxe dos regexes de Perl aceitas pelo módulo `regex`:

PUC-Rio - Certificação Digital Nº 0812596/CA

```
                RE  ←  concat ( " | " concat )*
            concat  ←  expression+
        expression  ←  atom quantifier?
        quantifier  ←  ("*" / "+" / "?" / numeric_quantifier)("?" / "+" /"")
numeric_quantifier     "{" digit ("," digit /",")? "}"
             digit  ←  [0-9]+
              atom  ←  "(?" extensions RE ")" /
                       backreference  /
                       "(" RE ")" /
                       "." /
                       "$" /
                       "^" /
                       !metacharacter . /
                       set  /
                       escape  /
                       class
        extensions  ←  ":" /  "=" / "!" / "<=" /  "<!" /  ">" / "#"
     metacharacter  ←  "[" / "]" / [ {}()^$.|*+?\ ]
               set  ←  "[" "^"? setItem (!"]" setItem)* "]"
           setItem  ←  range  /  item
             range  ←  item "-" (!"]" item)
              item  ←  escape / metacharacter / [a-zA-Z0-9]
             class  ←  "\"[dwshDWSH]
            escape  ←  "\" .
     backreference  ←  "\" [0-9]+
```

Este módulo implementa os seguintes regexes: lookahead positivo, lookahead negativo, expressões independentes, quantificadores possessivos, quantificadores preguiçosos, capturas, âncoras de início de entrada e fim de entrada, agrupamento simples e comentários. Construções comuns de regexes também são aceitas, como classes de caracteres, o operador ".", quantificadores numéricos e escapes.

As construções sintáticas não implementadas nesta versão são: modificadores embutidos, capturas nomeadas, recursão explícita, expressão condi-

PUC-Rio - Certificação Digital Nº 0812596/CA

cional, expressão dinâmica, código embutido e branch reset. No capítulo 2, apresentamos brevemente estas construções e explicamos porque estas não foram abordadas neste trabalho. Por exemplo, construções como modificadores embutidos e recursão explícita possuem poucos exemplos práticos descritos na literatura e outras, como código embutido e expressão condicional, são dependentes de um interpretador Perl.

Este módulo reconhece sintaticamente as construções de lookbehind positivo, lookbehind negativo e backreferences, porém estas não estão implementadas. Nesta versão, se o usuário tentar converter expressões que possuem estas construções, o `regex` informa através de uma mensagem de erro que não oferece suporte a estas construções. Embora seja uma construção muito comum, na seção 4.5 apontamos as dificuldades encontradas na implementação de backreferences e seus impactos no desempenho de um casamento. Como a conversão de lookbehind ainda é um problema em aberto, não sabemos se é possível a conversão para PEGs e, por isso, não implementamos esta conversão nesta versão.

Este módulo possui uma API simples composta apenas pela função `match`. Esta função converte a expressão em uma PEG de LPeg e, logo em seguinda, executa o casamento sobre a entrada. Se o casamento falhar, retornamos o valor `nil`; se a expressão não possui capturas, o casamento retorna um valor numérico que indica a posição do próximo caracter após o trecho que casou.

Abaixo temos um exemplo simples de como utilizar esta função:

```
require"regex"

local subject = "programming language"

regex.match(subject, "[0-9]")    -- nil
regex.match(subject, "[a-z]*")   -- 12
regex.match(subject, "([a-z]*)") -- {"programming"}
```

A regra para identificação das capturas é a mesma de Perl, ou seja, capturas são identificadas estaticamente da esquerda para direita. O casamento de expressões com capturas resulta em uma tabela Lua que contém os valores capturados. Nesta tabela, cada captura está indexada pelo seu identificador. Por exemplo, o casamento da expressão `((a)(b))(c)` com a entrada `"abc"` resulta na tabela de capturas {`"ab"`, `"a"`, `"b"`, `"c"`}. Como a identificação das capturas é feita estaticamente, capturas em diferentes alternativas possuem identificadores diferentes. Por exemplo, na expresão `((a)|(b))(c)`, a

PUC-Rio - Certificação Digital Nº 0812596/CA

subexpressão `(a)` possui identificador 2 e a subexpressão `(b)` possui identificador 3. O casamento desta expressão com a entrada `"ac"` resulta na tabela de capturas {`"a", "a", nil, "c"`}, enquanto que, se casarmos esta mesma expressão com a entrada `"bc"`, obtemos a tabela {`"b", nil, "b", "c"`}.

5.2 Módulo lregex

O módulo `lregex` é o segundo módulo que compõe a biblioteca Lua Regex. Esse módulo converte os regexes de Lua para PEGs de LPeg.

A implementação deste módulo é análoga a do módulo `regex`. Dado uma expressão regex de Lua, construimos sua respectiva árvore sintática e, logo em seguida, convertemos esta árvore em uma PEG usando a função Π.

Em (IdFC06) temos a descrição das estruturas sintáticas dos regexes de Lua. A PEG abaixo reconhece esta sintaxe:

```
        pattern  ←  "^"? patternItem+ "$"?
    patternItem  ←  characterClass  [*+?-]? /
                    backreference  /
                    "%b" . ./
                    "()" /
                    "(" patternItem+ ")"
characterClass  ←  "."  /
                    class  /
                    escape /
                    set   /
                    !magicCharacter .
 backreference  ←  "%"[1-9]
         class  ←  "%"[acdlpsuwxzACDLPSUWXZ]
        escape  ←  "%"![a-zA-Z0-9] .
           set  ←  "[" "^"?  setItem (!"]" setItem)*  "]"
magicCharacter  ←  "[" / "]" / [-()%.*+?]
       setItem  ←  range / element
       element  ←  item / class
         range  ←  item "-" (!"]" item)
          item  ←  escape / magicCharacter / [a-z A-Z 0-9]
```

PUC-Rio - Certificação Digital Nº 0812596/CA

Assim como o `regex`, este módulo não implementa backreferences. Com exceção desta construção, todas as outras construções estão implementadas.

A API do `lregex` implementa duas funções da biblioteca `string` de Lua. São elas: `match` e `find`.

5.3 Análise de desempenho

Nesta seção, apresentamos uma análise do desempenho que compara o tempo de casamento de uma expressão convertida pelo Lua Regex com o tempo de casamento de outras implementações.

Nesta análise, comparamos o Lua Regex com os regexes de Lua e Perl. Escolhemos essas implementações pois o Lua Regex oferece suporte a esses regexes. Em nossa análise também comparamos o Lua Regex com a biblioteca LPeg. Para cada expressão que testamos, construímos uma PEG equivalente a expressão e que procura casar da forma mais otimizada possível.

Todos os testes desta seção utilizam a Bíblia[1] como entrada. O texto completo possui 4.432.995 caracteres distribuidos em 99.830 linhas (cada linha possui um verso). Todos os testes utilizam Lua Regex 0.1, Lua 5.1, Perl 5.10.0 e LPeg 0.9. Os testes foram executados em ambiente Linux, Ubuntu 9.10, em um AMD Athlon 64 Dual Core com 2 Gb de RAM.

Nosso primeiro teste consiste em medir o tempo necessário para buscar a primeira ocorrência de uma string na Bíblia. A tabela abaixo lista as strings que são buscadas e o tempo (em milisegundos) que cada implementação gastou para encontrá-las. Nesta tabela, também indicamos a linha em que cada string se encontra. A distância entre as strings é de aproximadamente 20000 linhas, sendo que, a primeira string está a 20000 linhas do início da entrada e a última está próxima do fim da entrada. Por enquanto, desconsidere a coluna "regex^2".

Expressão	Perl	Lua	LPeg	regex	regex^2	linha
`Geshurites`	1.4	1.0	3.8	41	3.5	19929
`worshippeth`	2.1	2.5	10.0	88	10.5	42113
`blotteth`	3.3	3.0	13.5	117	13.5	59998
`sprang`	4.2	8.5	29.5	163	29.5	79993
`Even so, come, Lord Jesus`	2.9	3.5	18.0	197	18.0	99811

Tabela 5.1: Tempo (em milisegundos) para buscar uma palavra na Bíblia

À primeira vista, podemos notar que, em todas as implementações, o tempo de busca das três primeiras strings, "Geshurites", "worshippet"e "blotteth", é proporcional à distância do início da entrada. Já a string

[1]Project Gutenberg: http://www.gutenberg.org/dirs/etext90/kjv10.txt

PUC-Rio - Certificação Digital Nº 0812596/CA

"sprang", apesar de sua ocorrência na entrada ser anterior a "Even so, come, Lord Jesus", demora mais tempo para ser buscada. Isso ocorre porque, na Bíblia, o número de ocorrências de prefixos da string "sprang"[2] é muito maior que o número de ocorrência dos prefixos de "Even so, come, Lord Jesus"[3], o que resulta em diversas tentativas inúteis antes do casamento correto da string buscada.

Perl e Lua otimizam o casamento de expressões que são strings simples (sem escapes, repetições, etc). A otimização de Perl consiste em utilizar uma tabela estática que contêm a frequência de cada caracter em textos em inglês. Assim, para cada string buscada, Perl obtém o caracter desta string que possui a frequência mais baixa, procura por esse caracter na entrada e analisa apenas esses locais (Sto). Com isso, Perl consegue obter na prática maior velocidade para buscar strings simples. Lua, ao detectar que a expressão é uma string, faz a busca usando a subrotina `memchr` de C [4], que é muito mais eficiente que um loop externo verificando cada caracter (HS91). Dessa forma, Lua consegue um melhor desempenho comparado a LPeg e muito próximo de Perl.

Na coluna "LPeg" mostramos o tempo que uma PEG otimizada necessita para buscar uma dada string. Para cada uma das string buscadas, criamos uma PEG que tenta casar a string apenas quando o primeiro caracter for correto (Ier09). Por exemplo, para buscarmos a string `Geshurites`, criamos a PEG `S ← Geshurites / .[^G]*S` que executa a recursão do não-terminal `S` apenas quando o primeiro caracter for `G`. Mesmo com a PEG criada dessa forma, a busca de uma string é de 3 a 4 vezes mais lenta que a implementação de Lua.

Neste teste, o módulo `regex` apresenta tempos muito superiores se comparados com as outras implementações. No módulo `regex`, a busca é feita através da PEG `S ← p / .S` onde `p` é o padrão equivalente a expressão regex. Como podemos ver, a busca implementada dessa forma é muito lenta pois, esta PEG tenta casar o padrão `p` a cada caracter que avança na entrada. Podemos otimizar esta busca avançando todos os caracteres diferentes do primeiro caracter da string e tentar casar o padrão `p` apenas quando o primeiro caracter for o correto, assim como fizemos nas PEGs usadas no teste de LPeg. Implementar esta otimização é fácil já que, antes de convertermos a expressão, podemos obter o primeiro caracter que casa com a string buscada e, com isso, implementar a busca otimizada `S ← p / .[^x]*S`, onde `x` é o primeiro caracter da string. Como podemos ver na coluna "regex[2]", o resultado desta

[2] A letra "s" aparece 147341 vezes
[3] A letra "E" aparece apenas 2600 vezes
[4] O código fonte da biblioteca `string` de Lua: http://www.lua.org/source/5.1/lstrlib.c.html

PUC-Rio - Certificação Digital Nº 0812596/CA

otimização resulta em uma busca muito mais rápida que a busca anterior e praticamente igual aos resultados de LPeg puro, como era esperado.

Nosso segundo teste utiliza expressões que começam com repetições de classes de caracteres. Cada expressão deste teste procura descobrir qual a string que antecede as palavras buscadas no teste anterior. Para isso, a expressão começa casando o padrão `[a-zA-Z]+` seguido por um espaço em branco e, por fim, concatenado com uma string.

Na tabela abaixo, apresentamos o resultado deste teste e, por enquanto, desconsidere novamente a coluna "regex^2".

Expressão	Perl	Lua	LPeg	regex	regex^2	linha
`[a-zA-Z]+ Geshurites`	68.7	180.5	92.0	145	92.5	19929
`[a-zA-Z]+ worshippeth`	145.4	392.0	196.0	309	204.5	42113
`[a-zA-Z]+ blotteth`	195.9	516.0	273.0	422	272.5	59998
`[a-zA-Z]+ sprang`	265.0	692.0	366.0	567	366.5	79993
`[a-zA-Z]+ Even so, come, Lord Jesus`	3.0	869.0	462.5	712	462.5	99811

Tabela 5.2: Tempo (em milisegundos) para casar expressões com repetições

PUC-Rio - Certificação Digital Nº 0812596/CA

Neste exemplo, a primeira expressão casa com o trecho "the Geshurites", a segunda expressão com "heaven worshippeth" , a terceira com "that blotteth", a quarta com "it sprang" e, por fim, a última expressão não casa com nada, pois a string "Even so, come, Lord Jesus" não é precedida por uma palavra.

Como podemos ver, Perl ainda é a implementação que executa o casamento mais rápido. Porém, a última expressão, que não casa com nenhum trecho, requer apenas 3 "mágicos" milisegundos, mesmo que esta string esteja próxima do fim da entrada.

Neste exemplo, o casamento de Lua requer de 2.5 a 3 vezes mais tempo que a implementação de Perl. Porém, mesmo sendo mais lenta, esta implementação apresenta mais regularidade em seus valores. Por exemplo, quanto mais próxima a string está do fim da entrada, mais tempo é gasto para acha-la (ao contrário de Perl).

As PEGs utilizadas para calcular o tempo de casamento de LPeg são praticamente iguais às expressões. Por exemplo, para a expressão `[a-zA-Z]+ Geshurites`, utilizamos a PEG `[a-zA-Z]+ " Geshurites"`.

Neste teste, o módulo `regex` executa as repetições de forma mais eficiente que a implementação de Lua. Mostramos, no capítulo 4, como converter as repetições de regexes através da inserção de um novo não-terminal na PEG gerada e da inserção de uma nova regra que é composta por escolhas ordenadas. É usando esta conversão que o módulo `regex` mantém o comportamento não-

cego de regexes. Porém, como podemos ver nesta tabela, as repetições de `regex`, mesmo sendo mais eficientes que os de Lua, ainda são muito lentas em comparação com LPeg puro. Entretanto, podemos utilizar a otimização que propomos na seção 4.3. Como a expressão `[a-zA-Z]+` é seguida por um caracter que não pertence ao padrão repetido (o caracter de espaço), podemos criar uma PEG igual as utilizadas no teste de LPeg puro, ou seja, utilizando repetições cegas ao invés de chamadas de não-terminais. Implementar esta otimização é simples pois, em tempo de conversão da expressão, é fácil verificarmos se o caracter que sucede uma repetição casa com o padrão repetido. Se não casar, podemos criar PEGs que executam repetições cegas e, com isso, casar de forma mais eficiente. Como podemos ver na coluna "regex^2", o tempo de casamento desta implementação otimizada é na ordem de 1.5 vez mais veloz que a repetição baseada em chamadas de não-terminais.

O módulo `lregex` possui o desempenho semelhante ao módulo `regex`. Na tabela abaixo, apresentamos o tempo gasto pelo módulo `lregex` nos testes apresentados anteriormente. Nesta tabela, a primeira coluna apenas repete os valores de Lua para facilitar a comparação com o `lregex`.

PUC-Rio - Certificação Digital Nº 0812596/CA

Expressão	Lua	lregex	lregex[2]
`Geshurites`	1.0	51	4
`worshippeth`	2.5	87	9
`blotteth`	3.0	116	13
`sprang`	8.5	165	29
`Even so, come, Lord Jesus`	3.5	199	18
`[a-zA-Z]+ Geshurites`	180.5	158	111
`[a-zA-Z]+ worshippeth`	392.0	332	229
`[a-zA-Z]+ blotteth`	516.0	449	301
`[a-zA-Z]+ sprang`	692.0	606	409
`[a-zA-Z]+ Even so, come, Lord Jesus`	869.0	762	512

Tabela 5.3: Testes de busca e repetições de `lregex`

Note que o `lregex` possui o desempenho sutilmente pior se compararmos com o módulo `regex`. Isso ocorre porque, enquanto a função `match` do módulo `regex` retorna apenas a posição seguinte de onde casou, a função `find` do módulo `lregex` retorna a posição em que começou o casamento e a posição em que terminou o casamento. Para obter estas posições, a implementação de `lregex` utiliza duas capturas de posição, uma no início da PEG gerada e outra no final. Essa pequena diferença das tarefas de cada função resulta no casamento de `find` ser ligeiramente mais custoso que o casamento da função `match`.

Nosso próximo teste analisa o tempo gasto para procurar nomes que possuem diferentes grafias. Esse teste é uma adaptação do exemplo que mencionamos na seção 4.1 que, por sua vez, procura casar formas diferentes de escrita do nome `Jeffrey` (Fri06). Neste teste, utilizamos 4 nomes de personagens presentes na Biblia: `Jaakobah`, `Jehonathan`, `Bartholomew` e `Timotheus`.

No caso de `Jaakobah`, este pode ser escrito [5] com um ou dois 'a's no início, `Jaakobah` ou `Jakobah`; pode ser escrito com 'c' ao invés de 'k', `Jaakobah` ou `Jaacobah`; e pode terminar ou não com 'h', `Jaakobah` ou `Jaakoba`. Ao fazermos todas essas combinações, temos 8 formas diferentes de escrita do nome `Jaakobah`. O nome `Jehonathan` pode ser escrito sem o 'eh' e pode ter o último 'h' opcional. `Bartholomew` pode ser escrito sem o 'h' e terminar com 'w' ou 'u'. Por fim, `Timotheus` pode ser escrito como 'he' ou 'i' e terminar com 'us' ou 'o'.

Na tabela abaixo, mostramos os resultados do nosso teste ordenados pela linha em que o padrão se encontra na Biblia.

Perl é mais rápido que as implementações de LPeg e do módulo regex, porém, um dos seus resultados destoa dos demais. No caso, Perl demora

[5] Essas variações foram obtidas a partir de buscas simples no Google e na Wikipedia.

PUC-Rio - Certificação Digital Nº 0812596/CA

Expressão	Perl	LPeg	regex	linha
`Jaa?(k|c)obah?`	3.7	6.5	6.8	35074
`J(eh)?onath?an`	2.0	7.0	7.2	37116
`Barth?olome(w|u)`	5.6	14.6	14.6	77404
`Timot(he|i)(us|o)`	6.2	16.4	16.7	89196

Tabela 5.4: Expressões com sequências de alternativas (em milisegundos)

em torno de 3.7 milisegundos para achar a string `"Jaakobah"` e demora aproximadamente metade desse tempo para achar a string `"Jehonathan"` que está 2000 linhas a frente de `"Jaakobah"`. Nas medidas de LPeg e do módulo regex, os valores são regulares, no sentido de que, quanto mais distantes do início da entrada, maior o tempo para achar a string.

Em LPeg, construimos uma PEG semelhante a expressão, trocando apenas o operador de alternativa pelo operador de escolha ordenada. Para essas expressões, em particular, podemos construir a PEG dessa forma pois cada alternativa casa com prefixos diferentes, o que nos garante que sempre que uma alternativa casar, necessariamente a alternativa não casaria. Por exemplo, para a expressão `Jaa?(k|c)obah?` utilizamos a PEG `Jaa?(k/c)obah?`. Com a PEG construida dessa forma, o desempenho do casamento é em torno de 2 a 3 vezes mais lento que Perl.

No módulo regex, a PEG gerada para cada expressão é maior que a expressão original e da PEG utilizada no teste de LPeg. Por exemplo, a conversão da expressão `Ja(?:a|)(?:k|c)oba(?:h|)`, que possui 24 caracteres, resulta na PEG abaixo, composta por 68 caracteres:

```
Ja(a(koba(h/"")/coba(h/""))/(koba(h/"")/coba(h/"")))
```

O proximo teste consiste em procurar duas palavras específicas na mesma frase. Consideramos que duas palavras estão na mesma frase apenas se estiverem na mesma linha, separadas por zero ou mais palavras, virgulas ou espaços. Para isso, usamos expressões no seguinte formato:

```
Word1[a-zA-Z, ]*Word2 | Word2[a-zA-Z, ]*Word1
```

Por exemplo, para acharmos as palavras `"Adam"` e `"Eve"` em uma mesma frase, basta usarmos a expressão `Adam[a-zA-Z, ]*Eve|Eve[a-zA-Z, ]*Adam`. Ao casarmos essa expressão com a Biblia, obtemos a string `"Adam knew Eve"` na linha 254. Abaixo, mostramos os resultados ordenados pela linha em que este padrão casa, sendo que, o último padrão desta tabela não casa por não existir uma frase que contém ambas as palavras `"Abraham"` e `"Jesus"`.

PUC-Rio - Certificação Digital Nº 0812596/CA

Expressão	Perl	LPeg	regex	linha
`Adam - Eve`	0.1	0.2	1.0	254
`Samaria - Israel`	12.1	8.8	77.0	31141
`John - Jesus`	16.2	14.8	180.0	76785
`Judas - Jesus`	16.6	17.4	199.0	84619
`Jude - Jesus`	20.1	21.2	231.0	98317
`Abraham - Jesus`	21.0	25.4	236.0	não existe

Tabela 5.5: Expressões que buscam duas palavras na mesma frase (em milisegundos)

Neste teste, o desempenho de LPeg é similar ao de Perl. Dado duas palavras, por exemplo `"Adam"` e `"Eve"`, construimos uma PEG da seguinte forma:

```
Search ← S / .(![AE] .)* Search
     S ← A / C
     A ← "Adam" B
     B ← [a-zA-Z, ] B / "Eve"
     C ← "Eve" D
     D ← [a-zA-Z, ] D / "Adam"
```

Em nosso módulo regex, temos resultados dez vezes mais lentos que o do módulo LPeg. O módulo regex constroi uma PEG igual ao que utilizamos no teste de LPeg. Porém, como a expressão é basicamente uma alternativa, o módulo regex não consegue fazer a otimização de busca que mencionamos neste seção. Como vimos no primeiro teste dessa seção, a busca do padrão sobre uma entrada grande domina o tempo de casamento, o que explica os resultados do módulo regex serem mais lentos que LPeg puro.

Por fim, nosso último teste consiste em procurar frases que contém duas palavras especificas. A diferença deste teste em relação ao teste anterior está na porção de texto que é casado. No teste anterior procuramos, por exemplo, `"Adam"` e `"Eve"` na mesma frase e obtemos o trecho `"Adam knew Eve"`. Neste teste, estamos procurando a frase inteira que contém `"Adam"` e `"Eve"`, no caso, `"And Adam knew Eve his wife"`. Para isso, usamos expressões semelhantes ao do teste anterior, apenas adicionando repetições no início e no fim de cada alternativa:

PUC-Rio - Certificação Digital Nº 0812596/CA

```
[a-zA-Z, ]*Word1[a-zA-Z, ]*Word2[a-zA-Z, ]*|
[a-zA-Z, ]*Word2[a-zA-Z, ]*Word1[a-zA-Z, ]*
```

Abaixo, mostramos os resultados, ordenados pela linha em que o padrão casa. Note que, neste teste, medimos o tempo de casamento em segundos ao invés de milisegundos como fizemos nos testes anteriores.

Expressão	Perl	LPeg	regex	linha
`Adam - Eve`	0.03	0.05	0.04	254
`Samaria - Israel`	1.70	4.52	4.53	31141
`John - Jesus`	4.10	10.12	10.32	76785
`Judas - Jesus`	4.41	11.28	11.29	84619
`Jude - Jesus`	5.24	13.20	13.18	98317
`Abraham - Jesus`	5.31	13.45	13.52	não existe

Tabela 5.6: Expressões que buscam frases com duas palavras especificas (em segundos)

PUC-Rio - Certificação Digital Nº 0812596/CA

Perl é a implementação que apresenta maior desempenho em relação ao casamento feito por LPeg e pelo módulo regex.

Os testes de LPeg apresentam valores na ordem de 1.5 a 2.5 vezes mais lentos que Perl. Para cada teste, construimos uma PEG com a seguinte estrutura.

```
S  ←  A / C
A  ←  [a-zA-Z, ] A / "Word1" B
B  ←  [a-zA-Z, ] B / "Word2" [a-zA-Z, ]*
C  ←  [a-zA-Z, ] C / "Word2" D
D  ←  [a-zA-Z, ] D / "Word1" [a-zA-Z, ]*
```

Essa expressão regex, em particular, é potencialmente custosa, já que é composta por seis repetições gulosas e duas alternativas. Padrões com muitas repetições não são muito comuns na prática. Por exemplo, no livro *Mastering Regular Expressions* (Fri06), a expressão que possui mais repetições é composta por quatro repetições, no caso, `[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+`, que é utilizada para casar endereços de ip.

O módulo regex apresenta resultados praticamente iguais aos do teste de LPeg puro. Estes valores eram esperados, pois a nossa conversão, basicamente, constroi uma PEG idêntica as que utilizamos no teste de LPeg. Porém, não é

possível aplicar as duas otimizações que propomos anteriormente nesta seção. Como a expressão começa com uma repetição ao invés de uma letra fixa, a nossa otimização de busca não é feita durante a conversão. Dessa forma, é necessário tentarmos o casamento da expressão inteira para cada caracter da entrada, o que resulta em uma queda do desempenho do casamento. Além disso, cada repetição `[a-zA-Z, ]*` é seguida por uma palavra (`"Jesus"` ou `"Adam"`). Dessa forma, a otimização de repetições, que consiste em convertermos as possíveis repetições de regex em repetições cegas de PEGs, também não é feita já que essas palavras podem ser consumidas pela repetição se ela for executada de forma cega. Assim, cada repetição da expressão é convertida para uma chamada de não-terminal, como mostramos no caso 4-5, a fim de manter o comportamento não-cego das repetições de regexes. Porém, essa conversão, como vimos no nosso segundo teste de desempenho, potencializa a ineficiência do casamento.

PUC-Rio - Certificação Digital Nº 0812596/CA

6
Conclusão

Bibliotecas de casamento de padrão são ferramentas presentes em diversas linguagens de programação. Muitas destas bibliotecas, baseadas em regexes, permitem a criação de padrões poderosos, graças ao seu conjunto expressivo de contruções. Porém, um ponto fraco dos regexes é a ausência de um modelo formal. Sem um modelo formal é difícil inferir a semântica de algumas construções e, consequentemente, o comportamento de algumas combinações.

Neste trabalho, apresentamos um estudo da conversão de regexes em PEGs. Como resultado deste estudo, criamos uma função que converte expressões regulares em PEGs baseada em continuação. Também criamos extensões desta função que permitem a conversão de algumas extensões de expressões regulares. Entretanto, não conseguimos propor conversões para algumas extensões de expressões regulares mais comuns, como backreferences, lookbehinds e âncoras de início. Ao analisarmos essas construções, percebemos que estas possuem uma característica em comum, que é depender de trechos que antecedem a posição atual do casamento. Esta característica explica a dificuldade em convertê-las para PEGs, já que as PEGs consideram que apenas o restante da entrada pode afetar o casamento.

Outro resultado deste trabalho foi mostrar que é possível executar regexes sobre implementações de PEGs e, com isso, oferecer uma base teórica para possíveis otimizações. Este resultado mostra que PEGs, através de um conjunto muito menor de construções, se comparado com regexes, proporcionam poder de expressão equivalente às construções mais comuns de regexes.

Na implementação do Lua Regex, fizemos uso da teoria produzida neste trabalho para construir uma biblioteca regex baseada em PEGs. Além disso, fizemos uma análise de desempenho que compara a nossa implementação com Perl, Lua e LPeg. Nesta análise não tivemos a intenção de mostrar que "*a implementação A é mais rápida que B*", e sim, procuramos mostrar como o modelo formal de PEGs nos permite checar possíveis otimizações que podem ser feitas durante a conversão para PEGs. Por exemplo, há casos em que as repetições de regexes podem ser convertidas para repetiçoes cegas de PEGs ao invés de utilizarmos recursões na gramática e, dessa forma, conseguimos casar

PUC-Rio - Certificação Digital Nº 0812596/CA

essas repetições de forma mais eficiente.

Uma linha de trabalho futuro consiste em tentar solucionar o problema da conversão de lookbehinds em PEGs. Na seção 4.6, propusemos um novo predicado sintático para PEGs, *back*, que consiste em retroceder uma posição do casamento e tentar casar uma dada expressão. Apesar de sua simplicidade aparente, a inserção deste predicado implica em mudanças no modelo conceitual de PEGs que, originalmente, assume que somente o restante da entrada pode afetar o resultado do casamento. Com a possibilidade de retroceder uma (ou mais) posições durante o casamento, é possível que este predicado proporcione mais poder de expressão às PEGs. Em contrapartida, novos problemas podem surgir, como loops infinitos. Assim, um estudo minucioso ainda deve ser feito em torno deste novo predicado de PEGs.

PUC-Rio - Certificação Digital Nº 0812596/CA

Referências Bibliográficas

[Aho90] Alfred V. Aho. Algorithms for finding patterns in strings. pages 255–300, 1990.

[AU72] Alfred V. Aho and Jeffrey D. Ullman. *The theory of parsing, translation, and compiling*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1972.

[Cox07] Russ Cox. Regular Expression Matching Can Be Simple And Fast, 2007. `http://swtch.com/~rsc/regexp/regexp1.html`, último acesso em 30 de Maio, 2010.

[For02] Bryan Ford. Packrat parsing:: simple, powerful, lazy, linear time, functional pearl. In *ICFP '02: Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, pages 36–47, New York, NY, USA, 2002. ACM.

[For04] Bryan Ford. Parsing Expression Grammars: a recognition-based syntactic foundation. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 111–122, New York, NY, USA, 2004. ACM.

[Fou10] The Python Software Foundation. Regular Expression operations - Python v2.6.4 documentation, 2010. `http://docs.python.org/library/re.html`, último acesso em 20 de Janeiro, 2010.

[Fri06] Jeffrey Friedl. *Mastering Regular Expressions*. O'Reilly Media, Inc., 2006.

[GL09] Jan Goyvaerts and Steven Levithan. *Regular Expressions Cookbook*. O'Reilly Media, Inc., 2009.

[Gri71] Ralph E Griswold. *The SNOBOL 4 Programming Language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1971.

[Haz09] Philip Hazel. PCRE - Perl Compatible Regular Expressions, 2009. `http://www.pcre.org/`, último acesso em 14 Jan, 2010.

PUC-Rio - Certificação Digital Nº 0812596/CA

[HMRU00] John E. Hopcroft, Rajeev Motwani, Rotwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computability*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[HS91] Andrew Hume and Daniel Sunday. Fast string searching. *Softw. Pract. Exper.*, 21(11):1221–1248, 1991.

[Hut07] Graham Hutton. *Programming in Haskell*. Cambridge University Press, New York, NY, USA, 2007.

[IdFC06] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. *Lua 5.1 Reference Manual*. Lua.Org, 2006.

[Ier06] Roberto Ierusalimschy. *Programming in Lua, Second Edition*. Lua.Org, 2006.

[Ier08] Roberto Ierusalimschy. Parsing Expression Grammars for Lua, 2008. `http://www.inf.puc-rio.br/~roberto/lpeg/`, último acesso em 3 de Novembro, 2009.

[Ier09] Roberto. Ierusalimschy. A text pattern-matching tool based on Parsing Expression Grammars. *Softw. Pract. Exper.*, 39(3):221–258, 2009.

[IG04] The IEEE and The Open Group. The Open Group Base Specifications Issue 6, 2004. `http://www.opengroup.org/onlinepubs/009695399/basedefs/xbd_chap09.html`, último acesso em 30 Outubro, 2009.

[Lut06] Mark Lutz. *Programming Python*. O'Reilly Media, Inc., 2006.

[Med10] Sérgio Medeiros. *Correspondência entre PEGs e Classes de Gramáticas Livres de Contexto*. PhD thesis, PUC-Rio, 2010.

[MI08] Sérgio Medeiros and Roberto Ierusalimschy. A parsing machine for PEGs. In *DLS '08: Proceedings of the 2008 symposium on Dynamic languages*, pages 1–12, New York, NY, USA, 2008. ACM.

[OIdM10] Marcelo Oikawa, Roberto Ierusalimschy, and Ana Lucia de Moura. Converting regexes to Parsing Expression Grammars. In *XIV Brazilian Symposium on Programming Languages*, 2010.

[Sto] Robert Stockton. Search and Modification Operations. `http://www.cs.cmu.edu/afs/cs.cmu.edu/Web/People/rgs/pl-exp-regex.html`, último acesso em 9 de Agosto, 2010.

PUC-Rio - Certificação Digital Nº 0812596/CA

[TFH09] Dave Thomas, Chad Fowler, and Andy Hunt. *Programming Ruby 1.9: The Pragmatic Programmers' Guide*. Pragmatic Bookshelf, 2009.

[Tru] Iain Truskett. Perl - Regular Expressions Reference. `http://perldoc.perl.org/perlreref.html`, último acesso em 10 de Janeiro, 2010.

[War02] Paul Warren. regexp-based address validation, 2002. `http://www.ex-parrot.com/~pdw/Mail-RFC822-Address.html`, último acesso em 14 de Julho, 2010.

[WS00] Larry Wall and Randal L. Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000.

PUC-Rio - Certificação Digital Nº 0812596/CA