LEX

José João Almeida
José Bernardo Barros
Pedro Rangel Henriques

91/03/01

Contents

1 LEX

1.1 Genericamente

'E um gerador de programas (m'odulos) destinados a processamento lexico de textos.

O m'odulo gerado, pode funcionar como:
. Filtro para processamento l'exico
. Pattern Matching

. Reconhecedor de linguagens regulares simples
. Preprocessador simples
. Analisador l'exico [1] para liga,c ao a m'odulos de reconhecimento sint'atico

1.2 Funcionamento

Com base num texto "lex", 'e gerado um m'odulo C que contem um reconhecedor baseado em automatos deterministicos e que vai permitir encontrar "patterns" num texto de entrada e sobre eles executar uma ac,c ao semantica correspondente.

O m'odulo C 'e automaticamente escrito num ficheiro "lex.yy.c" que inclui uma fun,c ao "yylex()" , aqual executa o algoritmo que de seguida se apresenta:

```
FUNCAO yylex () : int
  REPETIR
  | yytext <- ""                 # yytext vai guardar o texto
  |                              # que correspondeu `a ExpRegular
  | c<- input()
  | ENQUANTO (c cabe numa ExpRegular)
  | | yytext <- concatenar ( yytext , c)
  | | c <- input()
  |
  | executar a accaoC correspondente a ExpRegular
  |
  ATE chegar ao fim input
  yywrap()
```

1.3 Estrutura dum texto LEX

```
<definicoes gerais>
"%%"
<regras de traducao>
"%%"
<rotinas C auxiliares>
```

Cada regra vai ter a seguinte estrutura:

```
<ExpRegular>  "{" <AccaoC> "}"
```

¡ExpRegular¿ vai definir uma por,c ao de texto sobre a qual pretendemos efectuar a ¡Ac,c aoC¿

[1] Reconhecedor dos s'imbolos terminais duma linguagem

1.4 Um exemplo simples

```
%%
km  printf("Kilometros");
%%
```

Este texto lex especifica um filtro para substituir as ocorrências de km por "Kilometros".

Por defeito todo o texto que n ao "encaixe" em nenhum pattern 'e simplesmente copiada para a sa'ida.

Neste exemplo tanto as ¡defini,c oes gerais¿ como as ¡rotinas C auxiliares¿ s ao inexistentes.

1.5 Utiliza,c ao

Para utilizar o gerador LEX devem executar-se as seguintes tarefas:
. criar source.l (criar texto lex)
. `lex source.l` (cria lex.yy.c)
. `cc lex.yy.c -ll` (cria programa executavel a.out)
. `a.out < entrada > saida` (executa o programa gerado produzindo "saida" por filtragem de "entrada")

1.5.1 Opções do comando LEX

```
-r   accoes em Fortran (RATFOR)

-t   o ficheiro lex.yy.c 'e enviado para standard output

-v   Fornece estat'isticas do automato gerado.
```

1.6 Funcionamento como preprocessador

Para devenvolver um preprocessador pode usar-se o m'odulo gerado atrav'es da principal corresponde ao seguinte algoritmo:

```
enquanto nao fim input
  yytext <- getSimbolo()
  efectuar accao correspondente ao tipo de simbolo encontrado
```

1.6.1 Exemplo: preprocessador para lisp

Pretende-se construir um preprocessador para uma linguagem tipo LISP que permita converter todas as invocaçoes de funçoes escritas na forma habitual f(x) para a forma (f x) usada em LISP. Pretende-se ainda que a defini,c ao de conjuntos `(makeset <elementos>)` possa ser feita atraves de `<elementos>` }

Conteudo do ficheiro em "lex":

```
$ cat pp.l

 ALFANUM ([a-zA-Z0-9])           /* definicoes de ExpReg     */
 ID ([a-zA-Z](\{ALFANUM\})*)
 E  ([{\bs}\bs,{\bs}n{\bs}t]*)               /* brancos                 */

 %%                              /*fim zona definicoes       */
                                 /* Regras                   */
 "\{"     printf("( makeset ");
 "\}"     printf(" )");
 ","     ;                       /*      come as virgulas   */
 . |                             /* qualquer outro caracter  */
 {\bs}n          ECHO;               /*      ou NL 'e ecoado    */
 {\bs}"(.)*{\bs}"   ECHO;               /*      nao altera strings */
 \{ID\}\{E\}{\bs}(   \{yytext[yyleng-1] = '{\bs}0'; /* retirar "("        */
              printf("( %s ",yytext);\} /* escreve "(" ID     */

 %%                              /*    codigo C auxiliar     */
                                 /*       nao foi preciso   */
```

utilizacao:

a) geraçao do programa

```
$ lex pp.l

$ cc -o pp lex.yy.c -ll
```

b) Texto a processar

```
$ cat f                          /* f- ficheiro a preprocessar */
 {1, 2, 3, 4, 5}
 f(a, b)
 g(f(x, c)) {1 2}
```

c) Resultado do preprocessamento

```
$ pp < f
 ( makeset 1 2 3 4 5 )
 ( f a b)
 ( g ( f x c)) ( makeset 1 2 )
```

1.7 Funcionamento como analisador l'exico.

Para funcionar como Analisador L'exico dum compilador (tipicamente designado por get_simbolo) a ser invocado por um Parser, cada vez que um simbolo 'e reconhecido, 'e necess'ario guardar o seu valor (numa vari'avel global) e retornar o respectivo c'odigo inteiro; para tal as regras ter ao o seguinte aspecto:

```
ExpReg    { yylval <-- [guardar o valor do simbolo];
            return(codigo); }
```

onde yylval 'e uma vari'avel global usada para passagem do valor e codigo 'e uma express ao inteira.

2 Exemplo:

```
%\{
#define INTEGER 33
%\}
...
[0-9]+   \{yylval = atoi (yytext); /* yylex() :                  */
         return(INTEGER); \}       /* logo que encontra um simbolo */
                                  /* devolve-o e termina a funcao */
...
```

Exemplos completos deste tipo de funcionamento ser ao dados no YACC.

3 Sintaxe dum texto LEX

Nos textos LEX, tem significado l'exico a coluna onde certos s'imbolos s ao colocados. Nomeadamente as linhas que come,cam com espa,co ou tab s ao copiadas para o texto final gerado sem alteraçao, enquanto que as linhas que comecem por qualquer outro caracter s ao normalmente processadas pelo lex.

No texto seguinte, @xx significa xx encostado 'a primeira coluna!!!

```
TextoLEX --> Definicoes CorpoPrincipal BlocoComplementar

Definicoes --> Definicao*

CorpoPrincipal --> @"%%" Clocal Regra*

BlocoComplementar --> @"%%" codCauxiliar
```

```
                 |                    /* bloco vazio */

   definicao --> @id ExpReg              (1)
              |   def_dimencoes          (2)
              |   def_cond_contexto      (3)
              |   def_tabela_traducao    (4)
              |   codigoCglobal          (5)
```

(1) associa id a ExpReg para uma maior clareza
(2) s'o necess'arias quando o scanner 'e muito grande (ver manual UNIX)
(3) Condi,c oes de Contexto: modo de tornar certas regras vis'iveis apenas em certos estados
(4) ver manual UNIX
(5) c'odigo C global; tipicamente usado para declara,c oes e inicializa,c oes de vari'aveis globais

```
   def_cond_contexto --> @"%s" id+
```

as condiçoes de contexto s ao estados com nome e permitem que regras a eles associdos s'o sejam vis'iveis em certas situaçoes. O estado por defeito 'e o estado 0; para se mudar de estado: "BEGIN id"

```
   codigoCglobal --> codigoC

   codigoC --> [codigo C desencostado]
            | @"%\{" [codigo qualquer] @"%\}"
```

para declarar variaveis globais, includes, e outro codigo C

```
   Clocal --> codigoC
```

O c'odigo C local 'e usado para fazer declara,c oes e inicializa,c oes dentro da fun,c ao "yylex".

```
   codCauxiliar --> [funcoes C e declaracoes globais]

   regra --> @ExpReg Accao

   Accao --> codigoC
         |                /* accao vazia       */
```

vazio = copiar yytext para output

```
ExpReg --> [contextEsq]["^"] ExpR ["$"][contextDir] (1)

contextEsq --> "<" id ("," id)* ">"               (2)

contextDir --> "/" ExpR                           (3)
```

(1) exemplo:
. ^aaa - aaa no in'icio de uma linha
. aaa$ - aaa no fim de uma linha

(2) indica o conjunto de estados em que esta regra 'e visivel; quando n ao 'e indicado contexto esquerdo, a regra 'e sempre visivel. Os identificadores de estado que aqui podem ser referidos s ao os id declarados em def_cond_contexto.

(3) regra so 'e vista se ExpR vier a seguir exemplo: .aaa/bbb aaa se bbb vier a seguir.

3.0.1 Express oes Regulares

As express oes regulares usadas em LEX s ao muito semelhantes ‘as j'a analisadas no "vi", "grep", "awk", etc.

```
ExpR--> carac              (o caracter indicado)
    |  "."                 (qualquer caracter diferente de {\bs}n("newline"))
    |  String              (exemplo: "abc" )
    |  ExpR ExpR           (concatenacao de Expressoes Regulares)
    |  "(" ExpR ")"        (agrupa termos, altera prioridades)
    |  "[" conjchar "]"    (qualquer caracter incluido em "conjchar")
    |  "[^" conjchar "]"   (qualquer caracter nao incluido em "conjchar")
    |  ExpR "*"            (0 ou mais repeticoes de "ExpR")
    |  ExpR "+"            (1 ou mais repeticoes de "ExpR")
    |  ExpR "?"            ("ExpR" ou nada )
    |  "\{" id "\}"          (expressao definida anteriomente)
    |  ExpR "\{" int1 "," int2 "\}"
                           (entre "int1" e "int2" repeticoes de "ExpR")
    |  ExpR1 "|" ExpR2     ("ExpR1" ou "ExpR2")
```

Exemplos:

```
(1) .                   {a,b,c,...}
(2) a.                  {aa,ab,ac,...}
(3) [a-z$]              {a,b,..,z,$}
(4) [^aA]               {x|not x in{a,A}}
```

```
(5) [A-Z][A-Z0-9]*      identificador habitual
(6) [+-]?[0-9]+         Inteiro habitual
(7) a{3,5}              {aaa,aaaa,aaaaa}

    conjchar --> (carac | gama )+

    gama --> carac "-" carac

    carac --> "{\bs}" octal
           |  "{\bs}" caracter_especial
           |  caracter_nao_especial
           |  "{\bs}n" "{\bs}t" "{\bs}f" ...

    caracter_especial --> *+{\bs}/[]()<>\{\}?^$",|-.    (e outros?)

    caracter_nao_especial --> todos menos os anteriores
```

3.1 Funçoes e Vari'aveis predefinidas

. **yytext** - texto (sequência de caracteres) que fez match com a express ao regular.
. **yyleng** - comprimento de yytext
. **ECHO** - equivalente a fprintf(yyout,". **REJECT** - ac,c ao de ignorar a express ao regular corrente com a qual foi feito o matching e procurar outra express ao regular.
. **unputc(c)** - volta a colocar o caracter na entrada.
. **yyless(n)** - volta a colocar os ultimos n caracteres na entrada para serem posteriormente processados (lidos)
. **input()** - le outro caracter de yyin
. **output(c)** - coloca c na saida yyout
. **yyin** - canal de comunica,c ao de entrada (default: stdin)
. **yyout** - canal de comunica,c ao de saida (default: stdout)
. **yywrap()** - procedimento invocado quando se atinge o fim de ficheiro (EOF)
. **yylineno** - variavel que contem o numero da linha que esta' a ser processada
. **yylval** - valor qualquer a associar ‘a express ao a reconhecida

4 Exemplo

Um problema habitual em desenvolvimento de software 'e a necessidade de testar um m'odulo constru'ido. Esse teste 'e frequentemente delicado e envolve por vezes necessidade de introduçao e an'alise de grandes quantidades de dados.

Nesta situçao 'e c'omodo construir um interpretador que permita seleccionar as funçoes e definir os respectivos parametros, quer interactivamente quer a partir de um ficheiro de "comandos", enviando o resultado para o ecran ou para um ficheiro.

No exemplo que se segue apresenta-se o texto "lex" correspondente a um m'odulo cujo interface se indica, salientando-se a simplicidade conseguida.

```
/*---------------------------------------------------------+
+   modulo ficheiro acesso directo                         +
+                                                          +
+        daf_novo    ( out:uni , nome , recl      ):bool    +
+        daf_open    ( out:uni , nome , recl      ):bool    +
+        daf_le      ( uni , rec , *atributos     ):bool    +
+        daf_escreve ( uni , rec , *atributos     ):bool    +
+        daf_reserva ( uni , out:rec              ):bool    +
+        daf_liberta ( uni , rec                  ):bool    +
+---------------------------------------------------------*/
%{
#define rec(l) r[l-'a']
#include "jjdaf.h"
%}

NOME ([^,)]*)
INT ([0-9]+)
R ([a-z])
E (" "*)
ATR ([^,)]*)

%s  erro
%%
    char nome[30];   /* Variaveis locais a yylex */
    int i;
    char re;
    int uni;
    char atrib[80];
    int r[26];

novo\({NOME}\,{INT}\)   {sscanf(yytext,"novo(%[^,],%d)",nome,&i);
                         daf_novo(&uni,nome,i);}

abre\({NOME}\,{INT}\)   {sscanf(yytext,"abre(%[^,],%d)",nome,&i);
```

```
                         daf_open(&uni,nome,i);}

reserva\({R}\)          {sscanf(yytext,"reserva(%c)",&re);
                         daf_reserva(uni,&rec(re));
                         printf("%c=%d\n",re,rec(re));}

escreve\({R}\,{ATR}\)   {sscanf(yytext,"escreve(%c,%[^)])",&re,atrib);
                         daf_escreve(uni,rec(re),atrib);}

le\({R}\)               {sscanf(yytext,"le(%c)",&re);
                         daf_le(uni,rec(re),atrib);
                         printf("atributo lido: %s\n",atrib);}

le\({INT}\)             {sscanf(yytext,"le(%d)",&i);
                         daf_le(uni,i,atrib);
                         printf("atributo lido: %s\n",atrib);}

letodos                 {for(i=1;!daf_le(uni,i,atrib);i++)
                           printf("%s\n",atrib);}

<erro>\n                {BEGIN 0; fprintf(stderr,"???\n");}

<erro>.*                 ;

" "|\n                   ;

.                       BEGIN erro;

%%
```

Geraçao do interpretador associado:

```
% cat makefile
  inter: inter.o jjdaf.o
        cc inter.o jjdaf.o -ll -o inter

  inter.o: inter.l jjdaf.h

  clean :
        rm *.o

% make
   lex  inter.l
        cc -O -c lex.yy.c
```

```
        rm lex.yy.c
        mv lex.yy.o inter.o
        cc -O -c jjdaf.c
        cc inter.o jjdaf.o -ll -o inter
```

Criaçao de um ficheiro de teste

```
% cat teste
   novo(malaquias,50)
   reserva(a)
   reserva(b)
   reserva(c)
   escreve(a,atributos do registo A)
   escreve(b,atributos do registo B)
   escreve(c,atributos do registo C)
   letodos
   escreve(a,abre:outros diferentes)
   le(a)
   le(b)
```

Teste:

```
% inter < teste
   a=1
   b=2
   c=3
   atributos do registo A
   atributos do registo B
   atributos do registo C
   atributo lido: abre:outros diferentes
   atributo lido: atributos do registo B
```