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