EasyRegex: Uma engine simplificada de expressões regulares

Relatório parcial
Linguagens Formais e Automatos

Período acadêmico 2013.2
Centro de Informática
Universidade Federal da Paraíba

Fernando S. de M. Brito (11111309)
David Leite Guilherme (11218646)
Jailson Junior Cunha de Souza Lima (11211409)

24 de março de 2014

Sumário

Lista de Figuras

1. Introdução

1.1. Motivação

Expressões regulares consistem em constantes e símbolos de operadores que denotam conjuntos de *strings* e operações sobre esses conjuntos [Sipser, 2012]. Em outras palavras, é um método formal para especificar padrões de texto. As expressões regulares são largamente empregadas em linguagens de programação, validadores de entradas de usuários em softwares e sistemas, analisadores léxicos, sistemas de busca e muito mais. Podemos citar como exemplos o analisador e validador para nomes de usuários do *Facebook*.

Existe bibliotecas para a utilização de expressões regulares implementadas em praticamente todas as linguagens de programação mais populares. Porém, estas bibliotecas tem como foco o desempenho, obviamente.

1.2. Objetivos

O *EasyRegex* se trata de uma biblioteca que suporta operações de busca de padrões de *strings* (descritas na forma de expressões regulares) em outras strings. O principal diferencial é que o mesmo possui um código fonte de fácil entendimento e bem documentado, se comparado com implementações de alto desempenho comumente usadas. Com isso, não só tivemos a chance de aprender bem como funciona uma *engine* de expressões regulares, mas também oferecemos a oportunidade para que outras pessoas estudem o nosso código-fonte.

Como artefatos produzidos, temos a biblioteca em si (código-fonte e binários), além de sua documentação interna, de um manual de uso e deste relatório. O presente texto documenta o processo de desenvolvimento e das dificuldades encontradas durante a implementação. Por último, uma curta apresentação explicando como utilizar o sistema e como navegar pela documentação será realizada para os alunos da disciplina.

2. Implementação

2.1. Tecnologias

Antes de começar a implementação, decidimos que iríamos usar uma linguagem de programação dinâmica, de alto nível e fracamente tipada, pois o nosso foco era produzir, em tempo hábil, um código-fonte que fosse o mais legível possível, sem se preocupar tanto com performance. Ficamos em dúvida entre *Ruby* e *Python*, por serem linguagem extremamente populares e com bastante documentação disponível. Devido à familiaridade prévia de alguns membros da equipe, optamos por *Ruby*[1].

Como dependências, o nosso projeto possui as seguintes bibliotecas (denominada *gems*, em *Ruby*):

- `Bundler`: gerenciador de dependências.
- `RSpec`: framework de testes. Usado na implementação de testes unitários no nosso projeto.

Utilizamos o *git*[2] para versionar o nosso código fonte e a plataforma colaborativa *GitHub*[3] como repositório central.

[1] Ruby programming language: http://www.ruby-lang.org/
[2] Git: http://www.git-scm.com
[3] GitHub: http://www.github.com

2.2. Algoritmo

Tomamos como base do nosso trabalho o artigo de Cox [2007], que é uma explicação bastante detalhada e didática do algoritmo clássico de Thompson [1968]. O algoritmo parte do fato de que é possível transformar uma expressão regular em um AFND (Automato Finito Não-Determinístico) [McNaughton and Yamada, 1960].

Resumidamente, o algoritmo (com as modificações de Cox [2007]) segue os seguintes passos:

1. `Conversão`: Converte a expressão regular de entrada para a forma pós-fixa;
2. `Compilação`: Cria um AFND a partir da forma pós-fixa;
3. `Execução`: Simula o AFND na *string* de entrada.

2.3. Arquitetura

No artigo de Cox [2007] é apresentado um código em C. Parte do nosso trabalho foi adaptar este código, que se encontra em uma linguagem estruturada, para um programa orientado a objetos.

A Figura 1 descreve as classes e pacotes utilizados na nossa biblioteca.

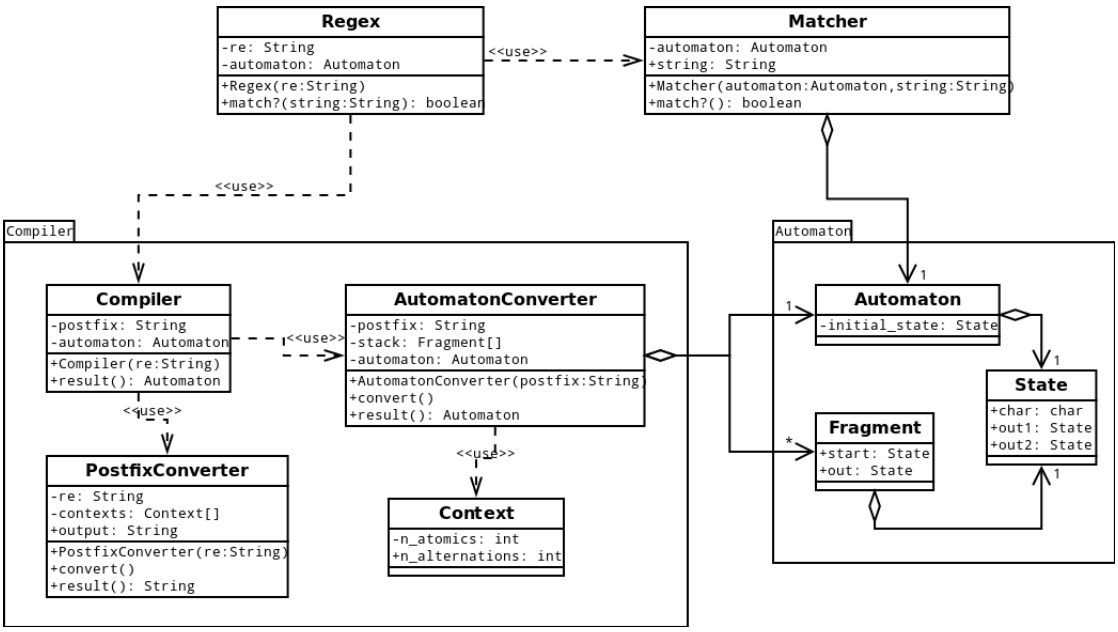

Figura 1: Arquitetura do sistema em UML.

Do ponto de vista do usuário, apenas a classe `Regex` precisa ser utilizada. Seu construtor aceita como parâmetro a expressão regular, e imediatamente o processo de *conversão*

e *compilação* é executado, guardando no objeto o AFND pronto. Em seguida, através do método `match?` é possível passar uma *string* para que seja verificado se a expressão regular *casa* com a string ou não.

3. Resultados

Encontramos algumas dificuldades durante a implementação devido a falta do uso explícito de ponteiros. Inicialmente ao implementar a ligação entre os fragmentos de estados do AFND. Enfrentamos também dificuldades ao implementar os testes automatizados. Pela dificuldade de se representar as possíveis saídas para o processo de *compilação*, no qual não conseguiamos representar o objeto AFND esperado. Na tentativa de representar o objeto AFND, tentamos imprimir o automato em formato de *string* porém entrava em loop infinito.

3.1. Operadores suportados

Foram implementados todos operadores básicos menos o `.`(qualquer caractere). São eles:

1. `ae`: Concatenação implícito, depois convertido em `"."` para o processo de *compilação*;
2. `a|b`: Disjunção;
3. `a?`: Uma ou nenhuma ocorrência;
4. `a*`: Nenhuma ou mais ocorrências;
5. `a+`: Uma ou mais ocorrências.

O operador de *qualquer caractere* representado não foi implementado pois o caractere ponto representa, na implementação, o operador de concatenação.

4. Conclusão

4.1. Limitações e trabalhos futuros

Devido às restrições de tempo, o projeto teve um escopo reduzido e apenas algumas funcionalidades essenciais foram implementadas.

Faltaram a implementação de *escape* de operadores e do operador de *qualquer caractere*. Outra grande limitação é a falta de um método para retornar todas as ocorrências de um determinado *match*.

Com todas as operações mais primitivas já implementadas, a implementação de outras funcionalidades pode ser facilmente desenvolvidas. Poderiamos implementar, por exemplo, alguns *açúcares sintático* como o *[a-Z]* que aceita qualquer caractere de a até Z.

Apesar de todas as limitações, acreditamos ter antigido o principal objetivo do trabalho. Durante o desenvolvimento, tivemos vários *insights* sobre a simulação de AFNDs e de detalhes importantes para o funcionamento de uma *engine* de expressões regulares. Ao disponibilizarmos o código fonte na internet, pretendemos ajudar outras pessoas a aprenderem o mesmo que aprendemos.

Referências

Russ Cox. Regular expression matching can be simple and fast. http://swtch.com/~rsc/regexp/regexp1.html, 2007. Acessado em: 12 de março de 2014.

R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *Electronic Computers, IRE Transactions on*, EC-9(1):39–47, March 1960. ISSN 0367-9950. doi: 10.1109/TEC.1960.5221603.

Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 3 edition, 6 2012. ISBN 9781133187790. URL http://amazon.com/o/ASIN/113318779X/.

Ken Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

Apêndices

A. Manual de instalação

Esse manual presume que você possui uma versão recente do *Ruby* (pelo menos 2.0.0) instalada em seu computador e que você está em um ambiente *Unix*.

A biblioteca *EasyRegex* está hospedada de forma pública no *GitHub* e a página do projeto pode ser acessada em https://github.com/fernandobrito/easy_regex.

O *download* do código fonte em sua versão mais recente está disponível em https://github.com/fernandobrito/easy_regex/archive/master.zip.

Após descomprimir o arquivo obtido no endereço anterior e entrar na pasta do projeto, é necessário instalar as dependências. Isso deve ser feito através do gerenciador de dependências *Bundler*, com o seguinte comando:

```
$ bundle
```

Caso ocorra algum erro alegando que o comando não existe, faz-se necessário instalar o *Bundler*:

```
$ gem install bundler
```

Pronto, a instalação está concluida. Opcionalmente o usuário pode gerar o pacote `.gem` através do comando `gem build easy_regex.gemspec` e instalá-lo no sistema com `gem install <arquivo gerado>`, mas para testar o funcionamento da biblioteca também é possível acessá-la diretamente da pasta onde rodamos os comandos anterior.

B. Manual de uso

O jeito mais simples de ter acesso à biblioteca é através do `console` interativo do *Ruby*. De dentro da pasta da biblioteca, execute o comando `irb`. Para testar o *console*, digite `1+1` e pressione `Enter`. Caso apareça `2` na tela, isso significa que o ambiente está funcionando corretamente.

Em seguinda, inclua a biblioteca através da seguinte operação:

```
require "./lib/easy_regex"
```

Se esta importação ocorrer com sucesso, o comando retornará `true`.

A principal classe do sistema é a `EasyRegex::Regex`. Para testar a expressão regular `ab+c` na string `abbbc`, utilizamos os seguintes comandos:

```
$ regex = EasyRegex::Regex.new('ab+c')
$ regex.match?('abbbc')
```

Caso ocorra um casamento entre a expressão regular e a *string*, `true` é retornado. Caso contrário, `false`.

Por padrão, a biblioteca possui um sistema de *log* bastante verboso que imprime todas as etapas do autômato e algumas informações adicionais. É possível esconder estas informações alterando o *level* do *log* de `Logger::DEBUG` para `Logger::ERROR` no arquivo `lib/easy_regex.rb`.

C. Manual do desenvolvedor

A biblioteca conta com testes unitários. A suite de testes pode ser executada através do comando:

```
$ rspec
```

A documentação do código foi feita respeitando a sintaxe do `YARD`[4]. É possível gerar as páginas de documentação da seguinte forma:

```
$ gem install yard
$ yard server
```

O último comando irá iniciar um servidor local que pode ser acessado pelo endereço http://localhost:8808. A navegação é feita pelas classes e métodos a partir do menu superior direito.

[4]Yay! A Ruby Documentation Tool: http://yardoc.org/