WIKIPÉDIA

Expressão regular

Origem: Wikipédia, a enciclopédia livre.

Em ciência da computação, uma **expressão regular** ou "Regex" (ou os estrangeirismos ***regex*** ou **regexp** [1] , abreviação do inglês *regular expression*) provê uma forma concisa e flexível de identificar cadeias de caracteres de interesse, como caracteres particulares, palavras ou padrões de caracteres. Expressões regulares são escritas numa linguagem formal que pode ser interpretada por um processador de expressão regular, um programa que serve um gerador de analisador sintático ou examina o texto e identifica as partes que casam com a especificação dada.

O termo deriva do trabalho do matemático norte-americano Stephen Cole Kleene, que desenvolveu as expressões regulares como uma notação ao que ele chamava de álgebra de conjuntos regulares. Seu trabalho serviu de base para os primeiros algoritmos computacionais de busca, e depois para algumas das mais antigas ferramentas de tratamento de texto da plataforma Unix.

O uso atual de expressões regulares inclui procura e substituição de texto em editores de texto e linguagens de programação, validação de formatos de texto (validação de protocolos ou formatos digitais), realce de sintaxe e filtragem de informação.

Índice

Conceitos básicos

Uma expressão regular (ou, um padrão) descreve um conjunto de cadeias de caracteres, de forma concisa, sem precisar listar todos os elementos do conjunto. Por exemplo, um conjunto contendo as cadeias "*Handel*", "*Händel*" e "*Haendel*" pode ser descrito pelo padrão `H(ä|ae?)ndel`. A maioria dos formalismos provê pelo menos três operações para construir expressões regulares.

Alternância

Uma barra vertical (`|`) separa alternativas. Por exemplo, `psicadélico|psicodélico` pode casar "*psicadélico*" ou "*psicodélico*".

Agrupamento

Parênteses (`(`, `)`) são usados para definir o escopo e a precedência de operadores, entre outros usos. Por exemplo, `psicadélico|psicodélico` e `psic(a|o)délico` são equivalentes e ambas descrevem "*psicadélico*" e "*psicodélico*".

Quantificação (ou repetição)

Um quantificador após um *token* (como um caractere) ou agrupamento especifica a quantidade de vezes que o elemento precedente pode ocorrer. Os quantificadores mais comuns são o ponto de interrogação `?`, o asterisco `*` e o sinal de adição `+`.

`?` indica que há zero ou uma ocorrência do elemento precedente. Por exemplo, `ac?ção` casa tanto "*acção*" quanto "*ação*".

`*` indica que há zero ou mais ocorrências do elemento precedente. Por exemplo, `ab*c` casa "*ac*", "*abc*", "*abbc*", "*abbbc*", e assim por diante.

`+` indica que há uma ou mais ocorrências do elemento precedente. Por exemplo, `ab+c` casa "*abc*", "*abbc*", "*abbbc*", e assim por diante, mas não "*ac*".

Essas construções podem ser combinadas arbitrariamente para formar expressões complexas, assim como expressões aritméticas com números e operações de adição, subtração, multiplicação e divisão. De forma geral, há diversas expressões regulares para descrever um mesmo conjunto de cadeias de caracteres. A sintaxe exata da expressão regular e os operadores disponíveis variam entre as implementações.

História

A origem das expressões regulares está na teoria dos autômatos e na teoria das linguagens formais, e ambas fazem parte da teoria da computação. Esses campos estudam modelos de computação (autômatas) e formas de descrição e classificação de linguagens formais. Na década de 1950, o matemático Stephen Cole Kleene descreveu tais modelos usando sua notação matemática chamada de "conjuntos regulares", formando a álgebra de Kleene. A linguagem SNOBOL foi uma implementação pioneira de casamento de padrões, mas não era idêntica às expressões regulares. Ken Thompson construiu a notação de Kleene no editor de texto QED como uma forma de casamento de padrões em arquivos de texto. Posteriormente, ele adicionou essa funcionalidade no editor de texto Unix ed, que resultou no uso de expressões regulares na popular ferramenta de busca grep. Desde então, diversas variações da adaptação original de Thompson foram usadas em Unix e derivados, incluindo expr, AWK, Emacs, vi e lex.

As expressões regulares de Perl e Tcl foram derivadas da biblioteca escrita por Henry Spencer, e no Perl a funcionalidade foi expandida posteriormente.[2] Philip Hazel desenvolveu a PCRE (Perl Compatible Regular Expressions), uma biblioteca usada por diversas ferramentas modernas como PHP e o servidor Apache. Parte do desenvolvimento do Perl 6 foi melhorar a integração das expressões regulares de Perl, e aumentar seu escopo e funcionalidade para permitir a definição de gramáticas de expressão de analisadores sintáticos.[3] O resultado foi uma mini-linguagem, as regras do Perl 6, usada para definir a gramática do Perl 6 assim como fornecer uma ferramenta para programadores da linguagem. Tais regras mantiveram as funcionalidades de expressões regulares do Perl 5.x, mas também permitiram uma definição BNF de um analisador sintático descendente recursivo.

O uso de expressões regulares em normas de informação estruturada para a modelagem de documentos e bancos de dados começou na década de 1960, e expandiu na década de 1980 quando normas como a ISO SGML foram consolidadas.

Teoria de linguagens formais

Expressões regulares podem ser expressas através da teoria de linguagens formais. Elas consistem de constantes e operadores que denotam conjuntos de cadeias de caracteres e operações sobre esses conjuntos, respectivamente. Dado um alfabeto finito Σ, as seguintes constantes são definidas:

- (*conjunto vazio*) ∅ denotando o conjunto ∅
- (*cadeia de caracteres vazia*) ε denotando o conjunto {ε}
- (*literal*) *a* em Σ denotando o conjunto {*a*}

As seguintes operações são definidas:

- (*concatenação*) *RS* denotando o conjunto { αβ | α em *R* e β em *S* }. Por exemplo, {"*ab*", "*c*"}{"*d*", "*ef*"} = {"*abd*", "*abef*", "*cd*", "*cef*"}.
- (*alternância*) *R*|*S* denotando o conjunto união de *R* e *S*. Usa-se os símbolos ∪, + ou ∨ para alternância ao invés da barra vertical. Por exemplo, {"*ab*", "*c*"}∪{"*dv*", "*ef*"} = {"*ab*", "*c*", "*dv*", "*ef*"}
- (*Fecho de Kleene*) *R** denotando o menor superconjunto de *R* que contém ε e é fechado sob concatenação. Esse é o conjunto de todas as cadeias de caracteres que podem ser construídas ao concatenar zero ou mais cadeias em *R*. Por exemplo, {"*ab*", "*c*"}* = {ε, "*ab*", "*c*", "*abab*", "*abc*", "*cab*", "*cc*", "*ababab*", "*abcab*", ... }.

As constantes e os operadores acima formam a álgebra de Kleene.

Para evitar parênteses, é assumido que o fecho de Kleene possui a maior prioridade, depois a concatenação e por fim a alternância. Se não houver ambiguidades, os parênteses podem ser omitidos. Por exemplo, `(ab)c` pode ser escrito como `abc`, e `a|(b(c*))` pode ser escrito como `a|bc*`.

Exemplos

- `a|b*` denota {ε, *a*, *b*, *bb*, *bbb*, ...}
- `(a|b)*` denota o conjunto de todas as cadeias de caracteres que contém os símbolos *a* e *b*, incluindo a cadeia vazia: {ε, *a*, *b*, *aa*, *ab*, *ba*, *bb*, *aaa*, ...}
- `ab*(c|ε)` denota o conjunto de cadeias de caracteres começando com *a*, então com zero ou mais *b*s e opcionalmente com um *c*: {*a*, *ac*, *ab*, *abc*, *abb*, ...}

A definição formal de expressões regulares é concisa e evita a utilização dos quantificadores redundantes `?` e `+`, que podem ser expressados respectivamente por `(a|ε)` e `aa*`. Por vezes o operador de complemento `~` é adicionado; *~R* denota o conjunto das cadeias de caracteres de Σ* que

não estão em *R*. Esse operador é redundante, e pode ser expressado usando outros operadores, apesar da computação para tal representação ser complexa.

Expressões regulares podem expressar linguagens regulares, a classe de linguagens aceita por um autômato finito. Entretanto, há uma diferença significativa na compactação. Algumas classes de linguagens regulares podem ser descritas somente por autômatos que crescem exponencialmente em tamanho, enquanto o tamanho das expressões regulares requeridas só pode crescer linearmente. Expressões regulares correspondem ao Tipo-3 das gramáticas da Hierarquia de Chomsky. Por outro lado, existe um mapeamento simples de expressões regulares para máquinas de estado finito não-determinísticas que não leva ao crescimento desgovernado do tamanho. Por essa razão, essas máquinas não determinísticas são geralmente usadas como representação alternativa das expressões regulares.

É possível escrever um algoritmo que, para duas expressões regulares dadas, decide se as linguagens descritas são essencialmente iguais. Reduz-se cada expressão na máquina de estado finito mínima, e determina-se se ambas as máquinas mínimas são isomórficas (equivalentes).

Vale notar que diversas implementações de expressões regulares implementam funcionalidades que não podem ser expressadas na álgebra de Kleene; ver abaixo mais sobre o assunto.

Sintaxe

POSIX

De 1986, a norma IEEE POSIX 1003.2 (POSIX.2) padroniza expressões regulares, e fornece duas especificações, a saber:

- o conjunto básico (BRE) e
- o conjunto estendido (ERE).

BRE: expressões regulares básicas

A sintaxe tradicional de expressões regulares em Unix seguiu convenções comuns, mas diferiu entre as implementações. A norma IEEE POSIX BRE (*Basic Regular Expressions*, do inglês, expressões regulares básicas) foi desenvolvida primordialmente por compatibilidade com a sintaxe tradicional, mas fornecia uma norma comum que desde então foi adotada por diversas ferramentas.

Na sintaxe de BRE, a maioria dos caracteres são tratados como literais — eles casam somente com eles próprios (por exemplo, `a` casa "*a*"). As exceções são chamadas metacaracteres ou metassequências, definidos abaixo:

Metacaractere	Descrição
Padrões individuais	
`.`	Casa qualquer caractere. Algumas implementações excluem quebra de linha e codificação de caracteres. Nas expressões POSIX de listas de caracteres (ver logo abaixo), o caractere ponto é tratado como o literal. Por exemplo, `a.c` casa "*abc*", etc., mas `[a.c]` casa somente "*a*", "*.*" ou "*c*".
`[ ]`	Lista de caracteres. Casa uma ocorrência de qualquer caractere contido na lista. Por exemplo, `[abc]` casa "*a*", "*b*" ou "*c*". É possível definir intervalos de caracteres: `[a-z]` casa qualquer caractere de "*a*" a "*z*", e `[0123456789]` é igual a `[0-9]`. O caractere `-` é tratado como literal se for o primeiro ou o último da lista, ou se for escapado: `[abc-]`, `[-abc]` ou `[a\-bc]`.
`[^ ]`	Lista negada de caracteres. Casa uma ocorrência de qualquer caractere não contido na lista. Por exemplo, `[^abc]` casa qualquer caractere que não seja "*a*", "*b*" ou "*c*". `[^a-z]` casa qualquer caractere que não esteja em caixa baixa.
Âncoras	
`^`	Casa o começo da cadeia de caracteres. Numa situação de múltiplas linhas, casa o começo das linhas. Logo percebe-se que as âncoras não casam pedaços do texto, elas servem apenas como uma referência.
`$`	Casa o fim da cadeia de caracteres ou a posição logo antes da quebra de linha do fim da cadeia. Numa situação de múltiplas linhas, casa o fim das linhas.
Captura de dados	
BRE: `\( \)` ERE: `( )`	Grupo de captura. Marca uma subexpressão. A cadeia de caracteres que casa com o conteúdo dentro dos parênteses pode ser chamada posteriormente.
`\n`	Associado com o item anterior. Casa a n-ésima subexpressão marcada, em que *n* é um dígito de 1 a 9. Essa construção é teoricamente irregular e não foi adotada pela sintaxe POSIX ERE. Algumas ferramentas permitem referenciar mais de nove grupos de captura.
Quantificadores (ou repetidores)	
`*`	Casa o elemento precedente zero ou mais vezes. Por exemplo, `ab*c` casa "*ac*", "*abc*", "*abbbc*", etc.. `[xyz]*` casa "", "*x*", "*y*", "*z*", "*zx*", "*zyx*", "*xyzzy*", e assim por diante. `\(ab\)*` casa "", "*ab*", "*abab*", "*ababab*", e assim por diante.
BRE: `\{m,n\}` ERE: `{m,n}`	Casa o elemento precedente pelo menos *m* vezes e não mais que *n* vezes. Por exemplo, `a\{3,5\}` casa somente "*aaa*", "*aaaa*", e "*aaaaa*". Esta funcionalidade não é encontrada em algumas implementações muito antigas. Outras opções incluem omitir um dos campos. Por exemplo, `a\{3,\}` casa pelo menos três "a"s. Já `a\{3\}` casa somente três "a"s. `b{0,}` é análogo a `b*`, `b{0,1}` é análogo a `b?` (ver o quantificador `?` abaixo) e `b{1,}` é idêntico a `b+` (ver o quantificador `+` abaixo).

Uma característica da BRE é que os metacaracteres geralmente exigem barras invertidas para serem tratados como tal. Por exemplo, em BRE, `a{1,2}` é composto somente por literais, e casará somente "*a{1,2}*". Para casar entre uma a duas ocorrências de "a", deve-se usar a expressão regular `a\{1,2\}`. A motivação desse sistema é a compatibilidade com sistemas antigos, já que na época da padronização já havia código Unix legado que usava chaves como literais.

ERE: expressões regulares estendidas

O significado dos metacaracteres serem escapados com a barra invertida é revertido na sintaxe POSIX ERE (*Extended Regular Expression*, do inglês, expressões regulares estendidas). Isso significa que não são usadas barras invertidas para identificar metacaracteres. Pelo contrário, elas servem justamente para transformar metacaracteres em literais. Retomando o exemplo da seção anterior, em ERE, `a{1,2}` casa uma a duas ocorrências de "a", enquanto `a\{1,2\}` casa o literal "*a{1,2}*".

Os seguintes metacaracteres foram adicionados:

Quantificadores

`?` Casa o elemento precedente zero ou uma vez. Por exemplo, `ba?` casa "*b*" ou "*ba*".

`+` Casa o elemento precedente uma ou mais vezes. Por exemplo, `ba+` casa "*ba*", "*baa*", "*baaa*", e

assim por diante. Como visto na seção de teoria, esse metacaractere pode ser simulado em BRE através de `aa*`.

Alternadores

| Casa ou a expressão que precede ou a expressão que sucede. Por exemplo, `abc|def` casa "*abc*" ou "*def*".

Ferramentas que adotaram a sintaxe incluem MySQL e PHP, esta, que suporta também as derivações de Perl no modelo do PCRE[4].

Classes de caracteres

Já que diversos grupos de caracteres dependem duma configuração de locale específica, a POSIX define algumas classes (ou categorias) de caracteres para fornecer um método padrão de acesso a alguns grupos específicos de caracteres bastante utilizados, como mostrado na seguinte tabela:

`[:alnum:]`	Caracteres alfanuméricos, o que no caso de ASCII corresponde a `[A-Za-z0-9]`.
`[:alpha:]`	Caracteres alfabéticos, o que no caso de ASCII corresponde a `[A-Za-z]`.
`[:blank:]`	Espaço e tabulação, o que no caso de ASCII corresponde a `[ \t]`.
`[:cntrl:]`	Caracteres de controle, o que no caso de ASCII corresponde a `[\x00-\x1F\x7F]`.
`[:digit:]`	Dígitos, o que no caso de ASCII corresponde a `[0-9]`. O Perl oferece o atalho `\d`.
`[:graph:]`	Caracteres visíveis, o que no caso de ASCII corresponde a `[\x21-\x7E]`.
`[:lower:]`	Caracteres em caixa baixa, o que no caso de ASCII corresponde a `[a-z]`.
`[:print:]`	Caracteres visíveis e espaços, o que no caso de ASCII corresponde a `[\x20-\x7E]`.
`[:punct:]`	Caracteres de pontuação, o que no caso de ASCII corresponde a `[-!"#$%&'()*+,./:;<=>?@[\\\]_`{|}~]`.
`[:space:]`	Caracteres de espaços em branco, o que no caso de ASCII corresponde a `[ \t\r\n\v\f]`. O Perl oferece o atalho `\s`, que, entretanto, não é exatamente equivalente; diferente do `\s`, a classe ainda inclui um tabulador vertical, \x11 do ASCII.[5]
`[:upper:]`	Caracteres em caixa alta, o que no caso de ASCII corresponde a `[A-Z]`.
`[:xdigit:]`	Dígitos hexadecimais, o que no caso de ASCII corresponde a `[A-Fa-f0-9]`.

Notar que as doze classes definidas acima também estão definidas na biblioteca padrão do C, na seção de funções de testes de caracteres do cabeçalho `ctype.h`.

Tais classes só podem ser usadas dentro de expressões de listas de caracteres. Diferentes locales podem fornecer classes adicionais. Uma extensão não POSIX difundida é `[:word:]` (atalho do Perl `\w`), geralmente definida como `[:alnum:]` ou traço baixo (_) e `[:ascii:]`, contendo somente caracteres ASCII (`[\x01-\x7F]`).

Pode-se negar uma classe de caracteres precedendo um acento circunflexo ao nome da classe. Por exemplo, para negar `[:digit:]` usa-se `[:^digit:]`.[5]

Limites de palavras

A norma POSIX define ainda dois metacaracteres especiais que servem para casar os limites de palavras nas cadeias de caracteres. Nesse contexto da POSIX, uma palavra é formada por caracteres `[:alnum:]` ou traço baixo (_). Assim como as âncoras, esses metacaracteres não casam pedaços do texto, elas servem apenas como uma referência. Eles são:

`[[:<:]]` Casa o começo de palavras.

`[[:>:]]` Casa o fim de palavras.

Perl e derivações

Perl possui uma sintaxe mais consistente e rica que as normas POSIX BRE e ERE. Um exemplo é que \ sempre escapa um caractere não alfanumérico. Devido ao poder de expressão, outras ferramentas adotaram a sintaxe do Perl, como por exemplo Java[6], JavaScript, PCRE, Python[7], Ruby e .NET[8]. Algumas linguagens e ferramentas como PHP suportam diversos tipos de expressões regulares.

Um exemplo de funcionalidade possível em Perl mas não em POSIX é a quantificação preguiçosa. Os quantificadores padrões das expressões regulares são "gananciosos", isto é, casam o quanto puderem, voltando atrás somente se necessário para casar o resto da expressão regular. Por exemplo, um novato no assunto tentando encontrar a primeira instância de um item entre os símbolos < e > no texto "*Outra explosão ocorreu em <26 de janeiro> de <2004>*" provavelmente usaria o padrão `<.*>`, ou similar. Entretanto, esse padrão retornará "*<26 de janeiro> de <2004>*" ao invés de "*<26 de janeiro>*", como esperado, pois o quantificador * é ganancioso — ele consumirá a quantidade máxima de caracteres, e "*26 de janeiro> de <2004*" possui mais caracteres que "*26 de janeiro*".

Apesar desse problema ser evitável de diferentes formas (por exemplo, especificando o que não casar: `<[^>]*>`), a maioria das ferramentas permitem que um quantificador seja preguiçoso, ou não ganancioso, ao suceder o quantificador com um ponto de interrogação. No exemplo anterior, a alternativa seria `<.*?>`. Seguem os quantificadores não gulosos:

Quantificadores não gulosos

`??`	Versão não gulosa de `?`. Dado o texto "*aaa*", `a?` casa "*a*" enquanto `a??` casa "".
`*?`	Versão não gulosa de `*`. Dado o texto "*aaa*", `a*` casa "*aaa*" enquanto `a*?` casa "".
`+?`	Versão não gulosa de `+`. Dado o texto "*aaa*", `a+` casa "*aaa*" enquanto `a+?` casa "*a*".
`{m,n}?`	Versão não gulosa de `{m,n}`. Dado o texto "*aaa*", `a{2,3}` casa "*aaa*" enquanto `a{2,3}?` casa "*aa*".

O PERL define algumas sequências de escape que servem como atalhos para certos metacaracteres:

Padrões individuais

- `\s` Casa espaços em branco, \n \r ou \t.
- `\S` Negação de \s: casa o que não for espaço em branco, \n \r ou \t.
- `\w` Casa letras, dígitos, ou '_'.
- `\W` Negação de \w
- `\d` Casa dígitos, de 0 a 9.
- `\D` Negação de \d

Âncoras

- `\b` Casa a separação de palavras, o que inclui também o começo (`^`) e o fim (`$`) da cadeia de caracteres testada. A definição dos caracteres que formam palavras varia de acordo com a implementação, mas é seguro assumir pelo menos `[a-zA-Z0-9_]`. Havendo suporte, o atalho `\w` é uma alternativa válida. O Java é uma notável exceção na medida em que suporta `\b` mas não `\w`.

Notar que apesar de parecida com os limites de palavras definidos pela POSIX, esta sequência de escape não distingue o começo e o final da palavra, somente a separação em si.

`\B` Negação de \b

`\A` Casa o começo da cadeia de caracteres. Numa situação de múltiplas linhas, não casa o começo das linhas seguintes, o que a difere de `^`.

`\Z` Casa o fim da cadeia de caracteres ou a posição logo antes da quebra de linha do fim da cadeia. Numa situação de múltiplas linhas, não casa o fim das linhas seguintes, o que a difere de `$`.

`\z` Casa o fim da cadeia de caracteres.

Além dos quantificadores preguiçosos e das novas sequências de escape, o Perl também adicionou uma forma nova de casamento de padrões que estendem a POSIX. São um conjunto de metacaracteres que seguem o padrão `(?…)`, listados abaixo:

`(?# )` Adiciona um comentário, ignorado nos casamentos.

`(?: )` Grupo de captura que não é salvo para nova chamada posterior (`\1`, `\2`, `\3`, ...)

`(?= )` Casa ocorrências do padrão atual a partir da posição atual até o final. Assim como as âncoras (`^`, `$`), o padrão não é incluído no casamento, servindo apenas na verificação. Por exemplo, o padrão `carro(?=azul)` aplicado a "*Um carro esportivo azul barato.*" casará "*carro*".

`(?! )` Negação de `(?=)`, casa a ausência do padrão atual a partir da posição atual até o final, e também não inclui o padrão no casamento. Por exemplo, o padrão `carro(?!amarelo)` casará em "*Um carro esportivo azul barato.*", entretanto `carro(?!azul)` não casará.

`(?<= )` Casa ocorrências do padrão atual a partir do começo até a posição atual, mas não inclui o padrão no casamento, servindo apenas na verificação. Por exemplo, o padrão `(?<=carro)azul` aplicado a "*Um carro esportivo azul barato.*" casará "*azul*".

`(?<! )` Negação de `(?<=)`, casa a ausência do padrão atual a partir do começo até a posição atual, e também não inclui o padrão no casamento. Por exemplo, o padrão `(?<!carro)amarelo` casará em "*Um carro esportivo azul barato.*", entretanto `(?<!carro)azul` não casará.

`(? )` Aplica modificadores a partes da expressão regular. Os valores aceitos são i (ignora diferenças entre maiúsculas e minúsculas), m (texto multilinha), s (texto monolinha) e x (inclusão de espaços e comentários).

`(?(A)B|C)` Um operador ternário, usado geralmente em conjunto dos grupos de captura. Por exemplo, a escolha duma ou outra subexpressão regular depende dum casamento ou não anterior.

`(?(A)B)` Variação binária da estrutura anterior, em que uma expressão é adicionada ao padrão se e somente se a condição é verdadeira.

Padrões para linguagens não regulares

Diversas funcionalidades encontradas em bibliotecas atuais de expressões regulares provem um poder de expressão que excede as linguagens regulares. Por exemplo, a habilidade de agrupar subexpressões com parênteses e chamar novamente o valor casado na mesma expressão significa que o padrão pode casar cadeias de palavras repetidas como "*papa*" ou "*WikiWiki*", os chamados quadrados na teoria de linguagens formais. O padrão para essas cadeias é `(.*)\1`. Entretanto, a linguagem de quadrados não é regular, nem livre de contexto. Casamento de padrões com um número indeterminado de referências anteriores, como suportado em ferramentas atuais, é NP-difícil.

Entretanto, as ferramentas que fornecem tais construções ainda usam o termo expressões regulares para os padrões, o que leva a uma nomenclatura que difere da teoria das linguagens formais. Por essa razão, algumas pessoas usam o termo *regex* ou simplesmente *padrão* para descrever esse conceito mais abrangente.

Implementações

Existem pelo menos dois algoritmos fundamentalmente diferentes entre si que decidem se e como uma expressão regular casa uma cadeia de caracteres.

O mais antigo e mais rápido faz uso dum princípio da teoria de linguagens formais que permite a todas as máquinas de estado finito não determinísticas serem transformadas em máquinas de estado finito determinísticas. Geralmente chamado de DFA, o algoritmo realiza ou simula tal transformação e então executa a máquina determinística resultante na cadeia de caracteres, um símbolo de cada vez. Esse último processo tem complexidade linear relativa ao tamanho da cadeia de caracteres. Mais precisamente, uma cadeia de caracteres de tamanho n pode ser testada numa expressão regular de tamanho m no tempo $O(n + 2^m)$ ou $O(nm)$, dependendo dos detalhes de implementação. Esse algoritmo é rápido, mas pode ser usado somente para casamentos e não para a nova chamada de grupos de captura, quantificação preguiçosa e diversas outras funcionalidades encontradas nas bibliotecas modernas de expressões regulares. Também é possível executar a máquina não determinística diretamente, construindo cada estado da máquina determinística quando necessário e então descartando-o no próximo passo. Isso evita a quantidade exponencial de memória necessária para a construção completa da máquina determinística, ainda que garantindo a busca em tempo linear.[9]

O outro algoritmo é casar o padrão com a cadeia de caracteres através de *backtracking*. Geralmente chamado de NFA, Seu tempo de execução pode ser exponencial, o que pode acontecer em implementações simples no casamento de expressões como `(a|aa)*b`, que forçam o algoritmo a considerar um número exponencial de subcasos. Implementações modernas geralmente identificam tais casos, e aceleram e abortam a execução. Apesar dessas implementações com *backtracking* garantirem tempo exponencial no pior caso, elas fornecem mais flexibilidade e poder de expressão.

Relacionamento com Unicode

Originalmente, as expressões regulares eram usadas com caracteres ASCII, mas várias implementações atuais suportam Unicode. Na maioria dos casos não há diferença entre conjuntos de caracteres, mas algumas questões são relevantes ao suportar Unicode.

Uma delas é a codificação suportada, já que algumas implementações esperam UTF-8, enquanto outras podem esperar UTF-16 ou UTF-32. Outra questão é estender as funcionalidades disponíveis para ASCII no Unicode. Por exemplo, em implementações ASCII, conjuntos de caracteres na forma `[x-y]` são válidos para quaisquer x e y no intervalo [0x00,0x7F] desde que o código de x seja menor que o código de y. A escolha natural seria permitir o mesmo em Unicode no intervalo de códigos [0,0x10FFFF], o que não é possível pois algumas implementações não permitem que conjuntos de caracteres ultrapassem os blocos de código disponíveis.

Do ponto de vista dos detalhes técnicos do Unicode, também surgem questões. Como a normalização, pois, em Unicode, mais de um código pode representar o mesmo caractere. Por exemplo, o caractere "é" pode ser representado por U+0065 (letra latina "e" minúsculo) combinado com U+0301 (diacrítico "acento agudo"), mas também pode ser representado como U+00E9 (letra latina "e" com diacrítico "acento agudo"). Também há os códigos de controle Unicode, as marcas de ordem de byte e as marcas de direção de texto, que devem ser tratados separadamente.

Uso

Expressões regulares são usadas por diversos editores de texto, utilitários e linguagens de programação para procurar e manipular texto baseado em padrões. Por exemplo, Perl e Tcl possuem suporte a expressões regulares nativamente. Diversos utilitários de distribuições Unix incluem o editor de texto ed, que popularizou o conceito de expressão regular, e o filtro grep.

Outro uso é a validação de formatos de texto (validação de protocolos ou formatos digitais). Por exemplo, ao receber a entrada dum campo de formulário duma aplicação que supõe receber um endereço de *email*, pode-se usar uma expressão regular para garantir que o que foi recebido de fato é um endereço de *email*.

Mais um uso é a implementação interna dum sistema de realce de sintaxe, como encontrado em ambientes de desenvolvimento integrado. Expressões regulares podem ser usadas para encontrar palavras reservadas, literais e outros tokens específicos, e para alterar a formatação do texto de acordo com o casamento feito.

Um uso difundido de expressões regulares é a filtragem de informação em bancos de dados de texto. Por exemplo, num arquivo de texto contendo cadastros de pessoas e suas datas de aniversário como a seguir:

```
1954-10-01 João Alberto
1976-07-25 Maria Eduarda
1966-10-22 Carlos Silva
```

Pode-se filtrar pessoas que nasceram num determinado ano, mês ou dia. Por exemplo, o uso do padrão `^[0-9]{4}-10-[0-9]{2} (.*)$` identifica o nome das pessoas que nasceram em outubro (mês 10). Para o cadastro acima seriam retornados dois grupos de captura, `\1` contendo "*João Alberto*" e `\2` contendo "*Carlos Silva*". Explorando o exemplo anterior e o uso de validação de formatos digitais, é possível usar expressões regulares para validar as datas presentes no arquivo de texto de aniversários acima. O padrão `(19|20)\d\d([- /.])(0[1-9]|1[012])\2([012][0-9]|3[01])` é usado para validar uma data entre 1900-01-01 e 2099-12-31.[10] Atentar que a separação entre ano, mês e dia pode se dar através de hífen, espaço em branco, barra ou ponto. Mas deve-se usar o mesmo símbolo de separação entre ano e mês e entre mês e dia, o que é possível através da nova chamada do grupo de captura anterior (o trecho `\2` do padrão). Atentar também que o padrão é incompleto na medida em que não diferencia a quantidade de dias em cada mês, o que resulta no casamento de até "*2000-02-31*" (31 de fevereiro), incorreta de acordo com o calendário gregoriano.

Referências

1. What Regular Expressions Are Exactly - Terminology (http://www.regular-expressions.info/tutorial.html)
2. Larry Wall e o time de desenvolvimento do Perl 5 (2006). «perlre: Perl regular expressions» (http://perldoc.perl.org/perlre.html) (em inglês)
3. Larry Wall (4 de junho de 2002). «Apocalypse 5: Pattern Matching» (http://dev.perl.org/perl6/doc/design/apo/A05.html) (em inglês)
4. «POSIX Regex: Introduction» (http://www.php.net/manual/en/intro.regex.php) (em inglês). 18 de julho de 2008. Consultado em 24 de julho de 2008
5. «Perl 5.10.0 documentation» (http://perldoc.perl.org/perlre.html) (em inglês). Consultado em 25 de julho de 2008
6. «Lesson: Regular Expressions» (http://java.sun.com/docs/books/tutorial/essential/regex/index.html). *The Java Tutorials* (em inglês). Sun Microsystems. Consultado em 25 de julho de 2008
7. A.M. Kuchling (19 de julho de 2008). «Regular Expression HOWTO» (https://docs.python.org/dev/howto/regex.html). *Documentação do Python* (em inglês). Python Software Foundation. Consultado em 24 de julho de 2008
8. «.NET Framework Regular Expressions» (http://msdn.microsoft.com/en-us/library/hs600312(VS.71).aspx). *.NET Framework Developer's Guide* (em inglês). Microsoft. Consultado em 25 de julho de 2008
9. Russ Cox (Janeiro de 2007). «Regular Expression Matching Can Be Simple and Fast» (http://swtch.com/~rsc/regexp/regexp1.html) (em inglês). Consultado em 20 de julho de 2008

10. Jan Goyvaerts (14 de março de 2008). «Example: Regular Expression Matching a Valid Date» (http://www.regular-expressions.info/dates.html) (em inglês). Consultado em 25 de julho de 2008

- «Regular Expressions» (http://www.opengroup.org/onlinepubs/007908799/xbd/re.html) (em inglês). The Open Group. 1997. Consultado em 19 de julho de 2008
- Aurélio Marinho Jargas (2009). *Expressões Regulares - Uma Abordagem Divertida* (http://www.piazinho.com.br) 3 ed. [S.l.]: Novatec. 208 páginas. ISBN 978-85-7522-212-6
- Ben Forta. *Sams Teach Yourself Regular Expressions in 10 Minutes*. [S.l.]: Sams. ISBN 0-672-32566-7
- Jeffrey Friedl. *Mastering Regular Expressions*. [S.l.]: O'Reilly. ISBN 0-596-00289-0
- Mehran Habibi. *Real World Regular Expressions with Java 1.4*. [S.l.]: Springer. ISBN 1-59059-107-0
- Francois Liger; Craig McQueen, Paul Wilton. *Visual Basic .NET Text Manipulation Handbook*. [S.l.]: Wrox Press. ISBN 1-86100-730-2
- Michael Sipser. *Introduction to the Theory of Computation*. [S.l.]: PWS Publishing. pp. 31–90. ISBN 0-534-94728-X
- Tony Stubblebine. *Regular Expression Pocket Reference*. [S.l.]: O'Reilly. ISBN 0-596-00415-X

Ver também

- Editor de texto
- Teoria da computação
- Autômato
- Linguagem regular
- Casamento de padrões

Ligações externas

- Manual de Regex (http://guia-er.sourceforge.net)
- Portal sobre expressões regulares (http://aurelio.net/er)
- Wiki sobre expressões regulares (http://www.regex.pro.br)
