Capítulo 1: Um Casador de Expressões Regulares

1.1 - Introdução

Código bonito geralmente é simples—claro e fácil de entender. Código bonito geralmente é compacto—o suficiente para fazer o trabalho bem feito e nada mais—mas não críptico a ponto de não poder ser entendido. Código bonito pode também ser geral, resolvendo uma ampla classe de problemas de forma uniforme. Ele pode ser descrito como elegante, uma amostra de bom gosto e refinamento.

Neste capítulo eu irei descrever um pedaço de código bonito, para casar expressões regulares que se encaixa em todos estes critérios.

Expressões regulares são uma notação para descrever padrões de texto, de fato é uma linguagem específica para casar padrões. Embora hajam muitas variações, todas compartilham a ideia de que a maioria dos caracteres em um padrão se casam com ocorrências literais deles mesmos, mas alguns "metacaracteres" tem significados especiais, por exemplo "*" indica algum tipo de repetição ou [...] significa qualquer um dos caracteres entre os colchetes.

Na prática, a maioria das buscas em programas com editores de texto são para palavras literais, então as expressões regulares são geralmente textos literais como "print" que irá casar "sprint" ou "printer" em qualquer lugar. Nos assim chamados meta-caracteres de nomes de arquivos em Unix e Windows, um "*" casa com qualquer número de caracteres, então o padrão "*.c" casa com todos os arquivos que terminam com ".c". Existem muitas, muitas variações das expressões regulares, mesmo em contextos nos quais as pessoas esperariam que fôssem as mesmas. O livro "Dominando Expressões Regulares" de Jeffrey Friedl é um estudo exaustivo do tópico.

Expressões regulares foram inventadas por Stephen Kleene no meio dos anos 50 como uma notação para autômatos finitos, e de fato elas eram equivalentes aos autômatos finitos que elas represenavam. Expressões regulares apareceram pela primeira vez nas configurações de um programa na versão de ken Thompson do editor de textos QED no meio dos anos 60. Em 1967, Ken pediu o registro de uma patente sobre um mecanismo para encontrar rapidamene textos que casavam com uma expressão regular; a qual foi concedida em 1971, uma das primeiras patentes de software nos Estados Unidos.

Expressões regulares se mudaram do QED para o editor Unix "ed", e então para a feramenta quintessencial do unix, o "grep", a qual foi criada por Ken ao realizar uma cirurgia radical no "ed". Através de todos estes programas amplamente usados, expressões regulares tornaram-se familiares por toda a comunidade Unix inicial.

O casador original de Ken era muito rápido, pois ele combinava duas ideias independentes. Uma era gerar instruções de máquina sob medida durante o casamento de padrões de forma que ela executasse na velocidade da máquina, não da interpretação. A outra era seguir adiante com todos os padrões casados em cada estágio, assim não haveria necessidade de voltar atrás para procurar por novos trechos de texto que poderiam casar com a expressão. O código do casador nos próximos editores de texto que Ken escreveu, como "ed", usava um algoritmo mais simples que voltava atrás quando necessário. Na teoria isso era mais lento mas nos padrões encontrados na prática raramente era necessário voltar atrás, então o código de "ed" e "grep" era bom o bastante para a maioria dos propósitos.

Os próximos casadores de expressões regulares como "egrep" e 'fgrep" adicionaram classes mais ricas de expressões regulares, e se focaram na execução rápida dos padrões, fôssem eles quais fôssem. Expressões regulares ainda mais complexas tornaram-se populares, e eram incluídas não só em biblioecas baseadas em C, mas também como parte da sintaxe de linguagens de script como Awk e Perl.

1.2 - A Prática de Programação

Em 1998, Rob Pike e eu estávamos escrevendo "A Prática da Programação". O último capítuulo do livro, "Noação", era uma coleção de um número de exemplos nos quais uma boa

noação levou a melhores programas e melhores programações. Isso incluia o uso de estruturas de dados simples (como o formato do "printf") e a geração de código à partir de tabelas.

Dada a nossa experiência em Unix e os muitos anos de experiência com ferramentas baseadas na notação de expressões regulares, nós naturalmente queríamos incluir uma discussão de expressões regulares, e parecia obrigatório que incluíssemos uma implementação também. Dada a nossa ênfase em ferramentas, também parecia melhor focar na classe de expressões regulares encontradas no "grep" ao invés dos caracteres especiais do shell,assim nós também poderíamos falar sobre o próprio projeto do "grep".

O problema era que qualquer pacote existente de casar expressões regulares era grande demais. O "grep" local tinha 500 linhas (cerca de 10 páginas do livro). Pacotes de expressão regular em software livre tendiam a ser enormes, basicamente tinham o tamanho do livro inteiro, pois eles haviam sido projetados para a generalidade, flexibilidade e velocidade; nenhum era adequado para a pedagogia.

Eu sugeri a Rob que nós precisávamos encontrar o menor pacote de expressões regulares que ilustrassem as ideias básicas enquanto ainda reconhecêssem uma classe não-trivial e útil de padrões. Idealmente o código deveria caber em uma única página.

Rob desapareceu em seu escritório, e pelo menos como eu me lembro, apareceu novamente em não mais do que uma ou duas horas com as 30 linhas de código C que apareceriam no Capítulo 9 do livro. Aquele código implementa um marcador de expressão regular que suporta os seguintes elementos:

c Casa com qualquer caractere literal c.

. Casa com um único caractere qualquer.

^ Casa com o começo de uma cadeia de caracteres.

$ Casa com o fim de uma cadeia de caracteres.

* Casa com zero ou mais ocorrências do caractere anterior.

Esa é uma classe de casos bastante útil; em minha própria experiência de uso de expressões regulares, isso facilmente abrange 95% de todos os casos. Em muitas situações, resolver o problema certo é um grande passo no caminho para um programa bonito. Rob merece crédito por escolher tão sabiamente dentre tantas opções, um conjunto de funcionalidade muito pequena, mas bem-definida e extensível.

A implementação de Rob em si é um exemplo perfeito de código bonito: compacto, elegante, eficiente e útil. É um dos melhores exemplos de recursão que eu já vi e mostra o poder dos ponteiros em C. Embora na época nós estivéssemos mais interessados em mostrar o papel importante de uma boa notação em tornar um programa mais fácil de usar e talvez mais fácil de escrever também, o código de expressão regular também foi uma forma excelente de ilustrar algoritmos, estruturas de dados, melhoria de performance e ouros tópicos importantes.

1.3 - Implementação

No livro, o casador de expressões regulares é parte de um programa que imita "grep", mas o código de expressão regular é completamente separável de sua vizinhança. O programa principal não é interessante aqui—ele simplesmente lê da entrada padrão ou de uma sequencia de arquivos, e imprime aqueles cujas linhas contém texto que casa com a expressão regular, assim como o "grep" originaç e muitas outras ferramentas Unix.

Este é o código de casamento:

Seção: Casador de Expressões Regulares:

```
/* match: search for regexp anywhere in text */
int match(char *regexp, char *text){
  if(regexp[0] == '^')
    return matchhere(regexp+1, text);
  do{    /* must look even if string is empty */
    if(matchhere(regexp, text))
      return 1;
  } while (*text++ != '\0');
  return 0;
```

```
}
/* matchhere: search for regexp at beginning of text */
int matchhere(char *regexp, char *text){
  if (regexp[0] == '\0')
    return 1;
  if (regexp[1] == '*')
    return matchstar(regexp[0], regexp+2, text);
  if (regexp[0] == '$' && regexp[1] == '\0')
    return *text == '\0';
  if (*text!='\0' && (regexp[0]=='.' || regexp[0]==*text))
    return matchhere(regexp+1, text+1);
  return 0;
}
/* matchstar: search for c*regexp at beginning of text */
int matchstar(int c, char *regexp, char *text){
  do {    /* a * matches zero or more instances */
    if (matchhere(regexp, text))
      return 1;
  } while (*text != '\0' && (*text++ == c || c == '.'));
  return 0;
}
```

A função `match(regexp, text)` testa se há uma ocorrência da expressão regular em qualquer lugar dentro do texto; ela retorna 1 se um casamento é encontrado e 0 caso contrário. Se há mais de um casamento, ela encontra o mais à esquerda e menor.

A operação básica de `match` é direta. Se o primeiro caractere da expressão regular é ˆ qualquer casamento possível deve ocorrer no começo da cadeia de caracteres. Isto é, se a expressão regular é ˆxyz, ela deve casar "xyz" só se "xyz" ocorrer no começo do texto, não no meio. Isso é testafo casando o resto da expressão regular somente com o começo do texto, e não com o resto.

Fora deste caso, a expresão regular iria casar com qualquer padrão dentro da string. Isso é testado casando o padrão contra cada posição de caractere do texto. Se há múltiplos casamentos, somente o primeiro será identificado.

Note que avançar na cadeia de caracteres de entrada é feito com um laço `do-while`, algo relativamente incomum na construção de programas em C.A ocorrência de um `do-while` deve sempre levantar uma questão: porque a condição de terminar o laço não é testada no começo do laço, antes que seja tarde demais? Mas o teste está correto aqui: como o operador * permite casar cadeias de caractere vazias, nós primeiro temos que testar se um casamento vazio é possível.

O grosso do trabalho é feito na função `matchhere(regexp, text)`, a qual testa se a expressão regular casa com o texto que começa logo ali. A função opera tentando casar o primeiro caractere da expressão regular com o primeiro caractere de texto. Se o casamento falhar, não pode haver casamento nesta posição do texto e `matchhere` retorna 0. Se o casamento for bem-sucedido, contudo, é possível avançar para o próximo caractere da expressão regular e o próximo caractere do texto. Isso é feito chamando `matchhere` recursivamente.

A situação é um pouco mais complicada por causa de alguns casos especiais, e é claro, a necessidade de interromper a recursão. O caso mais fácil é quando a expressão regular está no seu fim (`regexp[0] == '\0'`), então todos os testes anteriores foram bem-sucedidos, e assim a expressão regular casa com o texto.

Se uma expressão regular é um caracere seguido por *, `matchstar` é chamada para ver se a palavra casa com o fechamento. A função `matchstar(c, regexp, text)` tenta casar repetições do caractere "c", começando com zero repetições e contando atéque consiga casar com o restante do texto ou falhe e conclua não ser possível um casamento. Isso identifica um "menor casamento", o que é bom para simples casamentos de padrões como em "grep", onde tudo o que importa é encontrar o casamento o mais rápido possível. Um "maior casamento" é mais intuitivo e quase certamente melhor para umeditor de textos onde o texto casado será substituído. A maioria das bibliotecas de expressões regulares fornece ambas as alternativas, e o nosso exemplo apresenta uma simples variante de `matchstar` para este caso, mostrada depois.

Se a expressão regular é um $ no im da expressão, então o texto casa somente se estiver no fim.

Cso contrário, se nós não esamos no fim da cadeia de caracteres e se o primeiro caractere do texto casa com o primeiro caractere da expressão regular, tudo continua seguindo bem e iremos então testar se o próximo caractere de texto também casa com o próximo caractere da expressão regular fazendo uma chamada recursiva para `matchhere`. Esta chamada recursiva é o coração do algoritmo e o por quê do código ser tão compacto e limpo.

Se todas estas tentativas falharem, então não pode haver um casamento neste ponto na expressão regular, e então `matchhere` retorna 0.

Este código definitivamente usa ponteiros de C. Em cada estágio da recursão, se alguma coisa casa, a chamada recursiva que se segue usar aritmética de ponteiros para que a próxima função seja chamada com o próximo caractere da expressão regular e próximo caractere do texto. A profundidade da recursão não será maior que o tamanho do padrão, o que é normal se o seu uso for pequeno, mas que deixa o risco para esgotarmos todo o espaço da pilha.

1.4 - Alternativas

Esta é uma porção de código muito elegante e bem-escrita, mas não é perfeita. O que pode ser feito diferente? Pode-se reorganizar `matchhere` para lidar primeiro com $ antes de *. Embora não faça diferença aqui, me parece mais natural, e uma boa regra é cuidar dos casos simples antes dos mais difíceis.

Mas via de regra, contudo, a ordem dos testes é crítica. Por exemplo, neste teste de `matchstar`:

Seção: Trecho de matchstar:

```
    } while (*text != '\0' && (*text++ == c || c == '.'));
```

não importa o que aconteça, nós devemos sempre avançar para o próximo caractere da cadeia de caracteres, então o incremento em `text++` deve sempre ser feito.

Este código é cuidadoso com as condições de término. Geralmente, o sucesso de um casamento é determinado quando terminamos de percorrer a expressão regular ao mesmo tempo que o texto. Se ambos acabam ao mesmo tempo, isso indica um casamento; se um termina antes do outro, não há casamento. Isso é talvez mais óbvio em uma linha como:

Seção: Condição de Parada:

```
  if (regexp[0] == '$' && regexp[1] == '\0')
    return *text == '\0';
```

mas condições de parada mais sutis aparecem em outros lugares também.

Esta versão de `matchstar` que implementa o maior casamento mais à esquerda começa identificando uma sequencia máxima de ocorrências do caractere de entrada "c". Então, ela usa `matchhere` para tentar extender o casamento para o resto dos padrões e o resto do texto. Cada falha reduz o número de "c"s em um e tenta-se novamente, incluindo o caso de zero ocorrências.

Seção: Implementação alternativa de matchhere:

```
/* matchstar: leftmost longest search for c*regexp */
int matchstar(int c, char *regexp, char *text)
{
  char *t;
  for (t = text; *t != '\0' && (*t == c || c == '.'); t++)
  ;
  do { /* * matches zero or more */
    if (matchhere(regexp, t))
    return 1;
    } while (t-- > text);
  return 0;
}
```

Considere a expressão regular "(.*)", que casa qualquer texto arbitrário entre parênteses. Dado o texto alvo

```
for (t = text; *t != '\ 0' && (*t == c || c == '.'); t++)
```

um casamento maior logo no começo irá identificar a expressão inteira entre parênteses, enquanto um casamento menor iria parar ao encontrar o primeiro fechamento de parênteses. (É claro,

um casamento maior se começar da segunda abertura de parênteses também iria se extender até o fim do texto).

1.5 - Estendendo o Exemplo

O propósito de nosso livro era ensinar boas práticas de programação. Na época em que o livro foi escrito,Rob e eu estávamos no Bell Labs, assim não tínhamos experiência em primeira mão de como o livro seria usado em uma sala de aula. Foi gratificante descobrir que alguns dos materiais se sairam bem usados em aulas. Eu venho usado este código deste 2000 como um meio de ensinar importantes detalhes sobre programação.

Em primeiro lugar, ele mostra como recursão é útil e leva a código limpo de forma nova. Não é mais um exemplo de Quicksort (ou fatorial!), nem é algum tipo de código para percorrer uma árvore.

É também um bom exemplo de experimento com performance. Sua performance não é muito diferente das versões de sistema encontradas no "grep", o que mostra que a técnica recursiva não é muito cara e que não vale à pena tentar removê-la para deixar o código mais rápido.

Por outro lado, é também uma ótima demonstração da importância de um bom algoritmo. Se um padrão incluir muitos ".*"s, a implementação mais direa precisa ficar voltando pra trás muito e em alguns casos realmente irá executar lentamente. (O "grep" padrão do Unix tem a mesma propriedade). Por exemplo, o comando

```
grep 'a.*a.*a.*a'
```

leva cerca de 20 segundos para processar um arquivo de texto com 4MB em uma máquina típica. Uma implementação baseada em converter um autômato finito não-deerminístico para um automato determinístico, como em "egrep", terá uma performance muito melhor em casos difíceis—o mesmo padrão e a mesma entrada é processada em menos de um décimo de segundo., e no geral, o tempo de execução é independente do padrão.

Exensões para esa classe de expressões regulares pode formar o ponto de partida de uma série de exercícios. Por exemplo:

(1) Adicione outros metacaracteres, como "+" para uma ou mais ocorrências do caractere anterior, ou "?" para zero ou uma ocorrência. E alguma forma de escapar os metacaracteres para que eles possam ser usados como representação literal.

(2) Separe o processamento da expressão regular em uma fase de "compilação" e uma fase de "execução". A compilação converte a expressão regular em uma forma interna que faz com que o código de casamento seja mais simples e assim futuros casamentos executem mais rápido. Esta separação não é necessária para a classe simples de expressões regulares do projeto original, mas faz sentido em aplicações como "grep" onde a classe é mais rica e a mesma expressão regular é usada em um grande número de linhas de entrada.

(3) Adicione classes de caractere como [abc] e [0-9], as quais na notação convencional de "grep" casam "a" ou "b" ou "c" e um dígito respectivamente. Isso pode ser feito de muitas formas, a mais natural seria substituir os `char *`s do código original por uma estrutura:

Seção: Estrutura de Token de Expressão Regular:

```
typedef struct RE {
  int    type;   /* CHAR, STAR, etc. */
  char   ch;     /* the character itself */
  char   *ccl;   /* for [...] instead */
  int    nccl;   /* true if class is negated [^...] */
} RE;
```

e modificar o código básico para lidar com uma cadeia destas estruturas ao invés de uma cadeia de caracteres. Não é estritamente necessário separar a compilação da execução para esta situação, mas fazendo isso, a tarefa fica mais fácil. Estudantes que seguem o conselho de pré-compilar em tais estruturas invariavelmente se saem melhor do que aqueles que tentam interpretar algum padrão complicado de estrutura de dados na hora da execução.

Escrever especificações claras e sem ambiguidades para classes de caracteres é difícil, e implementá-las perfeitamente é pior, requerendo um monte de código edioso e pouco instrutivo. Eu

simplifiquei este exercício ao longo do tempo, e hoje mais frequentemente peço para as abreviações de Perl como "d para dígito e "D para não-dígito ao invés da notação de intervalos entre colchetes.

(4) Use algum tipo opaco para esconder a estrutura `RE` e todos os detalhes de implementação. Esta é uma boa forma de mostrar programação orientada à objetos em C, que não suporta muito mais além dissos. De fato, na prática usa-se uma classe para expressões regulares mas com nomes de funções tais como `RE_new()` e `RE_match()` para os métodos ao invés do açúcar sintático de linguagens orientadas à objeto.

(5) Modifique a classe de expressões regulares para funcionarem como os asteriscos de vários shells: os casamentos precisam ocorrer em todo o exto e "*" casa com qualquer número de caracteres enquanto "?" casa com um único caractere. Pode-se modificar o algoritmo ou mapear a entrada no algoritmo existente.

(6) Converta o código para Java. O código original usa ponteiros de C muito bem, mas é uma boa prática encontrar alternativas em uma linguagem diferente. As versões em java usam ou `String.charAt` (indexando ao invés de usar ponteiros) ou `String.substring` (mais próxima da versão com ponteiros). Nenhuma delas é tão clara como o código em C, e nenhuma é tão compacta. Embora performance não seja mesmo parte do exercício, é interessante notar que a implementação em Java executacerca de seis ou sete vezes mais devagar que as versões em C.

(7) Escreva uma classe que encapsule tudo e converta internamente de expressões regulares para as classes de casamento de padrão do Java, as quais separam a compilação e o casamento de forma bem diferente. Este é um bom exemplo do padrão Adaptador, que dá um rosto diferente para uma classe ou conjunto de funções existente.

Eu também usei este código extensivamente para explorar técnicas de teste de código. Expressões regulares são ricas o bastante para que testá-las não seja trivial, mas pequenas o bastante para que uma pessoa possa rapidamente escrever uma coleção substancial de testes a serem feitos automaticamente. Para as extensões listadas acima, eu peço para que estudantes escrevam um grande número de testes em uma linguagem compacta (outro exemplo de "notação") e usem estes testes em seu próprio código; naturalmente eu uso os testes deles nos códigos de outros estudantes também.

1.6 - Conclusões

Eu fui surpreendido em quão compacto e elegante este código era quando Rob Pike escreveu ele pela primeira vez—era muito menor e mais poderoso que eu achei ser possível. Pensando sobre isso, pode-se ver um número de razões so por quê este código é tão pequeno.

Primeiro, as funcionalidades foram bem escolhidas para serem as mais úteis e darem o maior destaque na implementação, sem qualquer obstáculo. Por exemplo, a implementação dos padrões "$\hat{}$" e "$" precisou de apenas 3 ou 4 linhas, mas mostram como lidar com casos especiais de maneira limpa antes de lidar com casos gerais de maneira uniforme. A operação de fechamento * é uma noção fundamental de expressões regulares e fornece a única forma de lidar com padrões de tamanhos desconhecidos, então tinha que estar presente. Mas adicionar coisas como "+" e "?" não acrescentaria nenhum grande ensinamento, então estas coisas são deixadas como exercícios.

Segundo, recursão vale à pena. Esta técnica de programação fundamental quase sempre leva á código menor, mais limpo e mais elegante que o escrito por meio de laços explícitos e este foi o caso aqui. A ideia de ir removendo um caractere do texto e um da expressão e ir percorrendo os demais recursivamente imita a estrutura recursiva do tradicional fatorial ou do tamanho de string, mas em uma maneira muito mais interessante e útil.

Rob me disse que a recursão não era tanto uma decisão explícita de projeto, mas uma consequência de como ele lidou com o problema: dado um padrão e um texto, escreva uma função que procure um casamento; o que por sua vez gerou uma função `matchhere`; e assim por diante.

"Eu tenho memórias vívidas de assistir o código quase se escrever assim sozinho. O único desafio era obte as condições limite adequadas para se sair da recursão. Colocado de outro modo, a recursão não é apenas o método de implementação, é também um reflexo do processo de pensamento que tomei ao escrever o código, o qual foi em parte responsável pela simplicidade do código. o mais importante, talvez, é que eu não tinha um projeto quando comecei, eu apenas iniciei a programar e vi o que se desenvolveu. De repente, estava feito."

Terceiro, este código realmente usa as estruturas da linguagem efetivamente. Ponteiros podem ser abusados, é claro, mas aqui eles são usados para criar expressões compactas que naturalmente expressam a extração de caracteres individuais e o avanço para o próximo caractere. O mesmo efeito pode ser obtido usando os índices de vetores e substrings, mas neste código ponteiros fazem um trabalho melhor, especialmente quando unidos às estruturas da linguagem X para auto-incremento e conversão implícita de valores verdade.

Eu não conheço qualquer outro pedaçõ de código que faz tanto em tão poucas linhas ao mesmo tempo em que fornece uma rica fonte de ensinamento e ideias.