Flex
Um Tutorial

Aulas

Compiladores

O que é o Flex?

Flex é uma ferramenta para geração automática de analisadores léxicos (*scanners*), isto é, programas que reconhecem padrões léxicos num texto. O Flex é uma evolução da ferramenta Lex sendo mais rápido (Fast Lex). Lex foi desenvolvido por *M. E. Lesk* e *E. Shmidt* (Bell Laboratories - AT&T) enquanto que o Flex é um produto da Free Software Foundation, Inc. Como são comumente distribuídos em sistemas Unix sua documentação se encontra na forma de *manual pages* para as entradas **lex**, **flex** e **flexdoc**.

Ao invés do programador escrever manualmente um programa que realize a identificação de padrões numa entrada, o uso do Flex/Lex permite que sejam apenas especificados os padrões desejados e as ações necessárias para processá-los.

Para que Flex/Lex reconheçam padrões no texto, tais padrões devem ser descritos através de expressões regulares.

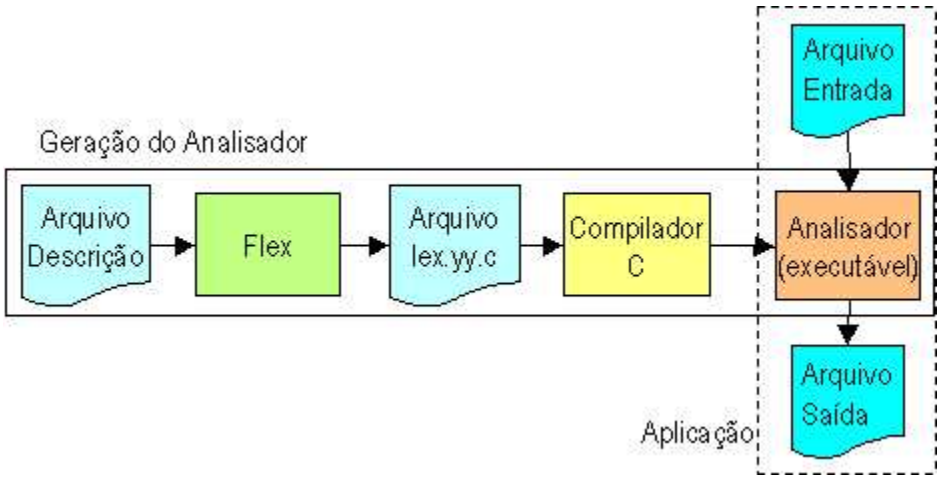
Como o Flex funciona?

Flex lê os arquivos de entrada especificados, ou a entrada padrão se nenhum arquivo for especificado, obtendo assim uma descrição do *scanner* a ser gerado. Este arquivo de entrada é o que chamamos arquivo de definição ou arquivo de descrição.

A descrição é realizada na forma de pares de expressões regulares e código C. Tais pares são denominados regras. As regras definem simultaneamente quais padrões devem ser procurados e quais as ações que devem ser executada quando da identificação deste padrão, ou seja, para cada padrão desejado pode ser associado um conjunto de ações escritas sob a forma de código C.

Flex gera como saída um arquivo fonte em linguagem C, cujo nome é `lex.yy.c`, no qual é definida a função `yylex()` e as variáveis global `yytext` e `yyleng`. A função `yylex()` é na verdade o analisador léxico gerado pelo arquivo de definição através do Flex. A variável global `yytext` contêm o texto do padrão reconhecido (uma *string*) no momento enquanto `yyleng` contêm o número de caracteres de tal *string*, podendo ambas serem usadas no trechos de código C que definem as ações associadas a cada padrão.

A seguir temos uma ilustração que indica os arquivos e etapas necessárias para produção de um *scanner* através da ferramenta Flex.



O arquivo `lex.yy.c` pode ser compilado para produzir um executável ou pode ser combinado com outros arquivo ou ainda modificado para se integrar com outros sistemas.

Quando executado, o programa gerado é capaz analisar uma cadeia de caracteres recebida como sua entrada

buscando ocorrências dos padrões especificados no arquivo de descrição. Quando um dos padrões é encontrados a varíavel `yytext` passa a apontar para a *string* do padrão encontrado e o comprimento da *string* é armazenado em `yyleng`. Após isto o programa gerado passa a executar as ações especificadas pelo código C associado a cada padrão.

Se nenhum padrão é encontrado então é tomada a ação *default* que é copiar o caractere para a saída. A entrada é varrida caractere a caractere até seu final, quando a função `yylex` retorna zero.

Se necessário a função `yylex` pode ser acionada outras vezes mas tal ação só será efetiva se uma chamada a função `yyrestart(FILE *f)` for executada informando um ponteiro válido para um arquivo de entrada.

Estrutura do arquivo de descrição

Todo arquivo de descrição do Flex possui três seções separadas por uma linha com apenas os caracteres `%%` colocados em seu início, como esquematicamente ilustrado a seguir:

```
DEFINIÇÕES
%%
REGRAS
%%
CÓDIGO
```

Seção DEFINIÇÕES

Como esperado a seção `DEFINIÇÕES` possui definições léxicas. É possível no entanto que esta seção permaneça vazia, isto é, sem definções. Toda *definição léxica* tem a forma:

nome definição

Como **nome** devemos usar uma palavra, iniciada por letra ou *underscore* (`'_'`) seguida de uma ou mais letras, dígitos, *underscore* ou traços (`'-'`). A **definição** segue o nome e se inicia no primeiro caractere não branco continuando até o final da linha. Usualmente as definições são conjuntos de caracteres ou expressões regulares contidas entre colchetes.

A seção `DEFINIÇÕES` pode ainda conter a declaração e inicialização de variáveis globais que poderão ser utilizadas nas ações e no código fornecido pelo programador.

Seção REGRAS

A seção `REGRAS` possui por sua vez as regras do analisador léxico a ser construído. Esta seção sempre possui regras, pois sem estas o analisador gerado apenas copia sua entrada para a saída.

Uma *regra* tem sempre a forma:

padrão ação

O **padrão** deve começar na primeira coluna do texto e são definidos como expressões regulares extendidas. A **ação** deve sempre ser iniciada na mesma linha e usualmente corresponde a um trecho de código C que será executado toda vez que o padrão for reconhecido na entrada.

Tanto na seção de `DEFINIÇÕES` como de `REGRAS` qualquer texto identado ou entre `%{` e `%}` é copiado literalmente para a saída (sem os caracteres `%{` e `%}`). Para o reconhecimento dos caracteres `{`, `}` e `%` estes devem aparecer não identados como padrões em separado. Em adição, apenas na seção de `DEFINIÇÕES` qualquer texto não identado é copiado literalmente para a saída.

Sob certos aspectos os caracteres `%%` indicam o início e o fim da seção `REGRAS`.

Seção CÓDIGO

A última seção, `CÓDIGO`, contêm todo o código C definido e fornecido pelo programador. Nesta seção usualmente temos declarada uma função `main()`, que define o início do programa que pode efetuar uma chamada a função `yylex()` (o *scanner* gerado por Flex). Outras funções, utilizadas nas ações definidas pelas regras, podem ser colocadas aqui. Esta seção também pode permanecer vazia, ou seja, sem código definido pelo programador.

Recomenda-se ainda a declaração de uma função `yywrap()` como abaixo:

```
int yywrap() {
 return 1;
}
```

Um exemplo básico

Um arquivo de descrição simples poderia ser como descrito a seguir:

```
        /* separe a primeira definicao por um tab */
        int     numeroDeLinhas=0,
                numeroDeCaracteres=0;

%%
\n      {
        /* incrementa numero de linhas */
        ++numeroDeLinhas;
        /* incrementa numero de caracteres */
        ++numeroDeCaracteres;
        }

.       {
        /* incrementa numero de caracteres */
        ++numeroDeCaracteres;
        }
%%
/* recomendavel declarar sempre
   funcao yywrap() */
int yywrap ();

/* programa principal */
main() {
  yylex(); /* scanner gerado por Flex */
  printf("Numero de Linhas = %d\n", numeroDeLinhas);
  printf("Numero de Caracteres = %d\n",numeroDeCaracteres);
}

int yywrap() {
 return 1;
}
```

Este *scanner* conta o número de caracteres e o número de linhas da entrada fornecida, produzindo como saída apenas duas mensagens informando o valor dos contadores de caracteres e linhas.

Neste exemplo a seção DEFINIÇÕES contêm apenas a declaração de duas variáveis globais `numeroDeLinhas` e `numeroDeCaracteres`, ambas acessíveis para a função `yylex()` que será construída pelo Flex e para `main()` declarado na seção CÓDIGO

Na seção REGRAS temos a declaração de duas regras:

- Uma para o caractere "\n" que incrementa tanto o número de linhas como o número de caracteres lidos.
- outra para os demais caracteres, indicada pelo caractere "." que incrementa o número de caracteres lidos. Note que o caractere "." tem o comportamente de uma cláusula *default* de uma diretiva *switch* da linguagem C.

Gerando um *scanner*

Para gerarmos um analisador léxico (*scanner*) devemos seguir os passos seguintes num ambiente Unix:

1. Executar o Flex para o arquivo de definição (supondo que seu nome seja `lex01.l`):

   ```
   $ flex lex01.l
   ```

   Isto produzirá um arquivo fonte em linguagem C que equivale ao *scanner* desejado. O arquivo C produzido tem sempre o nome `lex.yy.c`.

2. Compilar o código C gerado pelo Flex:

   ```
   $ cc lex.yy.c -o lex01
   ```

   O parâmetro `-o lex01` indica o nome do arquivo executável desejado. Sua omissão faz que o executável

seja produzido com o nome *default* `a.out`.

3. Executar o *scanner* gerado:

```
$ ./lex01
```

Ao executarmos o programa este fica aguardando uma entrada manual. Podemos assim digitar qualquer texto em uma ou mais linhas. O *scanner* não produz qualquer saída intermediária, isto é, não produz qualquer espécie de mensagem enquanto esta recebendo sua entrada. Podemos finalizar a entrada teclando `Ctrl+D` o que produz uma mensagem indicando o numero de linhas e caracteres fornecidos.

Um outro exemplo

Observemos o arquivo de definição a seguir:

```
        int subs=0;
%%
username        {
                printf("%s", getlogin());
                ++subs;
                }

%%

int yywrap();

main() {
  yylex();
  if (subs>0)
    printf("%d substituicoes realizadas.\n",subs);
}

int yywrap() {
  return 1;
}
```

Nele temos a definição de uma variável global `subs` que será destinada a contabilizar a quantidade de substituições realizadas pelo *scanner* (seção DEFINIÇÕES).

Na seção REGRAS temos a declaração de uma única regra, a qual reconhece a palavra `username`, cujas ocorrências serão contabilizadas e substituídas pelo nome do usuário obtido através da chamada de uma função da API do Unix. Como não existem outras regras nem mesmo uma para "." (todas as demais ocorrências), então toda a entrada que não corresponder a palavra `username` será transcrita *verbatim* para saída, isto é, exatamente como figurou na entrada.

Na seção CÓDIGO temos a declaração de uma função `main()` que aciona o *scanner* gerado por Flex e indica ao finao o número de substituições realizadas é indicado ao final da execução do programa.

Temos portanto um *scanner* capaz de substituir uma determinada ocorrência por outra, *convertendo* uma certa entrada para outra saída especificada. Isto poderia ser útil na substituição de sequências de caracteres por outros caracteres correspondendo a programas que realizam a filtragem da entrada.

Supondo que o arquivo de definição seja denominado `lex02.l` podemos gerar o *scanner* repetindo os passos indicados anteriormente (vide Gerando um *scanner*):

```
$ flex lex02.l
$ cc lex.yy.c -o lex02
```

Possuindo um arquivo `teste` o qual desejamos efetuar a substituição de `username` pelo nome do usuário corrente poderíamos escrever:

```
$ more teste | ./lex02.l
```

Ou

```
$ ./lex02 < teste
```

A saída do programa poderia igualmente ser redirecionada para um arquivo de nome `resultado`:

```
$ ./lex02 < teste > resultado
```

Mais um exemplo

Agora veremos um exemplo onde utilizamos expressões regulares para a definição de padrões que deverão ser reconhecidos pelo analisador léxico:

```
DIGIT            [0-9]
ID               [a-z][a-z0-9]*
WHITESPACE       [\t\n\r]

%%

{DIGIT}+         {
                 printf("Inteiro: %s\n", yytext);
                 }

{DIGIT}+"."{DIGIT}*      {
                 printf("Real: %s\n", yytext);
                 }

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

{WHITESPACE}+    /* Elimina espaços em branco */

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

int yywrap();

main() {
  yylex();
}

int yywrap() {
  return 1;
}
```

Neste arquivo de definição temos a declaração de definições léxicas ou invés de variáveis C:

```
DIGIT            [0-9]
ID               [a-z][a-z0-9]*
WHITESPACE       [\t\n\r]
```

Foram criadas três definições léxicas:

- DIGIT: que corresponde ao conjunto de caracteres de `'0'` até `'9'` que serão aceitos como dígitos.
- ID: que corresponde a sequências iniciadas por letras, seguidas de zero ou mais letras ou dígitos.
- WHITESPACE: que corresponde aos caracteres de tabulação, nova linha e retorno de carro.

Tais definições léxicas são usadas na seção REGRAS através de referências `{nome}`.

Imaginando que o arquivo de definição seja denominado `lex03.l` podemos gerar o *scanner* repetindo os passos indicados anteriormente (vide Gerando um *scanner*):

```
$ flex lex03.l
$ cc lex.yy.c -o lex03
```

Sua execução exibe uma mensagem indicando se a entrada é reconhecida como um valor inteiro, um valor real ou um identificador sendo que os demais caracteres são indicados como não reconhecidos.

Definição de Padrões

Os padrões de entrada utilizados nas seções DEFINIÇÕESe REGRAS são escritos através de uma notação ampliada das expressões regulares que são, da maior para menor precedência:

c	caractere c.

`.`	caractere exceto nova linha (`'\n'`).
`[abc]`	um caractere da classe de caracteres, no caso um `'a'` ou `'b'` ou `'c'`.
`[abj-oZ]`	caractere da classe de caracteres, no caso um `'a'` ou `'b'` ou qualquer caractere entre `'j'` e `'o'` ou `'Z'`.
`[^A-Z]`	um caractere que não esteja na classe das maiúsculas de `'A'` a `'Z'`.
`[^A-D\t]`	um caractere que não esteja na classe das maiúsculas de `'A'` a `'T'` ou tabulação `'\t'`.
`e*`	*zero* ou *mais* `e`, onde `e` é uma expressão regular.
`e+`	*um* ou *mais* `e`, onde `e` é uma expressão regular.
`e?`	*zero* ou *um* `e` (um `e` opcional) onde `e` é uma expressão regular.
`r{2,5}`	*dois* até *cinco* `e`, onde `e` é uma expressão regular.
`r{2,}`	*dois* até *mais* `e`, onde `e` é uma expressão regular.
`r{4}`	exatamente *quatro* `e`, onde `e` é uma expressão regular.
`{name}`	uma ocorrência da definição *name*.
`"[xy]\"oi"`	literal `[xy]"oi`.
`\x`	caractere especial se x é um a, b, f, n, r, t ou v.
`\0`	caractere `nul` (código ASCII zero).
`\123`	caractere cujo código octal é 123.
`\x2a`	caractere cujo código hexadecimal é 2a.
`(r)`	expressão regular `r` sendo que os parêntesis são usados para modificar precedência.
`rs`	concatenação das expressões regulares `r` e `s`.
`r | s`	expressão regular `r` ou `s`.
`r/s`	expressão regular `r` apenas se seguido por uma expressão regular `s`. A expressão `s` não faz parte do texto reconhecido, sendo denominada *trailing context*.
`^r`	expressão regular `r` apenas no início de uma nova linha (equivale a `<\n>r`).
`r$`	expressão regular `r` apenas no final de uma nova linha (equivale a `r/\n`).
`<s>r`	expressão regular `r` apenas se precedida pela expressão `s`.
`<s1,s2>r`	expressão regular `r` apenas se precedida pela expressão `s1` ou `s2`.
`<*>r`	expressão regular `r` em qualquer condição de início.
`<<EOF>>`	um *end-of-file* (fim de arquivo).

Considerações Adicionais

Note que a noção de uma nova linha ("new line") é exatamente o que o compilador C utilizado para compilar os arquivos gerados pelo Flex interpreta como um `'\n'`, ou seja, pode ser significativamente diferente entre sistemas particulares. Sistemas DOS ou Windows entendem uma nova linha como a sequência `'\r\n'` indicando que o caractere `'\r'` deve ser implicitamente filtrado da entrada ou explicitamente indicado pelas definições e regras. Por exemplo:

- Num sistema Unix `r/\n` ou `r$` indicam a expressão `r` no final de uma linha
- Num sistema DOS a expressão equivalente seria `r/\r\n`.

Dentro de classes de caracteres (conjunto de caracteres delimintados por colchetes) todos os operadores de expressões regulares perdem seu sentido exceto o caractere de *escape* `'\'`, o caractere traço `'-'` e apenas no início da definição da classe o caractere de negação `'^'`.

Prof. Peter Jandl Jr.