El generador de analizadores léxicos *lex*.
II

Teoría de Autómatas y lenguajes formales
Federico Simmross Wattenberg (fedesim@infor.uva.es)
Universidad de Valladolid

1. Sensibilidad al contexto

Lex dispone de una cierta capacidad (limitada) para reconocer lenguajes sensibles al contexto; es decir que ofrece la posibilidad de localizar lexemas "sólo en determinadas circunstancias". El caso más elemental de sensibilidad al contexto ya lo conocemos: se expresa con los metacaracteres '`^`'.y '`$`'. Además, lex reconoce también el carácter '/':

Siendo **a** y **b** dos expresiones regulares cualesquiera,

- **^a** significa "la expresión regular **a** al principio de una línea".
- **a$** significa "la expresión regular **a** al final de una línea".
- **a/b** significa "la expresión regular **a** pero sólo cuando va seguida de la expresión regular **b**".

En los tres casos, `yytext` contiene únicamente el texto que se ajusta a la expresión regular **a**.

Nótese que el significado en lex de ^ y $ es distinto del que tienen en las herramientas vistas anteriormente (grep, sed y awk), que no son sensibles al contexto. En éstas, tanto ^ como $ tienen significado por sí solos (no tienen que ir acompañados de la expresión regular **a** como en lex):

En grep, sed y awk,

- ^ significa "un principio de línea".
- $ significa "un final de línea".

Para simular este comportamiento en lex, se suele utilizar como patrón el carácter salto de línea '\n'. Recuérdese que no suele ser buena idea utilizar este carácter en expresiones regulares de grep, sed o awk, dado que están orientados a líneas y por tanto, intentar reconocer un salto de línea rompe su esquema de funcionamiento.

Otra diferencia notable entre lex y las demás herramientas vistas es, precisamente, que lex no está orientado a líneas, con lo cual, una vez que se reconoce un patrón regular no disponemos de toda la línea para manipularla sino sólo de la cadena de texto que coincide con el patrón (el array `yytext`). Esto puede parecer un inconveniente pero es más bien una ventaja, puesto que no estamos limitados a un patrón regular por línea, y además no existen restricciones sobre la longitud de una línea. En general, lex es capaz de hacer todo lo que hacen grep, sed y awk, y bastantes cosas más.

2. La acción especial `REJECT`

Al igual que ECHO, REJECT está implementada como una macro de C. Significa "volver a poner el patrón reconocido (`yytext`) en la entrada y comprobar si ésta se ajusta a otro patrón", es decir, que la regla *rechaza* el patrón reconocido.

La mayoría de las veces no es necesario utilizar REJECT. Además, basta con que aparezca una sola vez en el programa lex para que todo el analizador generado resulte sustancialmente más lento. Sin embargo, REJECT es útil cuando queremos que un mismo patrón sea reconocido por dos o más reglas distintas. También puede servir para forzar a lex a que reconozca una cadena de la entrada que no es la más larga posible.

3. Definiciones

Una definición Lex permite "abreviar" una expresión regular larga, de manera que sea más cómoda de usar posteriormente. Una vez definida, se puede utilizar la expresión regular abreviada simplemente escribiéndola entre llaves '{}', tanto en la sección de reglas como en la de definiciones. Se puede utilizar una definición dentro de otra definición. Veamos un ejemplo:

```
digito                  [0-9]
numero                  {digito}+
letra                   [A-Za-z]
identificador           {letra}({letra}|{digito})*

%%
{numero}                printf("NUMERO");
{identificador}         printf("IDENTIFICADOR");
```

4. Ejemplos

- Buscar y corregir (posibles) errores ortográficos en los que el verbo haber está escrito sin hache antes de un participio:

```
%%
" "a" "/[a-z]*[ai]do    printf (" ha ");
```

- Contar cuántas veces en un texto aparece la cadena ‘la’, y cuántas la cadena ‘ella’, teniendo en cuenta que dentro de ‘ella’ hay un ‘la’:

```
 int ella=0;
 int la=0;
%%
ella        {ella++; REJECT;}
la          la++;
.|\n        ;

%%
main(){
    yylex();
    printf ("Cadenas 'ella': %d\n Cadenas 'la': %d\n",
ella, la);
}
```

- En un programa en C, sustituir los números enteros por la palabra “entero”, y los reales por la palabra “real”:

```
dig     [0-9]
let     [a-zA-Z]
expo    [Ee][-+]?{dig}+

%%
{dig}+                  printf ("entero");
{dig}+\.{dig}*{expo}?   |
{dig}+{expo}            printf ("real");
```

5. Ejercicios

1- En el fichero `quijote.txt`, contar cuántas veces una palabra acaba en la misma letra con la que empieza la siguiente, teniendo en cuenta que entre dos palabras consecutivas puede haber signos de puntuación y saltos de línea.

2- Este programa debería sustituir los identificadores de un programa en C por la cadena IDENT, y las llamadas a función por la cadena FUNCALL. Aplicarlo al fichero `findip.c`, comprobar por qué no funciona bien e intentar que funcione lo mejor posible.

```
letra                   [A-Za-z]
digito                  [0-9]
ident                   {letra}({letra}|{digito})*

%%
{ident}                 printf("IDENT");
{ident}"("[^)]*")"      printf("FUNCALL");
```