Flex – Mini Manual

Gustavo Lermen
Adaptado de
http://www.gnu.org/software/flex/manual/

Introdução

Flex é um gerador de analisadores léxicos (também chamados de *scanners*). Um analisador léxico é um programa que reconhece padrões léxicos em um texto.
Para gerar o analisador léxico, um arquivo contendo a especificação dos padrões a serem reconhecidos deve ser passado como entrada para o Flex. Caso nenhum arquivo seja passado, a entrada padrão é lida.
A especificação do analisador léxico a ser gerado é dada na forma de pares contendo expressões regulares e a ação a ser executada. Estes pares também são chamados de "regras". Neste sentido, o que o Flex faz é criar um programa (código C ou C++) que ao reconhecer uma expressão regular contida na especificação, executa a ação correspondente.
Este programa gerado pode então ser compilado com qualquer compilador C ou C++ de modo a gerar um programa executável.

Formato do Arquivo de Entrada

O arquivo de entrada utilizado pelo Flex é formado por três seções. Cada seção é separada uma da outra através de uma linha contendo somente '%%'.

```
definições
%%
regras
%%
código de usuário
```

A seção de definições contém apenas definições de nomes que visam simplificar a especificação do analisador. Nesta seção também são definidas as *start conditions* que serão explicadas mais adiante.

Estas definições possuem o seguinte formato:

```
nome definição
```

onde nome é uma palavra iniciando com uma letra ou um *underscore* ('_'), seguido por zero ou mais letras, digitos, '_', ou '-'. A definição é iniciada no primeiro caractere diferente de espaço após o nome e segue até o final da linha. Esta definição pode posteriormente ser utilizada da seguinte forma: {nome}. Quando um nome é encontrado neste formato, ele é substituído por sua definição. Por exemplo, sejam as seguintes definições:

```
DIGITO    [0-9]
ID        [a-z][a-z,0-9]*
```

DIGITO neste caso é definido como uma expressão regular que casa com um único dígito. Da mesma maneira, ID é uma expressão regular que casa com uma letra seguida por zero ou mais letras ou dígitos. Neste sentido, uma referência a estas definições como esta,

```
{DIGITO}+"."{DIGITO}*
```

é equivalente a esta:

```
([0-9])+"."([0-9])*
```

e casa com um digito, seguido por um ponto, seguido por zero ou mais digitos.

Qualquer texto que esteja contido em um bloco definido por '%{' e '%}' é copiado para a saída. Desta maneira um código em C pode ser colocado dentro de um bloco destes de modo a ser copiado para o analisador gerado.

Qualquer texto que esteja dentro de um bloco %top é copiado para o inicio da saída antes de qualquer outra definição. Desta maneira o seguinte código:

```
%top{
        #include <stdint.h>
        #include <inttypes.h>
    }
```

é copiado para o início da saída.

A seção de regras contém regras no seguinte formato:

```
padrão    ação
```

onde o padrão não deve ser identado e a ação deve começar na mesma linha do padrão. Toda a vez que o padrão (que é especificado através de uma expressão regular) é encontrado, a respectiva ação é executada.

A seção de código de usuário é copiada para a saída. Nesta seção são definidas funções especificas do usuário que podem ser utilizadas nas ações.

Exemplos Simples

A seguinte especificação cria um analisador que ao encontrar o texto ‘username’ o substitui pelo nome do usuário.

```
%%
username  printf("%s\n",getlogin());
```

Por padrão, qualquer texto que não seja reconhecido pelo analisador é copiado para a saída. Desta maneira o analisador gerado através da especificação acima irá copiar toda a entrada para a saída, exceto quando a palavra username for encontrada. Neste caso a palavra username será substituída pelo nome do usuário. O símbolo ‘%%’ marca o inicio da seção de regras.

Abaixo outro exemplo simples:

```
int num_lines = 0, num_chars = 0;
%%
\n   ++num_lines; ++ num_chars;
.    ++num_chars;
%%
main() {
     yylex();
     printf("#  de linhas = %d\n", num_lines);
     printf("# de caracteres = %d\n", num_chars);
}
```

Neste exemplo o analisador gerado conta o número de caracteres e o número de linhas da sua entrada. A primeira linha declara duas variáveis globais (num_lines e num_chars) que terão

seu escopo válido tanto dentro da função yylex quanto da função main (declarada na seção de código de usuário). Existem apenas duas regras neste analisador: uma que reconhece uma nova linha (‘\n’) e outra que reconhece qualquer caractere. Ao ser encontrado o caractere de nova linha, ambos os contadores são incrementados. Ao ser encontrado um caractere apenas o contador de caracteres é incrementado.

Padrões

Os padrões utilizados pelo Flex são definidos a partir de um conjunto estendido de expressões regulares. Alguns exemplos são mostrados na tabela abaixo.

Padrão	Significado
‘x’	Reconhece o caractere ‘x’.
‘.’	Reconhece qualquer caractere. Exceto o caractere de nova linha.
‘[xyz]’	Reconhece uma classe de caracteres. Neste exemplo, um ‘x’ ou um ‘y’ ou um ‘z’ serão reconhecidos.
‘[abj-oZ]’	Reconhece uma classe de caracteres com intervalo. Neste exemplo, um ‘a’, ou um ‘b’, ou qualquer caractere entre ‘j’ e ‘o’, ou um ‘z’ serão reconhecidos.
‘[^A-Z\n]’	Reconhece uma classe de caracteres negada. Neste exemplo o analisador reconhece qualquer caractere que não seja maiúsculo ou nova linha.
‘r*’	Reconhece zero ou mais caracteres ‘r’
‘r+’	Reconhece um ou mais caracteres ‘r’
‘r?’	Reconhece zero ou um caractere ‘r’
‘r{2,5}’	Reconhece de 2 a 5 caracteres ‘r’
‘r{2,}’	Reconhece dois ou mais caracteres ‘r’
‘r{4}’	Reconhece exatamente 4 caracteres ‘r’
‘{nome}’	Reconhece a expansão da definição ‘nome’
‘ ”[xyz]\foo” ‘	Reconhece a seqüência de caracteres ‘[xyz]\foo’
‘\X’	Se X é um ‘a’, ‘b’, ‘f’, ‘n’, ‘r’, ‘t’ ou ‘v’, a representação ANSI-C do respectivo caractere é reconhecida. Caso contrário o literal ‘X’ é reconhecido. A \ é utilizada para “escapar” operadores, como por exemplo, o ‘*’.
‘(r)’	Reconhece um ‘r’. Os parênteses são utilizados para alterar a prededência.
‘rs’	Reconhece a expressão regular ‘r’ seguida da expressão regular ‘s’. Concatenação.

‘r|s’	Reconhece ou um ‘r’ ou um ‘s’
‘^r’	Reconhece um ‘r’ no início de uma linha.
‘r$’	Reconhece um ‘r’ no final de uma linha. É equivalente a ‘r/\n’.
‘<r>r’	Reconhece um ‘r’ somente quando o analisador estiver no estado <s>.
‘<s1,s2,s3>r’	Reconhece um ‘r’ quando o analisador estiver no estado s1, s2 ou s3.

Dentro de uma classe de caracteres, todos os operadores utilizados nas expressões regulares perdem sua função exceto pela ‘\’ e os próprios operadores utilizados na classe ‘-‘, ‘[]’ e, quando no início da definição da classe de caracteres o ‘^’.

Além de caracteres e intervalos de caracteres, classes de caracteres podem conter também expressões de classe de caracteres. Estas expressões são delimitadas por: ‘[:’ e ‘:]’. As expressões aceitas são mostradas na figura abaixo.

```
[:alnum:] [:alpha:] [:blank:]
[:cntrl:] [:digit:] [:graph:]
[:lower:] [:print:] [:punct:]
[:space:] [:upper:] [:xdigit:]
```

Todas estas expressões definem um conjunto de caracteres equivalente a função isXXX da linguagem C padrão. Por exemplo, a expressão `[:alnum:]` define o conjunto de caracteres para os quais a função `isalnum()` retorna verdadeiro (neste caso qualquer caractere alfanumérico). Desta maneira, as seguintes classes de caracteres são todas equivalentes:

```
[[:alnum:]]
[[:alpha:][:digit:]]
[[:alpha:][0-9]]
[a-zA-Z0-9]
```

Como a entrada é reconhecida

Quando o analisador léxico gerado é executado, ele analisa sua entrada a procura de cadeias de caracteres que satisfaçam os padrões especificados. Caso uma cadeia satisfaça mais de uma expressão, a ação correspondente à expressão que consumir o maior número de caracteres da entrada é executada. Caso ambas consumam o mesmo número de caracteres, a expressão que estiver listada primeiro é a escolhida.

Uma vez que a entrada é reconhecida, o texto consumido é disponibilizado através da variável global (ponteiro global) `yytext` e o tamanho da entrada é disponibilizado para o usuário através da variável global `yyleng`. A ação correspondente é então executada e o restante da entrada é lida a procura de mais cadeias que satisfaçam alguma das regras.

Caso nenhum padrão seja reconhecido, a regra padrão é executada: o próximo caractere da entrada é copiado para a saída. Desta maneira, o arquivo Flex mais simples possível possui o seguinte formato:

```
%%
```

Este arquivo gera um analisador que apenas copia a entrada para a saída (caractere por caractere).

Ações

Cada padrão em uma regra possui uma ação correspondente. Estas ações podem ser compostas por qualquer código C. Por definição o padrão termina no primeiro caractere de espaço após a sua especificação. O restante da linha é considerado ação. Se a ação é vazia o padrão reconhecido é descartado. Por exemplo, a seguinte especificação gera um analisador que elimina todas as ocorrências de "`zap me`"

```
%%
"zap me"
```

Todos os outros caracteres serão copiados para a saída (regra padrão). A seguinte especificação comprime múltiplos espaços em branco para apenas um espaço em branco e descarta qualquer espaço em branco no final de uma linha.

```
%%
[ \t]+    putchar(' ');
[ \t]+$   /*ignora este token*/
```

Caso uma ação contenha um `{`, esta se estende até o `}` correspondente. Neste caso uma ação pode ocupar mais de uma linha. Uma ação contendo apenas uma barra vertical (|) significa que a ação a ser executada é a ação da regra seguinte.

Ações podem conter qualquer tipo de código C, inclusive uma instrução `return` que pode retornar um valor para qualquer função que tenha chamado a função `yylex()`[1]. Cada vez que a função `yylex()` é chamada o processamento é continuado a partir do ponto em que a instrução `return` foi executada ou o final do arquivo encontrado.

As seguintes diretivas podem ser incluídas em ações:

ECHO	Copia o conteúdo de yytext para a saída.
BEGIN	Seguida por um nome, coloca o analisador no estado correspondente.
REJECT	Faz com que o analisador procure pela "segunda melhor" regra que satisfaça o padrão.

A seguinte especificação mostra a utilização da diretiva REJECT.

```
int word_count = 0;
```

[1] Função que executa o analisador léxico

```
%%
frob      special(); REJECT;
[^ \t\n]+ ++word_count;
```

O analisador gerado por esta especificação conta todas as palavras contidas na estrada e ainda executa a rotina `special()` toda vez que a palavra `frob` é encontrada. Sem a utilização da diretiva REJECT, qualquer ocorrência da palavra `frob` encontrada não seria contata pois o analisador normalmente executa apenas uma ação por token reconhecido.

Funções disponíveis para o usuário

A tabela a seguir sumariza as diferentes variáveis disponibilizadas para o usuário que podem ser utilizadas dentro das ações.

`char* yytext`	Armazena o texto correspondente ao padrão reconhecido.
`int yyleng`	Armazena o tamanho to texto correspondente ao padrão reconhecido
`FILE* yyin`	Ponteiro a partir de onde o Flex irá ler sua entrada (stdin por padrão).
`FILE* yyout`	Ponteiro para a saída (stdout por padrão).
`YY_START`	Retorna um valor do tipo inteiro que corresponde a *start condition* corrente. Pode ser utilizado com BEGIN para retornar o analisador para usa condição inicial.

Condições

Através da utilização de condições é possível realizar a ativação de regras de maneira condicional. Qualquer regra que possua um padrão precedido de `<sc>` somente será ativada quando o analisador estiver na condição chamada `sc`. A seguinte regra:

```
<STRING>[^"]*  { /*elimina o corpo de uma string
...
}
```

será ativada somente quando o analisador estiver na condição `STRING`. Da mesma maneira, a seguinte regra:

```
<INITIAL,STRING,QUOTE>\. {
...
}
```

será ativada somente quando o analisador estiver na condição INITIAL, STRING ou QUOTE.

Condições são declaradas na seção de definição (primeira seção) do arquivo de especificação. Declarações de condições devem ser feitas em linhas não identadas iniciando as com `%x` ou `%s` seguido de uma lista de nomes. Uma declaração de condição iniciando com `%x` define uma condição exclusiva. Uma declaração de condição iniciando com `%s` define uma condição inclusiva. Uma condição é ativada através da ação BEGIN. Até que o próximo BEGIN seja executado, regras pertencentes à condição corrente estarão ativadas e regras não pertencente a esta condição estarão desativadas. Se a condição é inclusiva, então regras que não possuem condição nenhuma (condição INITIAL) também estarão ativas. Caso a condição seja exclusiva, regras sem condição nenhuma serão desativadas. Um conjunto de regras dependentes da mesma condição exclusiva descreve um analisador independente de qualquer outra regra definida na especificação do analisador que não pertença a esta condição. Devido a isso, condições exclusivas tornam fácil a construção de mini-analisadores que analisam porções da entrada que venham a ser sintaticamente diferentes do resto como, por exemplo, comentários.

O seguinte exemplo:

```
%s exemplo
%%
<exemplo>foo   alguma_coisa();
bar            outra_coisa();
```

é equivalente à:

```
%x exemplo
%%
<exemplo>foo                    alguma_coisa();
<INITIAL,exemplo>bar            outra_coisa();
```

Sem o qualificador **`<INITIAL,exemplo>`** o padrão `bar` do segundo exemplo não seria ativado (mesmo que ele fosse reconhecido, sua regra não seria executada) quando o analisador estivesse na condição **`exemplo.`** Da mesma maneira, se apenas o qualificador **`<exemplo>`** tivesse sido utilizado a regra não seria executada quando o analisador estivesse na condição inicial. No primeiro exemplo, devido à condição ter sido declarada como inclusiva (`%s`), a regra é executada em qualquer condição (`INITIAL` ou `exemplo`).

A diretiva `BEGIN(0)` faz o analisador retornar a sua condição inicial, onde somente regras sem nenhuma condição são ativadas. Esta condição pode ser também referenciada como `INITIAL`. Desta maneira, `BEGIN(0)` é equivalente à `BEGIN(INITIAL)`.

A fim de ilustrar o uso de condições o seguinte exemplo demonstra duas interpretações diferentes para seqüências de caracteres do tipo ‘123.456’. Por padrão o esta seqüência de caracteres é tratada como três tokens diferentes, o inteiro 123, o ponto e o inteiro 456. Caso esta seqüência seja precedida pela palavra ‘expect-floats’ o analisador irá trata-la como uma única seqüência, reconhecendo desta maneira o número 123.456 .

```
%{
      Include <math.h>
%}
%s expect
%%
expect-floats           BEGIN(expect);
<expect>[0-9]+{.}[0-9]+ { printf ("Encontrado um float: %f\n", atof(yytext)); }
<expect>\n              { BEGIN(INITIAL);}
[0-9]+                  {printf("Encontrado um número inteiro: %d\n", atoi(yytext)); }
.                       printf("Encontrado um ponto\n");
```

As condições nada mais são do que números inteiros. Desta maneira a condição corrente pode ser obtida através da macro YY_START.

Gerando o analisador

A fim de gerar o analisador léxico alguns passos devem ser executados. Supondo que minha especificação esteja no arquivo especificação.l e que eu esteja usando o compilador gcc, os comandos a seguir devem ser executados:

```
> flex especificação.l –o analisador.c
> gcc analisador.c –o analisador
```

Exemplos

A seguir serão apresentados alguns exemplos de especificações de analisadores léxicos. O primeiro exemplo gera um analisador que conta o número de linhas e o número de caracteres da entrada.

```
%option noyywrap

%{
      #include <stdio.h>
      #include <time.h>

      int numlines = 0;
      int numchars = 0;
%}

%%

\n    numlines++; numchars++;
.     numchars++;

%%
```

```
int main(int argc, char *argv[]){
      yyin = fopen(argv[1], "r");
      yylex();
      fclose(yyin);
      printf("Número de caracteres: %d\n",
      numchars);
      printf("Número de linhas: %d\n", numlines);
      return 0;
}
```

O analisador gerado por este arquivo de especificação espera um arquivo como entrada. Este arquivo é então aberto e a entrada padrão do Flex (`yyin`) é redirecionada para este arquivo. Caso quiséssemos que a saída fosse redirecionada para um arquivo poderíamos utilizar a mesma abordagem com o ponteiro `yyout`. A opção especificada na primeira linha do arquivo indica que o analisador gerado não irá tratar múltiplos arquivos de entrada. Mais informações a respeito desta e também de outras opões podem ser encontradas no manual do Flex.

O segundo exemplo gera um analisador que toda vez que encontra a string `<date>` dentro de um arquivo, substitui a mesma pela data corrente. Este exemplo utiliza à mesma abordagem do exemplo anterior onde à entrada padrão do Flex é redirecionada para um arquivo passado como parâmetro para o programa.

```
%option noyywrap

%{

      #include <stdio.h>
      #include <time.h>

%}

%%
"<date>" {time_t t;
        time(&t);
        printf("%s", ctime(&t));}
%%

int main(int argc, char *argv[]){
      yyin = fopen(argv[1], "r");
      yylex();
      fclose(yyin);
      return 0;
}
```

Caso o seguinte arquivo seja passado como parâmetro de entrada:

```
A data de hoje é <date> .
```

A seguinte será gerada:

```
A data de hoje é Wed Jul 11 23:20:45 .
```

O terceiro exemplo apresenta uma especificação que gera um analisador que reconhece um subconjunto da linguagem de programação Pascal. Este exemplo faz uso de definições de modo a simplificar a construção de expressões regulares.

```
%option noyywrap

%{
      #include <math.h>
%}


DIGIT [0-9]
ID    [a-z][a-z0-9]*


%%

{DIGIT}+ { printf("Numero inteiro encontrado: %s (%d)\n", yytext,
atoi(yytext));}

{DIGIT}"."{DIGIT}* {printf("Numero float encontrado: %s (%f)\n",
yytext, atof(yytext));}

if|then|begin|procedure|function    {
            printf("Palavra reservada encontrada: %s\n ", yytext);}

{ID}  {printf("Identificador encontrado: %s\n", yytext);}

"+"|"-"|"*"|"/" {printf("Operador encontrado: %s\n", yytext);}

"{"[\^{}}\n]*"}"

[ \t\n]+

.     printf("Caractere nao reconhecido: %s\n", yytext);

%%

int main(int argc, char *argv[]){
      yyin = fopen(argv[1], "r");
      yylex();
      fclose(yyin);
      return 0;
}
```

Interface com Bison

O Flex é muito utilizado em conjunto com o gerador de analisadores sintáticos Bison. Para analisar uma entrada o analisador sintático gerado pelo Bison faz uso da rotina `yylex()`. Esta rotina é utilizada pelo analisador sintático para que este obtenha o próximo token a ser tratado. O valor deste token é normalmente colocado na variável global `yyval`. Para utilizar o Flex em conjunto com o Bison, a opção `-d` deve ser utilizada no Bison. Deste modo um arquivo chamado `y.yab.h` é gerado contendo as definições de todos os tokens (definidos pela seção `%token` no Bison). Este arquivo deve então ser incluído no arquivo de especificação do Flex. O seguinte exemplo mostra esta integração:

```
%{
include "y.tab.h"
}%
%%
```

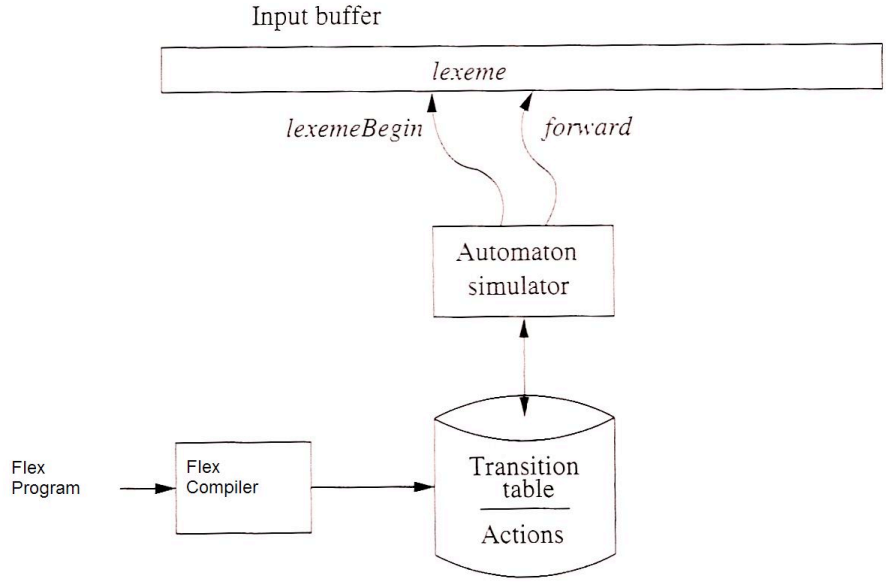
```
[0-9]      {yylval = atoi(yytext);
            return TOK_NUMERO;}
```

Neste exemplo, a definição de TOK_NUMERO está no arquivo `y.tab.h`.

Estrutura de um analisador léxico gerado pelo Flex[2]

Um analisador léxico gerado pelo Flex possui a estrutura mostrada na figura abaixo. A análise léxica é realizada por um programa fixo que simula a execução de um autômato. O restante do analisador é gerado pelo Flex quando ele lê o arquivo de especificação.

Um programa Flex é convertido em uma tabela de transições que é utilizada pelo programa que simula o autômato.



Neste esquema o programa simulador do autômato lê a entrada e caso algum padrão seja reconhecido, a ação correspondente é executada.

[2] Adaptado de *Compilers – Principles Techniques and Tools*