Tema 3

Análisis Léxico - Flex

Universidad Europea

LAUREATE INTERNATIONAL UNIVERSITIES

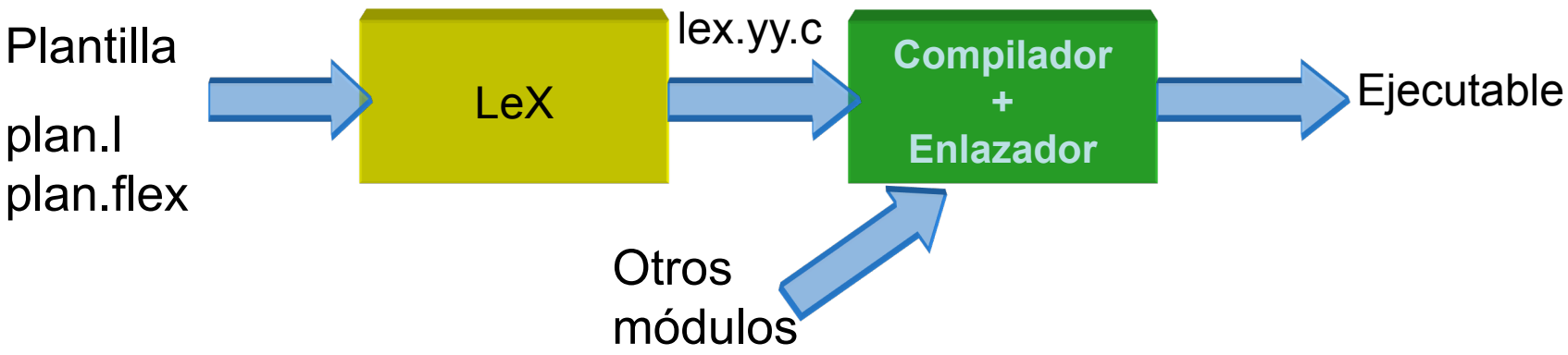
Autor: Prof. Leopoldo Santos Santos

2018

Análisis léxico

- Flex. Esquema de Funcionamiento
- Programación
- Plantilla
- Formato de un fichero Flex
- Sección de Declaraciones, Reglas y Código de Usuario
- Ejemplos de Uso
- Patrones y prioridades

Flex. Esquema de funcionamiento



Programación con LEX

- **Lex: Lexical Analyzer Generator**
  - GNU: flex: fast lex
- **Lex programa fuente**

  {definición}

  %%

  {reglas}

  %%

  {subrutinas de usuario}

  Reglas: <expresión regular> acción

  Cada expresión regular especifica un *token*

  Acción: fragmentos de código C que especifican que hacer cuando un *token* es reconocido

Plantilla de Flex

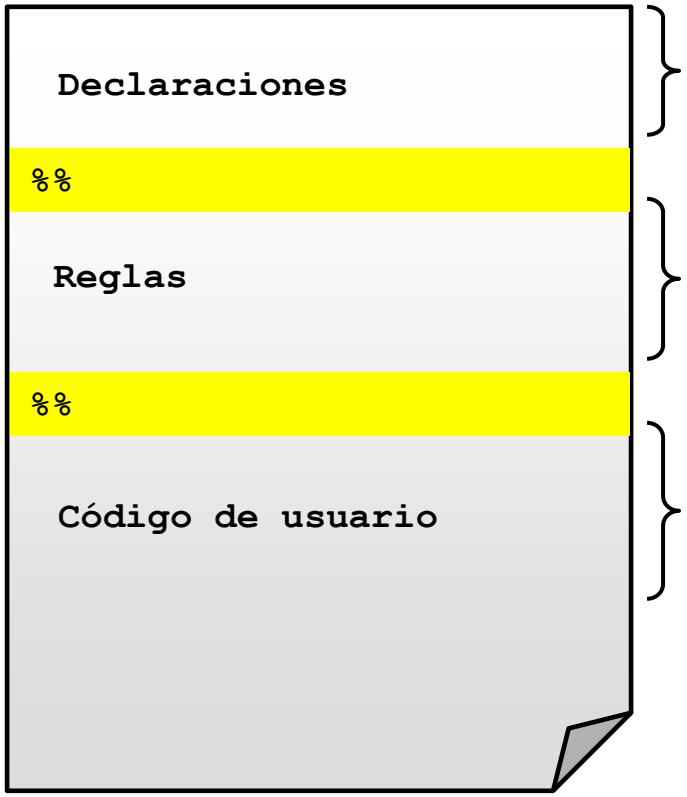
- LeX evalúa un fichero de entrada para implementar un analizador léxico especificado
- Las reglas de reconocimiento tienen la estructura patrón-acción, mediante expresiones regulares
- El reconocimiento puede ser contextualizado mediante condiciones de arranque
- El analizador está contenido en la función yylex(): código C

- Proceso de Compilación

```
home\> flex prueba.flex
home\> gcc lex.yy.c –L/lib –lfl
```

Formato de FleX

- Tres partes separadas por una línea con “%%”



Previas al uso del scanner

Patrones de entrada y acciones a realizar cuando concuerda la entrada, es obligatoria

Funciones C que se copian al fichero de salida

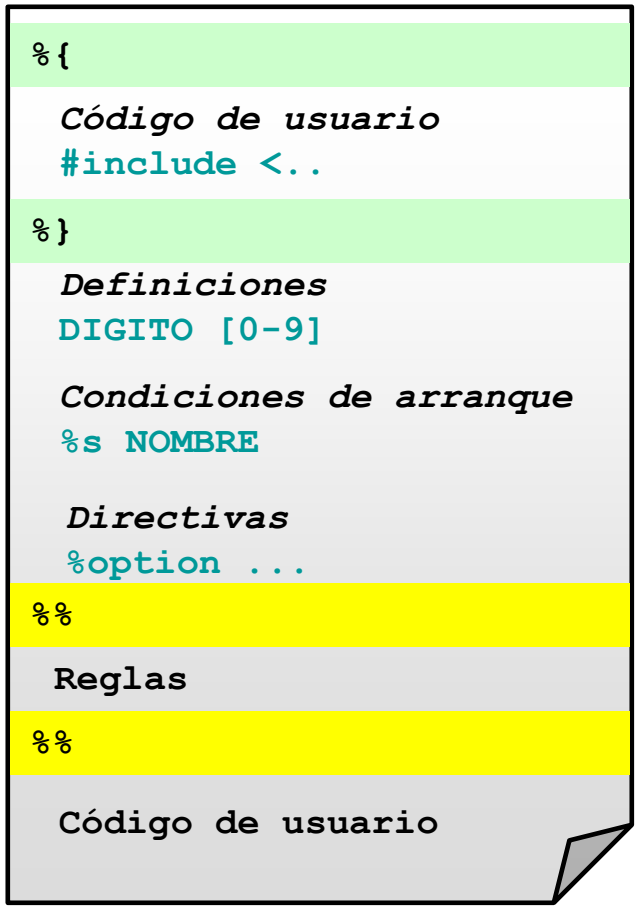
Formato de Flex

- **Funcionamiento general**: el analizador creado busca en la entrada ocurrencia de los patrones
  - Cuando encuentra un patrón, ejecuta sus acciones
  - Si no devuelven el control (return), continúa la búsqueda de nuevos patrones
  - Si varios patrones encajan, selecciona el más largo y el primero declarado
- Por defecto, el texto que no encaja con ningún patrón lo envía a la salida.

Sección de Declaraciones

- Contiene cuatro tipos de declaraciones
  - **Código C** del usuario que aparecerá copiado en el fichero de salida, delimitado por "`%{`" y "`%}`"
    - Incluir fichero de cabecera
    - Declarar variables globales
    - Declarar procedimientos que se describirán en la parte de sección Código de usuario
  - **Definición de alias**, poner nombre a expresiones regulares
    - Aparece al principio de la línea
    - Se separa la expresión regular por un espacio o tabulador
  - Definición de las **condiciones de arranque**
  - **Directivas** que controlan el comportamiento de LeX
- Pueden aparecer en cualquier orden

Sección de Declaraciones



Sección de Declaraciones

- Definiciones
  - Dar nombre a patrones complejos facilitando la legilibilidad
  - Se pueden anidar, otras definiciones se pueden utilizar entre llaves

```
entero      [0-9]+
real        {entero}.{entero}
real2       {real}[eE][\+\-]?{entero}
numero      {entero}|{real}|{real2}
```

Representación de patrones

Literal	“x”	La cadena x
Literal	*	El carácter literal: ‘*’
Definición	{nombre}	La expresión regular nombre (uso)
Selección	a|ab	Selección de una alternativa: {a, ab}
Rango	[ad-gB-EG2-4]	{a,d,e,f,g,B,C,D,E,G,2,3,4}
Negación del Rango	[^a-z]	Cualquiera excepto minúsculas
Agrupar	(a-z)|(0-9)	Para agrupar patrones [a-z0-9]
Numeración	r*	Ocurrencia de r >=0
	r+	Ocurrencia de r >0
	r?	Cero o una ocurrencia de r
	r{2,4}	De 2 a 4 ocurrencias de r
Cualquier carácter (no \n)	.	(.|\n)* representa cualquier fichero
Localización	^r	r al principio de la línea
	r$	r al final de la línea

Sección de Declaraciones

- **Código de usuario**

```
%{
/* Este bloque aparecerá tal cual en el fichero yy.lex.c */
#include <stdio.h>
#include <stdlib.h>
#define VALUE 33
int nl, np, nw;
%}
```

  - Se copia tal cual en la cabecera del fichero lex.yy.c
  - La variables serán globales a todas las funciones del fichero

Sección de Declaraciones

- Directivas de funcionamiento
  - Cambian el funcionamiento por defecto de LeX

    `%option case-insensitive`
    - Considera las mayúsculas y minúsculas como el mismo carácter

    `%option yylineno`
    - Permite utilizar la función **`yylineno`** que guarda el número de línea que está procesando

    `%option noyywrap`
    - Permite usar LeX sin necesidad de definir la función **`yywrap`** que maneja varios ficheros de entrada

Sección de Reglas

- Utilizan el formato PATRÓN ACCIÓN
- Los patrones pueden utilizar definiciones, expresiones regulares y condiciones de arranque
- Las acciones son código C, excepto la activación y desactivación de las condiciones de arranque
- El lexema que concuerda con el patrón está en la variable **`yytext`**, su longitud es **`yyleng`**
  - **`Ej.:[a-z] {printf("%s",yytext); return(1);}`**

Sección de reglas

- Reglas para la identificación de patrones
  - Siempre que para la entrada puedan aplicarse varias reglas:
  1. Se aplica el patrón que concuerda con el mayor número de caracteres en la entrada
     - Con la entrada `abc`:

```
a   {return (1);}
ab  {return (2);}
c   {return (3);}
abc {return (4);}
```

  2. Si hay dos patrones que concuerdan con el mismo número de caracteres en la entrada, entonces se aplica el que esté definido primero

Sección de Código de Usuario

- Código C que es copiado al fichero lex.yy.c
- Si no escribimos esta sección es equivalente a escribir:

```
int main ()
{
   yyin = stdin;
   yylex ();
}
```

- Ejemplo de main que acepta un fichero como entrada

Sección de Código de Usuario

```
int main (int argc, char *argv[])
{
  if (argc == 2)
  {
     yyin = fopen (argv[1], “rt”);
     if (yyin == NULL)
     {
          printf (“Fichero %s erróneo\n”, argv[1]);
          exit (-1);
      }
   }
   else yyin = stdin;
   yylex ();
   return 0;
}
```

Ejemplo: Fichero de entrada y otro de salida

```
int main(int argc,char *argv[])
{
   if ((yyin = fopen(argv[1], "rt")) == NULL)
{ printf("\nNo se puede abrir el archivo: %s\n", argv[1]);
}
   else
{
          if ( (output = fopen( "./salida.txt", "wt" )) == NULL )
          {
                    printf("Error creando el fichero de salida.\n");
          }
          else
          {
                    fprintf(output,"%-30s%-30s%-15s\n", "TOKEN","LEXEMA","LINEA");
                    fprintf(output,"%-30s%-30s%-15s\n", "-----","------","-----\n");
                    yylex();
                    printf("\n¡¡Fichero salida.txt generado correctamente!!\n");
          }
}
fclose(yyin);
return 0;
}
```

Variables, Funciones, Procedimientos, Macros

- Flex incorpora facilidades, las más comunes son:

Variable	Tipo	Descripción
yytext	char * o char []	Contiene la cadena de texto del fichero de entrada que ha encajado con la expresión regular descrita en la regla.
yyleng	int	Longitud de yytext yylength = strlen (yytext)
yyin	FILE*	Referencia al fichero de entrada.
yyval yylval	struct	Contienen la estructura de datos de la pila con la que trabaja YACC. Sirve para el intercambio de información entre ambas herramientas.

Variables, Funciones, Procedimientos, Macros

Método	Descripción
yylex()	Invoca al Analizador Léxico, el comienzo del procesamiento.
yymore()	Añade el yytext actual al siguiente reconocido.
yyless(n)	Devuelve los n últimos caracteres de la cadena yytext a la entrada.
yyerror()	Se invoca automáticamente cuando no se puede aplicar ninguna regla.
yywrap()	Se invoca automáticamente al encontrar el final del fichero de entrada.

- yyerror e yywrap pueden ser reescritos

Nombre	Descripción
ECHO	Escribe yytext en la salida estandar. ECHO = printf (“%s”, yytext)
REJECT	Rechaza la aplicación de la regla. Pone yytext de nuevo en la entrada y busca otra regla donde encajar la entrada. REJECT = yyless (yylength) + ‘Otra regla’
BEGIN	Fija nuevas condiciones para las reglas. (Ver Condiciones sobre Reglas).
END	Elimina condiciones para las reglas. (Ver Condiciones sobre Reglas).

Ejemplo de uso

Ejemplo 1: comando wc de Linux

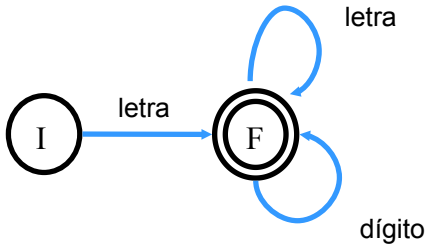
```
%{
#include <stdio.h>
int nc, np, nl;
%}
%%
/*----- Sección de Reglas ----------------*/
[^ \t\n]+ { np++; nc += yylength; }
[ \t]+ { nc += yylength; }
$ { nl++; nc++; }
%%
```

Proceso de Compilación

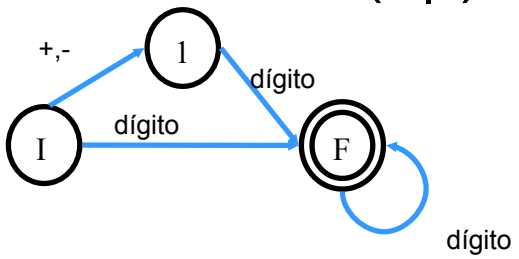
```
/*----- Sección de Procedimientos --------*/
int main (int argc, char *argv[])
{
if (argc == 2)
{
yyin = fopen (argv[1], “rt”);
if (yyin == NULL)
{
printf (“El fichero %s no se puede abrir\n”, argv[1]);
exit (-1);
}
}
else yyin = stdin;
nc = np = nl = 0;
yylex ();
printf (“%d\t%d\t%d\n”, nc, np, nl);
return 0;
}
```

Ejemplos de patrones léxicos

- ER identificador: letra (letra|dígito)*



- ER número entero: (+|-)? digito+



- ER número real:

  (+|-)? digito+ (.digito*)?((e|E)(+|-)digito+)?

Ejemplos patrones léxicos

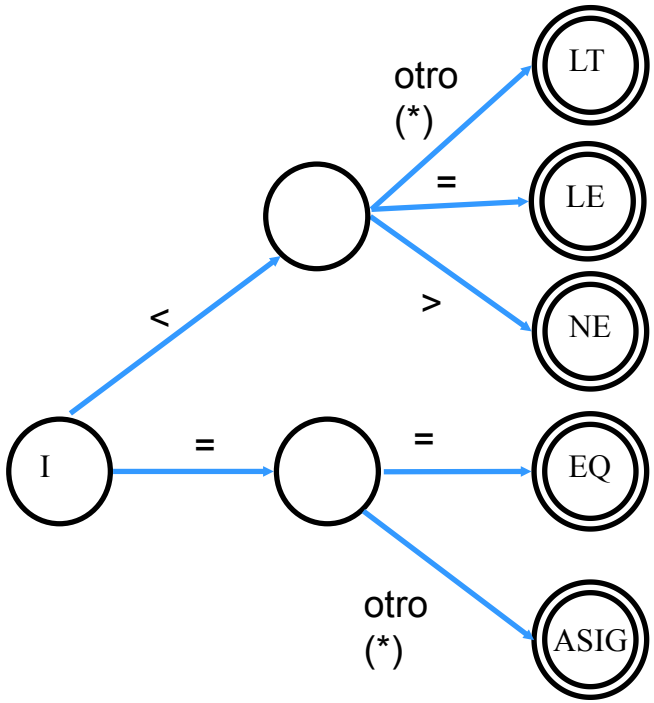
```
%{
#include <stdio.h>
int nc, np, nl;
//comentario de una linea
%}
%option noyywrap

digito [0-9]
letra [a-zA-Z]
%%
{letra}({letra}|{digito})*
                {printf("token IDENTIFICADOR: %s\n",yytext);}
[-+]?{digito}+  {printf("token ENTERO: %s\n",yytext);}
[-+]?{digito}+(.{digito}+)?((E|e)[-+]?{digito}+)?
                {printf("token REAL: %s\n",yytext);}
.|\n {/*no hace nada con resto*/}

%%
```

Patrones con prioridades

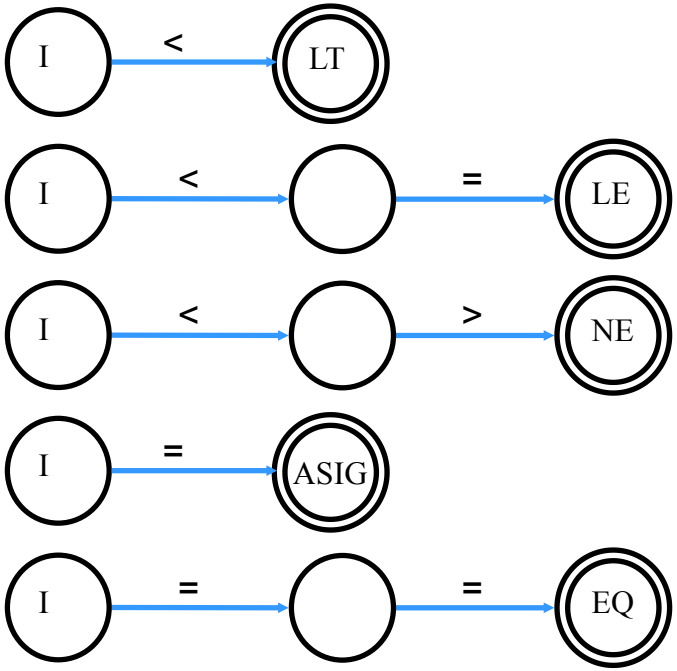
- Tokens cuyos patrones comienzan igual
  - EQ: “==”
  - ASIG: “=”
  - LT: “<”
  - LE: “<=”
  - NE: “<>”



*: devolver carácter entrada

Patrones con prioridades

- Ambigüedad: tokens cuyos patrones comienzan igual
  - EQ: “==”
  - ASIG: “=”
  - LT: “<”
  - LE: “<=”
  - NE: “<>”



  - Pueden definirse varios patrones solapados. **Prioridad** el del lexema reconocido más largo

Patrones con prioridades

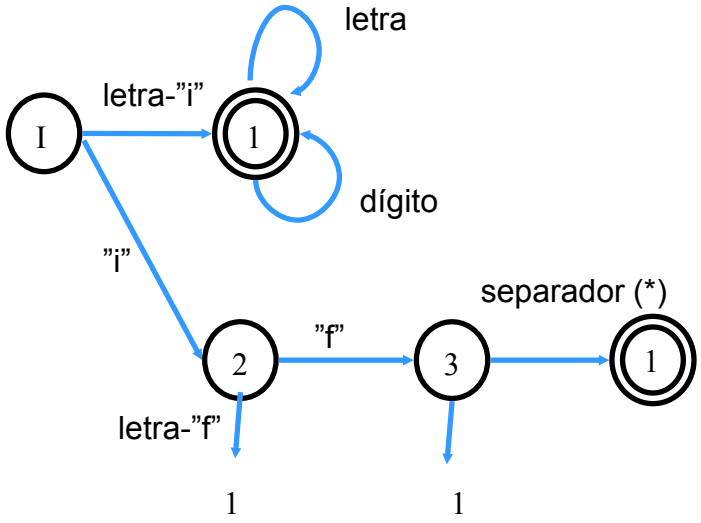
```
%{
#include <stdio.h>
int nc, np, nl;
//comentario de una linea
%}
%option noyywrap

%%
%%
"<"  {printf("token LT\n");}
"<=" {printf("token LE\n");}
"<>" {printf("token NE\n");}
"=" {printf("token ASIG\n");}
"==" {printf("token EQ\n");}
.|\n {/*no hace nada con resto*/}


%%
```

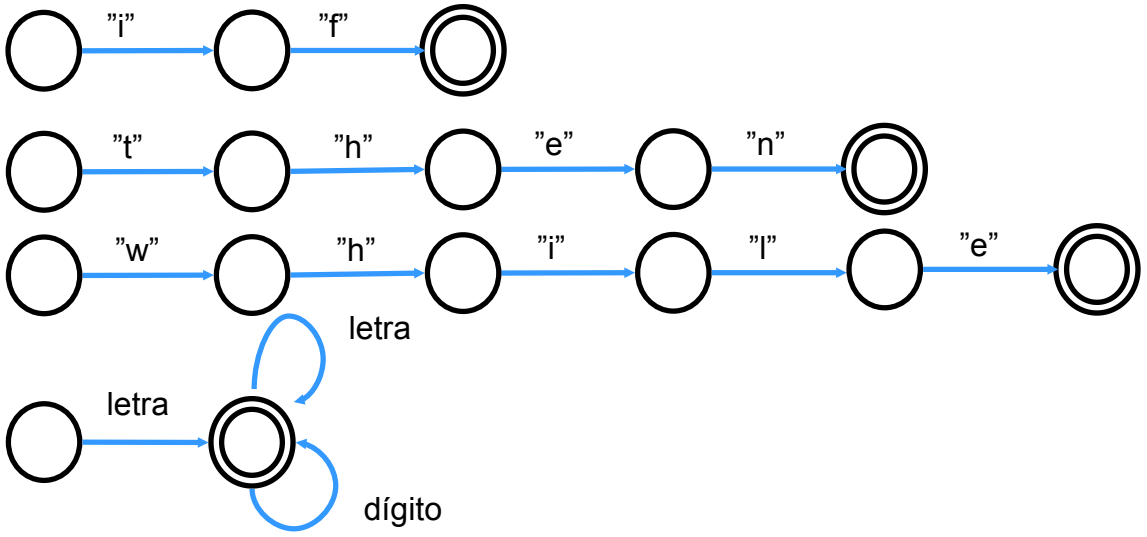
Distinguir palabras reservadas

- Ambigüedad: algunos patrones pueden coincidir (palabras reservadas explícitas):
  - IDENTIFICADOR: letra (letra|dígito)*
  - IF: “if”
  - THEN: “then”
  - WHILE: “while”



Distinguir palabras reservadas

- **Ante coincidencia en longitud, se toma el definido primero**
  - IDENTIFICADOR: letra (letra|dígito)*
  - IF: “if”
  - THEN: “then”
  - WHILE: “while”



Patrones palabras reservadas

```
%{
#include <stdio.h>
int nc, np, nl;
//comentario de una linea
%}
%option noyywrap

digito [0-9]
letra [a-zA-Z]
%%
"if"  {printf("token IF\n");}
"else" {printf("token ELSE\n");}
"while" {printf("token WHILE\n");}
{letra}({letra}|{digito})* {
                printf("token IDENTIFICADOR: %s\n",yytext);}
.|\n
{/*no hace nada con resto*/}

%%
```

Utilizando las condiciones de arranque

- <cond1 [, cond2]*>patrón {acciones}

```
{id} { ECHO; }
“/*” { BEGIN comentario; }
<comentario>”*/” { END comentario; /* = BEGIN 0; */ }
<comentario>. { }
```

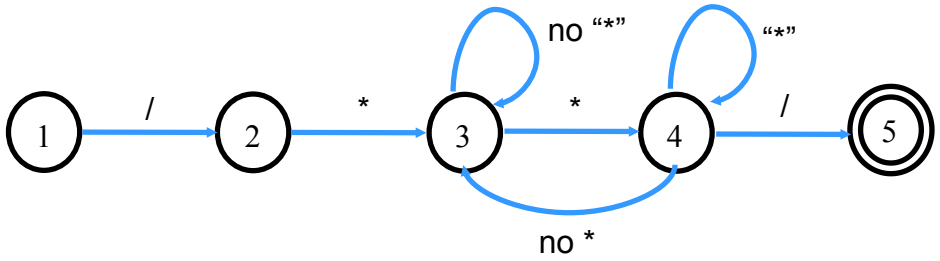
- Con BEGIN 0 se vuelve al estado en el cual los patrones no necesitan condiciones de arranque para ser evaluados
- Para indicar la condiciones iniciales de procesamiento

```
%start cond1 [, cond2]*
```

Utilizando las condiciones de arranque

- Ejemplo práctico: comentarios en C

```
/*comentario en C ***/
```



- Expresión Regular para el AFD **muy complicada**
  - Los estados léxicos permiten con Lex representar no sólo expresiones regulares

Patrones comentarios C

```
%%
%{
#include <stdio.h>
int nc, np, nl;
//comentario de una linea
%}
%option noyywrap

%x comentario

%%
^"//".*\n {ECHO;}
"/*" { BEGIN comentario; }
<comentario>"*/" {BEGIN INITIAL;}
<comentario>.|\n {ECHO;}
.|\n
```