Protótipo reconhecedor de cadeias utilizando expressões regulares e autômatos de estado finito para a plataforma J2ME: Trabalhos iniciais

Thiago Galves Moretto[1]

[1]Centro de Ciências Exatas e da Terra
Universidade Católica Dom Bosco
Av. Tamandaré, 6000, Jardim Seminário
79117-900 Campo Grande, MS

thiago@moretto.eng.br

Resumo. *Este artigo apresenta os primeiros resultados dos testes de uma implementação de uma biblioteca para reconhecimento de cadeias de caracteres através de expressões regulares. Estes primeiros passos já utilizam a tecnologia J2ME (Java 2 Micro Edition - Plataforma para desenvolvimento de aplicações para aparelhos móveis em linguagem de programação Java), considerando suas limitações de processamento e memória.*

1. Introdução

Expressões regulares fornecem um meio prático para descrição de linguagens regulares por meio de símbolos e operadores, essa descrição auxilia no processo de reconhecimento de cadeias de caracteres em uma gama de aplicações como compiladores e processadores de texto [Donald E. Knuth, 1977]. Neste mesmo espoco entram os dispositivos reconhecedores de linguagens, os autômatos de estado finito determinísticos e não-determinísticos, estes, são os tópicos mais antigos da teoria da computação e que fornecem um modelo computacional mais adequado para ser aplicado [Yu, 1997, Harry R. Lewis, 2000].

Introduzimos essa teoria da computação, em especial na área de reconhecimento de padrões em cadeias de caracteres, para os aparelhos portáteis como os celulares, computadores de mão, entre outros, que estão cada vez mais populares em todo mundo, diversos fabricantes buscam melhorar e aprimorar estes aparelhos atraindo vários tipos de mercado consumidor, desde um usuário comum ou mesmo para automação e serviços. Este trabalho apresenta um protótipo de compilador de expressões regulares desenvolvido sobre a plataforma *J2ME*, que por sua vez não possui nenhum recurso de geração ou reconhecimento de linguagens regulares até este momento.

J2ME é uma plataforma e um conjunto de especificações para o desenvolvimento em uma variedade de dispositivos portáteis, desenvolvida pela *Sun Microsystems* [Microsystems, 2000] padronizada pela comunidade e uma série de fabricantes, fornece meios de produzir aplicativos portáteis utilizando a linguagem orientada a objetos *Java* [Muchow, 2004]. Para que todo esse ambiente funcione em diversos tipos de dispositivos e marcas há uma comissão regulamentando toda modificação e melhora da plataforma, chamada de *JCP* (*Java Community Process*).

A organização deste texto esta feita da seguinte forma, na seção 2 apresentaremos o *J2ME* e organização de sua arquitetura, na seção 3, o funcionamento do protótipo é apresentado, na seção 3.1 abordamos as operações elementares nas expressões regulares e

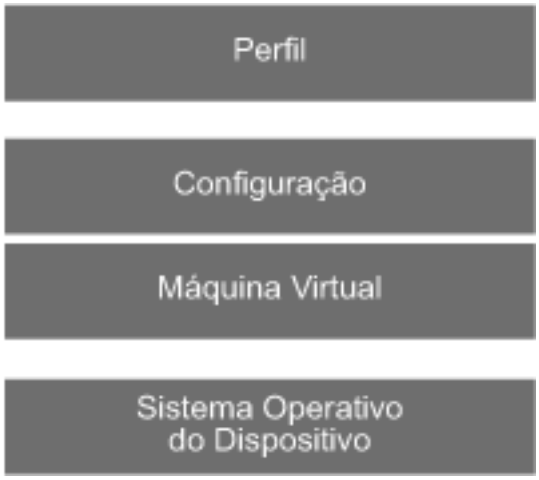

Figura 1: As três camadas da arquitetura da plataforma J2ME. Fonte: http://livromidp.jorgecardoso.org/html/index.html

os algoritmos utilizados. Na seção 3.2 mostramos as regras de construção das expressões do protótipo proposto. Na seção 4, abordamos os experimentos realizados. Na seção 5 apresentamos os resultados de testes e discusão. Na seção 6 comentamos trabalhos correlatos na área e por fim na seção 7, temos as considerações finais.

2. Java 2 Micro Edition: J2ME

J2ME é um conjunto de especificações para que seja possível fornecer um ambiente de execução de aplicações Java em dispositivos com baixo poder de processamento e memória, como celulares, *Smart Cards*, *PDA's* e *Set-top Boxes* [Microsystems, 2000]. Para isto a *Sun MicroSystems* e uma série de desenvolvedores e fabricantes definiram uma arquitetura em três camadas (ver Figura1), a máquina virtual, a configuração e o perfil [Muchow, 2004, Lamsal, 2001].

A camada da máquina virtual pode ser comparada ao funcionamento da máquina virtual *Java* existente para as aplicações *J2SE* (*Java 2 Standard Edition*)[2], mas com as limitações impostas pela arquitetura física do equipamento a máquina virtual precisa ser adaptada aos dispositivos em que é direcionada. Essa máquina virtual adaptada e reduzida é chamada de *KVM* (*Kilobyte Virtual Machine*), for projetada para ser menor e mais eficiente possível consumindo apenas algumas dezenas de *kilobytes*.

A camada superior à *KVM* é a configuração, definida pela especificação *CLDC* (*Connected Limited Device Configuration*) ou *CDC* (*Connected Device Configuration*), no qual esta definida uma série de bibliotecas e pacotes para o desenvolvedor ter acesso a serviços e recursos oferecidos pela máquina virtual, são nessas bibliotecas que a aplicação tem acesso a recursos de comunicação. Como os dispositivos diferem muito em recursos, processamento e memória, é na camada de configuração que esta definido o que pode ser oferecido ou não, para aquele modelo de dispositivo e o que é possível realizar com a memória e processamento disponível.

A camada de perfil é a mais próxima do desenvolvedor, definida pela especificação *MIDP* (*Mobile Information Device Profile*), fornece uma série de bibliotecas de apoio. A camada de perfil trata de recursos mais destinados às funcionalidades das aplicações em si, estruturas de dados, um simples banco de dados e outras bibliotecas para tratamento de informação. Na especificação *MIDP* encontram-se também o perfil do aparelho, tamanho da tela, botões disponíveis, entre outros.

[2]Plataforma padrão de desenvolvimento utilizando a linguagem Java. Fonte: http://java.sun.com

3. Protótipo reconhecedor de cadeias de caracteres

O protótipo reconhecedor de cadeias proposto e implementado, utiliza bibliotecas definidas pela especificação *MIDP*, ele recebe de entrada uma expressão regular, a transforma em um autômato finito não-deterministico com transições em vazio ($\epsilon - NFSA$), e fornece um autômato finito não-determinístico ($NFSA$), suprimindo as transições em vazio.

Este protótipo implementa algoritmos que trabalham sobre as operações elementares das expressões regulares, a concatenação, a união, e a estrela de *Kleene*, também conhecida por fecho transitivo reflexivo [Harry R. Lewis, 2000]. Estes operadores básicos tornam possíveis todas as composições para gerar qualquer linguagem regular, porém, no mundo real, criar composições somente com estes operadores se torna custoso, ou seja, expressões regulares extensas e complexas.

Utilizando exemplos de outros programas semelhantes[1], outros operadores foram criados neste protótipo. Estes novos operadores são de alto nível, ou seja, o analisador sintático da expressão tem uma tarefa a mais, substituir estes operadores (e expressões) por sua representação equivalente utilizando apenas os operadores elementares, descritos anteriomente. Desta forma, para o criador da expressão regular, geralmente o usuário final ou o desenvolvedor que esteja utilizando o protótipo, pode utilizar esta mais rica forma de criar as expressões, sem ter de saber dos muitos detalhes de sua implementação.

3.1. Operações elementares em expressões regulares

Para que o algoritmo realize as operações e forneça apenas um único autômato resultante $M = (K, \Sigma, S, \delta, F)$, regras são seguidas para cada tipo de operação. Estas permitem que o resultado seja um autômato finito não-determinístico com transições em vazio com o mesmo poder de representatividade que a expressão regular. Um autômato finito é um dispositivo mais prático de se representar computacionalmente, fornecendo um meio mais adequando para utilizá-lo como reconhecedores de linguages regulares [Harry R. Lewis, 2000, Hopcroft et al., 2001]. As operações que o protótipo trabalha são a união, concatenação e fecho transitivo reflexivo.

A união, representada na expressão regular pelo símbolo 'I', tem como característica produzir um autômato que reconhece qualquer uma de duas linguagem representadas por dois outros autômatos em um único. Essa operação pode ser representada da seguinte forma: seja duas linguagens reconhecidas por autômatos,$L[M1]$ e $L[M2]$, uma união é definida por $L[M] = L[M1] \cup L[M2]$. Formalmente o algoritmo para realizar a união é dado como:

$$L[M] = L[M_1] \cup L[M_2]$$
$$K = K_1 \cup K_2 \cup \{s\}$$
$$\delta = \delta_1 \cup \delta_2 \cup \{(s, \epsilon, s_1), (s, \epsilon, s_2), (x, \epsilon, F) \forall x \in F_1 \cup F_2\}$$

A *concatenação* de dois autômatos é colocá-los de maneira consecutiva, de tal forma que a cadeia a ser aceita é uma resultante da primeira com a segunda, uma após da outra e não ao contrário. Formalmente a *concatenação* $L[M] = L[M_1] \circ L[M_2]$ de dois autômatos finitos é mostrada a seguir. Nas expressões regulares é dada por "$a \circ b$"ou simplesmente "ab".

$$L[M] = L[M_1] \circ L[M_2]$$
$$K = K_1 \cup K_2 \cup \{s\}$$
$$\delta = \delta_1 \cup \delta_2 \cup \{(x, \epsilon, s_2) \forall x \in F_1\}$$

[1] http://java.sun.com/j2se/1.5.0/docs/api/java/util/regex/Pattern.html

O *fecho transitivo* e reflexivo (ou *estrela de Kleene*) é um operador unário e sufixo onde, em expressões regulares é simbolizado por '*' e '+' (quando não é *reflexivo*), é o mesmo que dizer que a cadeia ou simbolo operando pode repetir várias vezes ou até nenhuma. Dessa forma a *estrela de Kleene* também representa a cadeia vazia ϵ. Formalmente essa operação é dada por $L[M] = L[M_1]^*$, as regras a seguir são para adicionar a característica de *fecho transitivo reflexivo* no dado autômato.

$$L[M] = L[M_1]^*$$

$$K = K_1$$

$$\delta = \delta_1 \cup \{(x, \epsilon, s_1) \forall x \in F_1\}$$

O protótipo segue as regras para gerar um único autômato de estado finito não-determinístico com transições em vazio, com esse dispositivo já é possível reconhecer uma cadeia, dizer se ela pertence ou não à linguagem. Mas a generalização que os autômatos não-determinísticos apresentam é um problema no reconhecimento, podendo haver transições em vazio, e além disso duas transições para um mesmo símbolo de entrada, faz com que o algoritmo de reconhecimento tenha que testar todos os caminhos possíveis, para que o mesmo, ache uma sequência de movimentos que leve o consumo de toda a cadeia e o conduza para um estado final.

Algoritmo 1 Criação da tabela de estados alcançaveis com transições em vazio

entrada: M, o autômato finito não-determinístico com transições em vazio.
saída: T, a tabela *hash* com os grupos de estados alcançaveis com transições em vazio.

```
1: para cada S em M[K] faça
2:     para cada X em { (S, ε, X) } faça
3:         T(S) = T(S) ∪ X
4:     fim para
5: fim para
```

O protótipo segue algoritmos para remover transições em vazio. O **algoritmo 1** cria uma tabela $hash$ onde a chave é a referência de determinado estado e o valor apontado é um vetor com todos os estados alcançáveis com transições em vazio, isto é feito para todos os estados do autômato. Após isto, estes estados, são unidos em um só, removendo as transições em vazio e compondo as novas transições. O **algoritmo 2** realiza a união de estados que agregam estados em comum, este algoritmo é executado após a criação da tabela, que por sua vez é utilizada neste.

Algoritmo 2 Algoritmo de união de estados

entrada: S, o estado corrente.
saída: T, a tabela *hash* com os grupos de estados alcançaveis com transições em vazio.

```
1: para cada S em T faça
2:     se A ⊆ T(S) então
3:         T(A) = T(A) ∪ T(S)
4:         T(S) = φ
5:     fim se
6: fim para
```

Após a execução destes dois algoritmos apresentados, a partir da tabela resultante é gerado um autômato finito não-determinístico sem estados em vazio e com novos estados. Isto facilita o trabalho do algoritmo que testa a aceitação da entrada com o autômato finito gerado.

3.2. Regras de construção

Como dito anteriormente, as operações elementares já descritas fornecem todos os meios para representar qualquer linguagem regular. Neste protótipo introduzimos novos operadores e expressões que fornecem uma mais rica gama de representações de linguagens.

O analisador sintático (AS) do prótotipo solicita ao $\epsilon - NFSA$ várias composições de concatenação, união e de fecho, de acordo com a entrada. O AS quando encontra uma composição que não faz parte das expressões regulares básicas, verica se é um dos operadores de alto nível, se for, faz a tradução, e a solicita ao $\epsilon - NFSA$ compor as novas representações. Os operadores de alto nível são quatro, a *expressão de interval*, o *opcional* e o *repetidor*.

1. Expressão de intervalo: Este operador facilita a grafia de uma expressão que aceita em um determinado ponto um intervalo símbolos (baseados nos valores da tabela *ASCII*). A gramática desta expressão é $[m - z]$, onde m é o símbolo inicial e o z é o final, Se $m < z$ o AS alerta sobre um erro. A equivalência desta expressão para os operadores elementar é de: $(m|n|p...|z)$. (**Observação: Este operador não necessita outro operando senão os próprios.**).

2. Opcional: simbolizado por ?, este operador sufixo diz ao AS que o operando (ou expressão) anterior é opcional. O AS sinaliza ao $\epsilon - NFSA$ que o estado inicial da expressão anterior deve ser final.

3. Repetidor: é um operador sufixo simbolizado pela expressão n, onde n é um inteiro. O AS irá repetir a expressão anterior n vezes, e se esta for uma operação de alto nível, irá traduzi-lá depois.

4. Experimentos Realizados

Alguns experimentos foram realizados afim de obter resultados válidos, mensurar tempo, e validar as cadeias aceitas e rejeitadas. No protótipo foi utilizado uma classe que executa diversos testes em sequência validando as aceitações das cadeias e mensurando o tempo de geração da expressão regular em uma tabela de transições de estados.

A expressão E a seguir representa uma operação aritmética de dois operandos, um operador (entre eles, +, *, - e /), e um resultado. Considerando qualquer quantidade de espaços, entre, antes e após operadores e símbolos. Esta expressão foi obtida na intenção de verificar se uma a cadeia é uma expressão aritmética válida descrita pela expressão regular.

$$E = (\ *?)([0-9]+)(\ *?)(\backslash\backslash+|\backslash\backslash*|/|-)(\ *?)([0-9]+)(\ *?) = (\ *?)([0-9]+)(\ *?)$$

As cadeias a seguir foram aceitas pelo autômato resultante gerado pelo protótipo. Segundo as regras de construção estas cadeias são representadas pela expressão regular.

$w = 758\ +\ 231\ =\ 89993$

$w = 758 + 231\ \ = 89993$

$w =\ \ 758\ \ / 231\ \ = 89993$

As cadeias a seguir foram rejeitadas. Estas, não são representadas pela expressão regular, e devem ser rejeitadas pela autômato. Nos testes estas foram rejeitadas.

$w = 758\ \ 231\ \ \ \ = 89993$

$w = 758 + 231 =$

$w =\ \ + 231\ \ =$

Outro teste é utilizando parte de um comando *SQL*[2]. Gerenciadores de bancos de dados utilizando analisadores léxicos para validar e identificar uma entrada de comando *SQL*.

$E = (\ *?)SELECT\ (\ *?)((\backslash\backslash*)|[a-z]+)\ (\ *?)FROM\ (\ *?)[a-z]+$

As cadeias a seguir são geradas pela expressão regular. São aceitas pela autômato.

$w = \ SELECT \quad *FROMtabela$

$w = \ \ SELECT\ id\ FROM\ tabela$

$w = SELECT \quad id \quad\quad FROM \quad tabela$

Em contra partida, as cadeias a seguir não são geradas e não foram aceitas.

$w \ = \ SELECT\ FROM\ tabela\ WHERE$

$w \ = \ SELECT \ * \ FROM$

A expressão a seguir demonstra a utilização do *repetidor* e o *opcional*. Representa um número telefônico básico onde a primeira parte pode conter de três a quatro números, ou seja, três são obrigatórios mas o quarto é opcional. A segunda parte deve conter obrigatoriamente quatro números.

$([0-9]3)([0-9]?)-([0-9]4)$

As seguintes cadeias foram aceitas:

$w \ = \ 3363-2749$

$w \ = \ 363-2749$

$w \ = \ 241-2649$

As seguintes cadeias foram rejeitadas pois não são representadas pela expressão.

$w \ = \ 53-26494$

$w \ = \ 241-649$

Os teste apresentados até então apenas exemplificam uma aceitação por completo da cadeia de entrada. Há vários casos que aparece a necessidade de procura de uma palavra ou uma expressão em um texto. O protótipo, de padrão, faz aceitação de cadeia por completo, porém há a opção de aceitação por procura, ou seja, percorre o texto até encontrar um padrão que seja aceito pelo autômato.

Seja E a expressão: $mari(a|o)$, verificar se E esta contido em T, onde T é a frase a seguir:

maria é mais esperta que mario.

O algoritmo faz a aceitação desta cadeia pois encontra pelo menos um padrão aceito dentro da frase. Entretanto, a frase abaixo não contém nenhum padrão que é aceito, portanto, essa frase não é reconhecida pelo protótipo.

joão é mais rápido que augusto

5. Resultados e Discussão

A **tabela 2** apresenta além dos testes exemplificado, a estatíticas de todos os testes feitos, bem como, o número de entradas, a porcentagem de acertos, e o tempo gasto para,

[2]http://en.wikipedia.org/wiki/SQL

transformar a expressão regular e testar todas as entradas. A **tabela 1** é apenas uma referências das expressões regulares de cada teste. Pelos resultados obtidos, o tempo não foi uma questão que o protótipo trabalhou com êxito. As novas implementações devem ser voltadas para a otimização dos algoritmos propostos, e novos testes devem mostrar sua eficiência e eficácia. Todos os testes foram executados em uma máquina *desktop*.

Tabela 1: Tabela de expressões utilizadas

Teste	Expressão
1	$[0-9]([0-9]*)\backslash + [0-9]([0-9]*) = [0-9]([0-9]*)$
2	$[7-9][8-9]$
3	$(*?)([0-9]+)(*?)(\backslash + \mid \backslash * \mid / \mid -)(*?)([0-9]+)(*?) = (*?)([0-9]+)(*?)$
4	$[0-9]10$
5	$[a-z]([A-z]3)([a-z]2)$
6	$mari(a\|o)$
7	$(\ *?)SELECT\ (\ *?)((\backslash\backslash *)\|[a-z]+)\ (\ *?)FROM\ (\ *?)[a-z]+$
8	$([0-9]3)([0-9]?) - ([0-9]4)$

Tabela 2: Acertos, entradas e tempo gasto

Teste	% de acertos	Entradas	Tempo
1	100	5	146ms
2	100	9	3ms
3	100	12	116ms
4	100	8	104ms
5	100	11	698ms
6	100	7	10ms
7	100	5	189ms
8	100	5	47ms

Notando as expressões utilizadas, as que apresentavam muitas operações de união se mostraram mais lentas. A causa disto é que os algoritmos utilizados apresentam deficiência na remoção das transições em vazio e na composição dos novos estados compatíveis. Para verificar isto, **tabela 3** mostra a diminuição da quantidade de estados. As expressões que fazem o uso mais intenso de operação de união, acabam possuindo mais estados e em conseqüência os algoritmos ficam deficientes.

Tabela 3: Redução do número de estados

Teste	Antes	Depois	% Redução
1	232	6	98%
2	16	3	82%
3	256	6	98%
4	380	11	98%
5	996	7	99,9%
6	14	6	58%
7	454	17	97%
8	306	9	98%

6. Trabalhos correlatos

Existem diversos trabalhos na área de linguagens formais, Klein [Bruggemann-Klein, 1993] apresenta a construção de um autômato finito não-determinístico em tempo quadrático utilizando algoritmos e técnicas que evitam ambigüidades, Klein formaliza dizendo que um autômato finito não-determinístico, com transições em vazio (ϵ-NFA), não apresenta ambigüidade se para cada palavra w há apenas um caminho que conduza para um estado final, minimizando disperdício de processamento ao testar outros caminhos até que não haja mais possibilidades.

Hosoya [Hosoya, 2003], apresenta um trabalho similar ao de Klein propondo um algoritmo para verificação de ambigüidade assegurando que sempre vai haver um único caminho que reconhece uma cadeia de entrada. O mesmo trabalho tem como objetivo o desenvolvimento da técnica pra melhorar o processamento de arquivos *XML*.

7. Considerações Finais

Este trabalho apresentou problemas clássicos para uma tecnologia relativamente nova, o protótipo ainda em fase de implementação e testes possui determinada simplicidade, porém deixa claro a idéia e a possibilidade e a proposta de torná-lo produto. Há diversos algoritmos na área de teoria da computação que podem ser aplicados para otimização ou mostrar alternativas mais econômicas e equivalentes.

Os algoritmos propostos visam atender apenas a necessidade de funcionamento do protótipo, a melhora vem de buscar uma solução mais eficientes e que tenha sua eficácia comprovada. As próximas pesquisas visam chegar à um conjunto de soluções viáveis para atender essa necessidade. A teoria da computação e o conjunto de dispositivos de linguagens formais é vasto é não são novos, diversas implementações podem ser utilizadas como base para compor um novo protótipo que tenha sua funcionalidade comprovada.

Referências

Bruggemann-Klein, A. (1993). Regular expression into finite automata. *Theoret. Comput. Sci. 120*, pages 192–213.

Donald E. Knuth, J.H. Morris, V. P. (1977). Fast pattern matching in strings. *SIAM Journal on Computing, vol 6. no. 2*, pages 323–350.

Harry R. Lewis, C. H. P. (2000). *Elementos de Teoria da Computação*. Bookman, 2nd edition.

Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2001). Introduction to automata theory, languages, and computation, 2nd edition. *SIGACT News*, 32(1):60–65.

Hosoya, H. (2003). *Regular Expression Pattern Matching: A simpler design*. RIMS, Kyoto University, Kyoto, JP.

Lamsal, P. (2001). J2me architecture and related embedded technologies.

Microsystems, S. (2000). Java 2 platform micro edition (j2me) technology for creating mobile devices white paper. Technical report.

Muchow, J. W. (2004). *Core J2ME*. Pearson Makron Books, 1st edition.

Yu, S. (1997). Regular languages. *Handbook of Formal Languages, vol. I, Springer, Berlin*, pages 41–110.