Seminario FLEX

Compiladores

Dpto. de Ingeniería de la Información y las Comunicaciones

Estudio de la Herramienta Flex

1. Formato de un fichero Flex
2. Acciones de Flex
3. Uso de Flex
4. Esquema para trabajar con Flex
5. Funciones y variables de Flex
6. Ejemplos
7. **Ejercicio primero**: Comenzando a trabajar con *Flex*
8. **Ejercicio segundo**: Diseño de un analizador léxico
9. Condiciones de contexto o de arranque
10. **Ejercicio tercero**: comentarios multilínea y *modo pánico*

Formato de un fichero Flex

Formato de un fichero.l :
{definiciones}
%%
{reglas}
%%
{subrutinas de usuario}

Las tres secciones son opcionales.

- **Definiciones** : Se especifican, entre otras cosas, macros que dan nombre a expresiones regulares auxiliares que serán utilizadas en la sección de reglas. Se puede incluir código C, condiciones de contexto, tablas, etc.
- **Reglas** : En esta sección hay secuencias del tipo r_i $accion_i$; dónde r_i es una expresión regular y $accion_i$ es un bloque de sentencias en C que se ejecutan cuando se reconoce el lexema asociado a esa expresión regular.
- **Subrutinas de usuario** : Procedimientos auxiliares necesarios.

Acciones de Flex

- Para que Flex *devuelva un código de token* después de reconocer un lexema mediante una expresión regular, debemos poner, después de ésta, la acción correspondiente, que en este caso será:

  $\rightarrow$ return código_token ;

- Para que ignore una secuencia de caracteres sin devolver ningún código de token, ponemos al lado de la expresión regular correspondiente la sentencia nula:

  $\rightarrow$;

- Si lo que lee de la entrada no lo reconoce, por defecto lo escribe en la salida. Por tanto, si queremos filtrar la entrada completamente, debemos especificar expresiones regulares que reconozcan cualquier secuencia de caracteres de entrada.

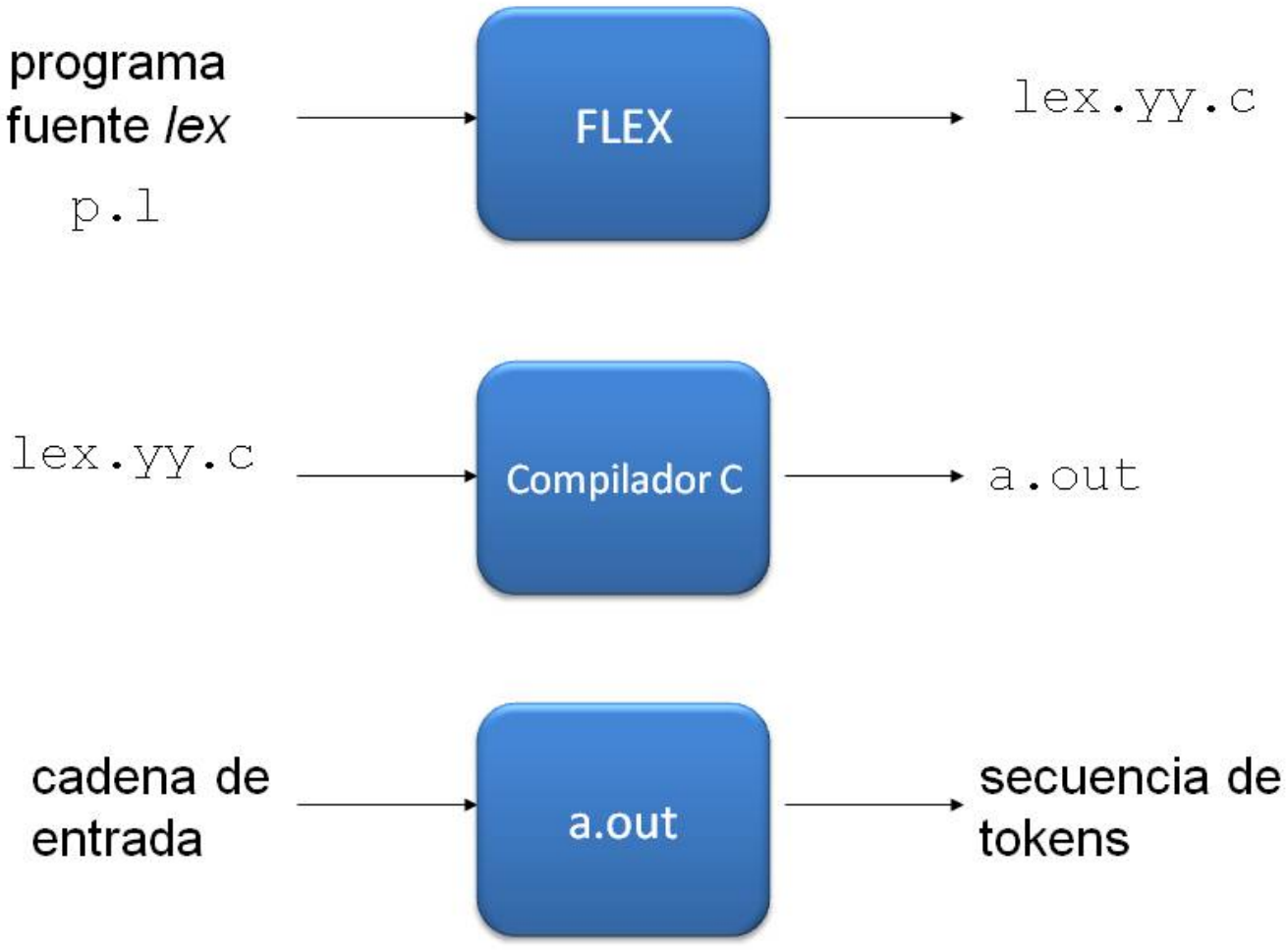
Uso de Flex

- A partir de un fichero, que suele tener extensión .l, genera otro, llamado lex.yy.c, que contiene la función `yylex()`, con la que se reconoce, cada vez que es llamada, un lexema y se ejecuta una acción.
- La creación de un analizador léxico usando flex se realizaría con los siguientes comandos:

```
flex fichero.l
gcc [fichero.c] lex.yy.c -lfl -o fichero
```

según el siguiente esquema:

Esquema para trabajar con Flex



Funciones y variables de Flex

Podemos usar las variables globales *yytext*, *yyleng* e *yylineno*, y las funciones *yymore()*, *yyless(n)* e *yywrap()*.

- *yytext*: puntero a la última cadena de texto que se ajustó al patrón de una expresión regular.
- *yyleng*: variable global que contiene la longitud de *yytext*.
- *yylineno*: activando la opción *yylineno*[1] en la sección de declaraciones, flex genera un analizador que mantiene el número de la línea actual leída desde su entrada en la variable global *yylineno*.
- *yymore()*: sirve para indicar que la próxima vez que se empareje una regla, el valor nuevo debería ser concatenado al valor que contiene *yytext* en lugar de reemplazarlo.
- *yyless(n)*: permite retrasar el puntero de lectura, de manera que apunta al carácter n de *yytext*.
- *yywrap()*: función que se ejecuta cada vez que se alcanza el final del fichero.

[1] `%option yylineno`

Funciones y variables de Flex

Podemos cambiar la entrada y salida estándar haciendo uso de los punteros *yyin* e *yyout* y de las funciones *input( )*, *output(c)* y *unput(c)*.

- *yyin* puntero al descriptor de fichero de dónde se leen los caracteres.
- *yyout* puntero al fichero en el que escribe el analizador léxico al utilizar la opción ECHO.
- *input()* lee desde el flujo de entrada el siguiente carácter.
- *output(c)* escribe el carácter c en la salida.
- *unput(c)* coloca el carácter c en el flujo de entrada, de manera que será el primer carácter leído en próxima ocasión.

Ejemplos: lexico1.h

```
#define     BEGINN      1
#define     END         2
#define     READ        3
#define     WRITE       4
#define     ID          5
#define     INTLITERAL  6
#define     LPAREN      7
#define     RPAREN      8
#define     SEMICOLON   9
#define     COMMA       10
#define     ASSIGNOP    11
#define     PLUSOP      12
#define     MINUSOP     13
#define     REALLIT     14
```

Ejemplo 1: lexico1.l

```
%{
#include "lexico1.h"
%}
digito                  [0-9]
letra                   [a-zA-Z]
entero                  {digito}+
%%
[ \n\t]+                                ;
"--"(.*)[\n]                            ;
begin                                   return BEGINN;
end                                     return END;
read                                    return READ;
write                                   return WRITE;
{letra}({letra}|{digito}|_)*            return ID;
{entero}                                return INTLITERAL;
"("                                     return LPAREN;
")"                                     return RPAREN;
";"                                     return SEMICOLON;
","                                     return COMMA;
":="                                    return ASSIGNOP;
"+"                                     return PLUSOP;
"-"                                     return MINUSOP;
{entero}[.]{entero}                     return REALLIT;
.                                       error_lexico();
%%

error_lexico()
{
  printf("\nERROR, símbolo no reconocido %s\n",yytext);
}

main() {
    int i;
    while (i=yylex())
        printf("%d %s\n",i,yytext);
    printf("FIN DE ANALISIS LEXICO\n");
}
```

Ejemplo 2: lexico2.l

```
%{
#include "lexico1.h"
%}
digito                  [0-9]
letra                   [a-zA-Z]
entero                  {digito}+

%%
[ \n\t]+                                    ;
"--"(.*)[\n]                                ;
begin                                       return BEGINN;
end                                         return END;
read                                        return READ;
write                                       return WRITE;
{letra}({letra}|{digito}|_)*                return ID;
{entero}                                    return INTLITERAL;
"("                                         return LPAREN;
")"                                         return RPAREN;
";"                                         return SEMICOLON;
","                                         return COMMA;
":="                                        return ASSIGNOP;
"+"                                         return PLUSOP;
"-"                                         return MINUSOP;
{entero}[.]{entero}                         return REALLIT;
.                                           printf("Error en carácter %s",yytext );
%%
```

Ejemplo 2: main.c

```
#include <stdio.h>
#include "lexico1.h"

extern char *yytext;
extern int  yyleng;
extern FILE *yyin;
FILE *fich;
main()
{
    int i;
    char nombre[80];
    printf("INTRODUCE NOMBRE DE FICHERO FUENTE:");
    gets(nombre);
    printf("\n");
    if ((fich=fopen(nombre,"r"))==NULL) {
        printf("***ERROR, no puedo abrir el fichero\n");
        exit(1); }
    yyin=fich;
    while (i=yylex()) {
        printf("TOKEN %d",i);
        if(i==ID) printf("LEXEMA %s  LONGITUD %d\n",yytext,yyleng);
        else printf("\n");}
    fclose(fich);
    return 0;
}
```

Ejercicio primero: Comenzando a trabajar con Flex

Analizador léxico con Flex para un lenguaje de programación

Usa los ficheros anteriores para realizar un reconocedor de un lenguaje semejante a PASCAL:

- Genera una analizador con `lexico1.h` y `lexico1.l` y pruébalo.
- Genera una analizador con `lexico1.h`, `lexico2.l` y `main.c` y pruébalo.

Ejercicio segundo: Diseño de un analizador léxico

Analizador léxico con Flex para un lenguaje de programación

Usa y modifica los ficheros anteriores para realizar un reconocedor de los tokens de un lenguaje en el que:

- Todos los programas comienzan con '`main ()`'.
- **Tipo de dato**: entero.
- No se declaran **variables**.
- Los **identificadores** empiezan por letra, '_' o '$' y se componen de letra, '_', '$' o dígito.
- Solo hay **constantes** enteras.
- Los **comentarios** comienzan por '//' y terminan con *new-line*.
- **Sentencias**:
  - `ID=expresion;`
    para asignaciones de expresiones aritméticas con '+', '-' y paréntesis.
  - `print ( lista de expresiones entre comas );`
- **Palabras reservadas**: '`main`', '`print`'.
- Las sentencias se separan con ';'.
- El cuerpo del programa está delimitado entre '{' y '}'.
- Separadores de tokens: espacios en blanco, tabuladores y *new-line*.
- Los **caracteres especiales** con significado serían, por tanto:
  '(', ')', '=', ';', '+', '-', ',', '{' y '}'.

Condiciones de contexto o de arranque

Las **condiciones de arranque o de contexto** se usan cuando una regla sólo va a ser ejecutada si el intérprete del autómata se encuentra en una condición de entrada x. En este caso la regla debería llevar el prefijo <x>

- Declaración de cond. En la sección de definiciones se pone:
  - → %s *cond* o %*start cond* (inclusivas): Las reglas sin condiciones de arranque también estarán activas cuando se active *cond*.
  - → %*x cond* (exclusivas): Solamente las reglas cualificadas con la condición de arranque estarán activas al activar *cond*.
- Activación de cond. Se usa la acción BEGIN:
  - → BEGIN *cond*.
  - → Poniendo BEGIN(0) o BEGIN(INITIAL) se retorna al estado original, donde sólo las reglas sin condiciones de arranque están activas.

Condiciones de contexto o de arranque

- Se puede usar <cond1,cond2,...> para referenciar una regla que se activa con *cond1* o bien con *cond2*,...
- Una regla con el prefijo <*INITIAL*> sólo estará activa en el estado inicial.
- <*> empareja todas las condiciones de arranque.

Condiciones de contexto o de arranque

Ejemplo Condiciones de Arranque o de contexto.

Copiar la entrada en la salida, cambiando "hola" por "holaa" si la línea empieza con una a, por "holab" si empieza con una b y por "holac" si empieza con una c.

SOLUCIÓN 1

```
    int flag;
%%
^a {flag = 'a'; ECHO;}
^b {flag = 'b'; ECHO;}
^c {flag = 'c'; ECHO;}
\n {flag =  0 ; ECHO;}
hola {
    switch (flag)
        {
            case 'a': printf("holaa"); break;
            case 'b': printf("holab"); break;
            case 'c': printf("holac"); break;
            default: ECHO; break;
        }
    }
```

Condiciones de contexto o de arranque

Ejemplo Condiciones de Arranque o de contexto.

Copiar la entrada en la salida, cambiando "hola" por "holaa" si la línea empieza con una a, por "holab" si empieza con una b y por "holac" si empieza con una c.

SOLUCIÓN 2

```
%START AA BB CC
%%
^a {ECHO; BEGIN AA;}
^b {ECHO; BEGIN BB;}
^c {ECHO; BEGIN CC;}
\n {ECHO; BEGIN 0 ;}
<AA>hola printf("holaa");
<BB>hola printf("holab");
<CC>hola printf("holac");
```

Ejercicio tercero: comentarios multilínea y modo pánico

Comentarios multilínea tipo C

Completa el analizador del ejercicio segundo para que:

- Reconozca y elimine los comentarios multilínea tipo C.
- Reconozca e imprima los comentarios multilínea tipo C.

Recuperación de errores en modo pánico

Completa el analizador del ejercicio segundo para que:

- Una vez encontrado un carácter erróneo, localice el siguiente comienzo de un token correcto e imprima correctamente el mensaje de error.