Práctica 3: Construcción de una mini shell

Diseño de Sistemas Operativos – U.L.P.G.C.

David Jesús Horat Flotats
Enrique Fernández Perdomo

Índice

Introducción

El objetivo de esta práctica es realizar un programa utilizando las llamadas al sistema de manejo de procesos fork, wait, exec, que permita ejecutar cualquier comando del sistema operativo o ejecutar otro programa, lanzar en modo tanda & y que admita pipe |, así como redireccionamiento de entrada < y salida >. Se buscarán los programas automáticamente en los directorios /bin, /usr/bin y en el directorio actual.

Para ello usaremos la distribución Linux Fedora Core 3[1] tanto en los ordenadores del laboratorio como en los nuestros. No usaremos ningún entorno integrado de desarrollo, sino un editor de texto (vim) y un compilador (gcc). Además, para la parte del análisis léxico y sintáctico, hemos decidido usar Flex y Bison respectivamente.

Flex[2] es un generador de analizadores léxicos. Si no se da ningún nombre de fichero, Flex lee la entrada estándar (stdin) donde se encuentra la descripción del analizador léxico que se debe generar. La descripción debe realizarse por pares conformados por expresiones regulares y código en C, llamados reglas. Flex genera como salida un código en C dentro del archivo "lex.yy.c" en donde se define un procedimiento llamado "yylex()". Cuando se llama, analiza la entrada para encontrar ocurrencias de las expresiones regulares definidas y, en caso de encontrar alguna, ejecuta el código en C asociado a su regla.

Bison[3] es un generador de analizadores sintácticos. Bison convierte la descripción de una gramática independiente del contexto LALR en un programa en C que analiza la gramática descrita. Bison es compatible con Yacc: todas las gramáticas bien formadas de Yacc deberían funcionar en Bison sin cambio alguno. Cualquier persona que esté familiarizada con Yacc debería ser capaz de usar Bison sin problemas.

1 Fedora Core: http://fedora.redhat.com/
2 Flex: http://www.gnu.org/software/flex/
3 Bison: http://www.gnu.org/software/bison/

Características principales

- ✔ Admite varios pipes "|"
- ✔ Admite varios punto y coma ";"
- ✔ Admite redirección a fichero "> fichero"
- ✔ Admite anexión a fichero ">> fichero"
- ✔ Admite entrada desde fichero "< fichero"
- ✔ Controla redirección a múltiples ficheros
- ✔ Admite redirección desde descriptor de fichero "2> fichero" (p. ej. para errores)
- ✔ Admite pasar comandos a segundo plano "&"
- ✔ Prompt personalizado: muestra el login, el hostname y la carpeta actual
- ✔ Comando ls con color (opción –color)
- ✔ Se admiten tres tipos de ristras:
  - Comillas dobles ""
  - Comillas simples "
  - Formato ANSI-C $", con control de caracteres de escape
- ✔ Análisis léxico y sintáctico con herramientas profesionales (Flex y Bison)
- ✔ Admite creación de alias
- ✔ Admite caracteres comodín (* y ?)
- ✔ Admite envío de señales a los comandos lanzados (pasa señal SIGINT, de Ctrl+C, a los hijos)
- ✔ Cambio de directorio con todas las opciones posibles (según las del Bash)
- ✔ Evita que se mate al shell con killall desde él mismo
- ✔ Admite modo depuración independiente para:
  - el análisis léxico

- el análisis sintáctico
- los alias
- las estructuras

Diagrama general de bloques

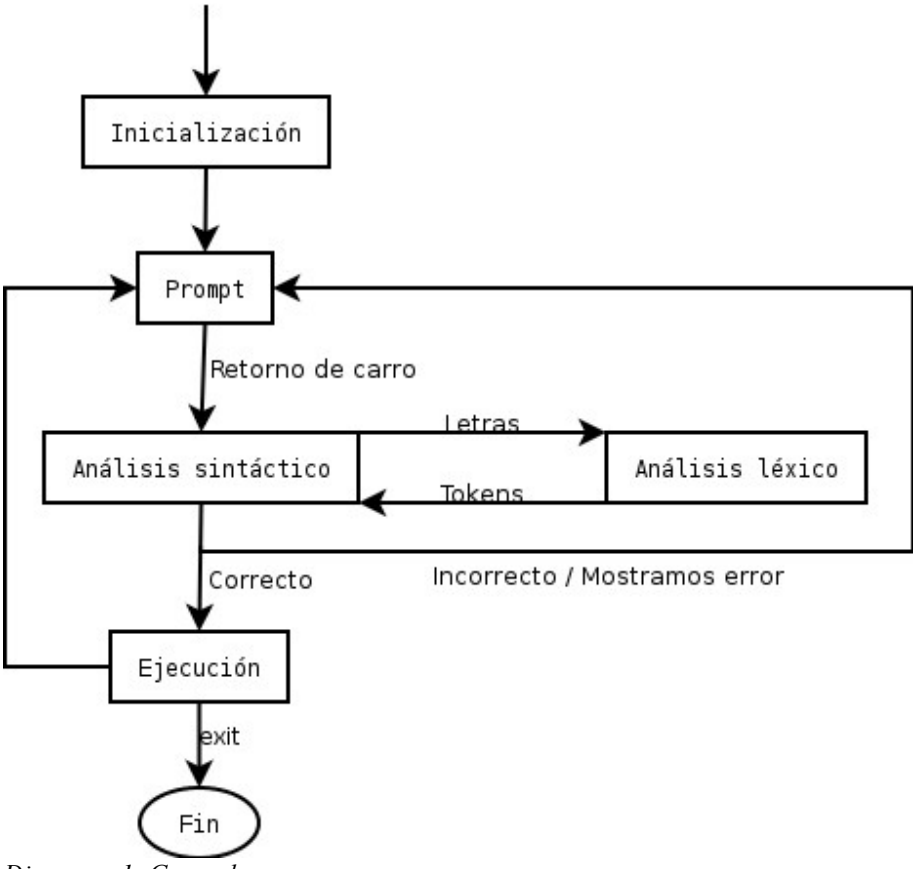

Diagrama 1: General

Una vez realizadas las inicializaciones pertinentes, mostramos el *prompt* a la espera de que el usuario introduzca la sentencia que se debe ejecutar. Dicha sentencia se termina cuando el usuario introduce el salto de línea. A continuación, se comprueba la correcta sintaxis de la sentencia introducida por el usuario. En caso incorrecto, se muestra un error y se vuelve a esperar a que el usuario introduzca la sentencia a ejecutar. En caso correcto, se pasa a la siguiente fase: la ejecución. Una vez se haya ejecutado el comando, se vuelve a esperar a que el usuario introduzca la sentencia a ejecutar. Si la sentencia fuera un "exit", saldríamos del programa.

Distribución de los ficheros

Lista de ficheros

ejecucion.c/h	Implementación de la ejecución de órdenes y alias.
lex.yy.c	Implementación del analizador léxico (creado por Flex).
lexico.l	Descripción en Flex del analizador léxico.
Makefile	Descripción de órdenes para la utilidad make.
shell	Nuestro shell (código máquina).
sintactico.tab.c/h	Implementación del analizador sintáctico (creador por Bison).
sintactivo.y	Descripción en Bison del analizador sintáctico.
structs.c/h	Estructuras para la ejecución de órdenes y alias y funciones de prueba.

ejecucion.h

El módulo **ejecucion** es el que se encarga de la ejecución de los comandos, apoyándose en la estructura **Orden** fundamentalmente. Proporciona la función ejecutar desde la interfaz, para que se use en el **main** presente en el fichero **sintactico.y**.

```
int ejecutar(struct Orden *o,struct Alias *alias);
```

ejecucion.c

Además de la función **ejecutar** se implementa de forma auxiliar la función **ejecutarComando** que simplemente realiza la ejecución de un único comando, con las salidas y entradas ya fijadas correctamente (en el caso de redireccionamiento).

```
int ejecutarComando(struct Orden *o) {
	// Ejecución del comando (se busca en las rutas del sistema)
	int i = 0;
	struct Orden *aux = (struct Orden *)malloc(sizeof(struct Orden));
inicializarOrden(aux);
	if(strcmp(o->comando,"ls") == 0) insertarArgumento(aux,"--color");
	execvp(o->comando,vectorArgv(o,aux->argumentos));
	printf("Comando %s no encontrado\n",o->comando);
```

```
    exit(1);
}
```

Realiza la ejecución con la llamada al sistema **execvp** que requiere un vector de argumentos, que se construye con la función **vectorArgv**, que en el caso de ejecutarse el comando ls también añadirá el argumento **--color** a dicho vector. Además, **execvp** permite que el comando se busque en todo el path, de modo que no supone ningún problema la búsqueda de los programas situados en los directorios del sistema. Si **execvp** fallará se muestra un error indicando que el comando no se ha encontrado y se finaliza el programa con **exit(1)**, ya que es un proceso hijo del padre (que es el propio shell) el que ejecuta el comando, es decir, el que ejecuta esta función.

Por otro lado se tiene la propio funcion **ejecutar** de la que se explican a continuación todos sus pasos. Sus variables locales son las siguientes:

```
    int estado;
    struct Orden *ordenActual;
    int salida, entrada, anexar;
    int salidaAnterior, entradaAnterior, anexarAnterior;
    int tuberia[2];
    int tuberiaAnterior = -1;
```

La variable **estado** simplemente toma el estado de finalización del proceso hijo que realiza la ejecución de una orden en concreto, el cual en realidad no se usa con ninguna intención especial.

Con **ordenActual** se consigue tener un puntero a la orden que hay que ejecutar en cada iteración del bucle que recorre todas las órdenes que tiene la orden a ejecutar.

Para controlar las redirecciones se tienen los ficheros de **salida**, **entrada** y **anexar**, así como los anteriores (**salidaAnterior**, **entradaAnterior** y **anexarAnterior**) para permitir el intercambio de salidas y entradas cuando se tienen comandos concatenados mediante pipes; no quiere decir que las redirecciones intervengan en los pipes, sino que estas variables son para controlar la entrada y salida que usa el comando a ejecutar, lo cual se maneja tanto en el proceso de redirección como en el ejecución de comandos en que hay pipes. De hecho, para los pipes se usa la variable **tuberia[2]**, que es la que permite tener el descriptor de entrada y salida al pipe. Igualmente, se controla la **tuberiaAnterior** para poder hacer uso de múltiples comandos concatenados con pipe, de forma que se van alternando los descriptores de salida y entrada, pues la salida del comando anterior será la entrada del comando actual, haciéndolo con el pipe.

Las primeras tareas consisten en el tratamiento de los alias. En primer lugar se trata de comprobar si lo que se pretende es ver todos los alias, crear uno nuevo o eliminar uno existente, lo cual se hace respectivamente con comandos de la forma **alias**, **alias NOMBREALIAS=COMANDO** y **unalias NOMBREALIAS**.

```
	// Muestreo de todos los alias o Creación de un alias
	if(strcmp(o->comando,"alias") == 0){
		if(o->argumentos){
			if((!o->argumentos->siguiente) ||
			  (!o->argumentos->siguiente->siguiente) ||
			  (strcmp(o->argumentos->siguiente->valor,"=") != 0)) printf("Declaración de alias incorrecta\n");
			else insertarAlias(a,o->argumentos->valor,strcat(o->argumentos->siguiente->siguiente->valor,"\n"));
		}else imprimirAlias(a);
		return 0;
	}

	// Eliminación de un alias
	if(strcmp(o->comando,"unalias") == 0){
		if(o->argumentos) eliminarAlias(a,o->argumentos->valor);
		return 0;
	}
```

Seguidamente, antes de la ejecución de cualquier comando hay que comprobar que no se trata de un alias, por lo que hay que realizar una búsqueda del comando en la estructura de alias. En la implementación actual sólo se permite que el alias se ponga como el primer comando de toda la expresión.

De este modo, si **buscarAlias** se verifica, se obtiene el puntero al alias en cuestión en **aliasAux**. A continuación se procede a expandir el alias. Para la expansión del alias se copiar la orden actual **o** en **ordenAux** para proceder a liberar la orden **o** y realizar el análisis sintáctico de la orden almacenada como ristra en la estructura **Alias** del alias actual (contenido en **aliasAux**).

Para hacer posible el análisis sintáctico mencionado, se requiere modificar la entrada yyin del analizador sintáctico, de forma que la nueva entrada será un **FICHEROAUXILIAR** definido como una macro (#define FICHEROAUXILIAR "_orden_auxiliar_", fichero que no debería existir, pues se perdería). Se salva la entrada actual en **yyinAnterior** y **yyin** recibe el descritptor del fichero auxiliar

mencionado, en el que se pone el comando almacenado como ristra en el alias (situando el cursor al inicio del fichero con **fseek**). Luego se realiza el análisis sintáctico con **yyparse()**. Hecho esto se cierra el fichero auxiliar y se borra. Finalmente, se restuara la entrada del analizador sintáctico y se hace la fusión de la orden a la que equivalía el alias con la orden parseada anteriormente para el alias, que pudiera contener parámetros adicionales, todo ello con **fusionarOrden**. Luego puede liberarse la memoria ocupada por **ordenAux**, que ya no es necesaria.

```
        // Expansión de un alias (para construir la orden y ejecutarla)
        struct Alias *aliasAux = (struct Alias *)malloc(sizeof(struct Alias));
        if(aliasAux = buscarAlias(a,o->comando)){
                struct Orden *ordenAux = (struct Orden *)malloc(sizeof(struct Orden));
                copiarOrden(ordenAux,o);
                liberarOrden(o); o = (struct Orden *)malloc(sizeof(struct Orden));
inicializarOrden(o);


                FILE *yyinAnterior = yyin;
                yyin = fopen(FICHEROAUXILIAR,"w+");     // Se escribe la orden en FICHEROAUXILIAR
                fprintf(yyin,aliasAux->comando);
                fseek(yyin,0,SEEK_SET);
                yyparse();                                      // Se parsea la orden del
FICHEROAUXILIAR
                fclose(yyin);
                remove(FICHEROAUXILIAR);
                yyin = yyinAnterior;                    // Se restaura la entrada de parseo


                fusionarOrden(o,ordenAux); liberarOrden(ordenAux);
        }
```

En este punto, ya se tiene una orden de forma correcta, susceptible de ejecutarse. Como se puede componer de múltiples órdenes o comandos, separados por pipe o punto y coma, se realiza el control de la orden actual a ejecutar con la variable **ordenActual**, que se inicializa a **o**:

```
ordenActual = o;
```

Seguidamente se entra en el bucle de ejecución de todas las órdenes:

```
while (ordenActual != NULL) {...}
```

Los pasos realizados en cada iteración dependerán de los campos de la orden. A continuación se explica cada uno de los mismos, que en función de dichos campos, las condiciones decidirán si

se realizan o no.

En primer lugar, se crean los fichero de salida y anexar, lo cual es útil cuando son más de uno, para que se creen todos, si bien sólo se hará efectiva la redirección a uno de ellas (el último en la línea de comando); esto hace que el proceso sea el mismo que el del Bash.

```
crearFicherosOrden(ordenActual);
```

A continuación se crean las tuberías, en la primera iteración sólo podrá ser de salida, pero la implementación se realiza para que funcione para todas las iteraciones. En primer lugar se trata la tubería de entrada. Como ya se ha mencionado, no procede para el primer comando, por lo que **tuberiaAnterior** se inicializó a -1, de forma que nunca se ejecuta este código para le primera iteración.

En sucesivas iteraciones, es decir, si hay varios comandos y la concatenación fue con un pipe, en el comando anterior se habrá abierto un pipe para que la salida del comando se deposite en el mismo (y **tuberiaAnterior** ya no valdrá -1). Ahora se tomará como entrada la salida de dicha tubería anterior, siempre y cuando el comando actual no tenga una redirección de un fichero de entrada.

Para la tubería de entrada, la tarea a realizar consiste en duplicar la entrada estándar (**STDIN_FILENO**) y almacenarla en **entradaAnterior**. Seguidamente, con **dup2**, se realiza el cambio de la entrada, que ahora será **tuberiaAnterior**. Luego se cierra **tuberiaAnterior**.

```
// Tubería de entrada
if (tuberiaAnterior != -1 && !ordenActual->fentradaActual) {
        entradaAnterior = dup(STDIN_FILENO);
        dup2(tuberiaAnterior,STDIN_FILENO);
        close(tuberiaAnterior);
}
```

Para la tuebería de salida será la que verdaderamente realiza la creación del pipe con la función **pipe**. Se creará la tubería simpre que haya una orden después de la actual y sea con un pipe, lo cual se puede ver en el campo **pipe** de la **ordenActual**. No obstante, existe una excepción, es decir, si existe un fichero de salida o de anexar para la orden actual, su salida no se pasará al siguiente comando, sino que se redireccionará al fichero de salida o anexar. En tal caso no se crea el pipe.

Al crear la tubería el proceso consiste en sustituir la salida estándar (**STDOUT_FILENO**). La

salidaAnterior será precisamente la salida estándar, que se toma con **dup**. Luego, con **dup2** se hace que la nueva salida sea **tuberia[1]** y luego ya puede cerrarse **tuberia[1]**.

Finalmente, a **tuberiaAnterior** se le asigna **tuberia[0]** que será la entrada para el siguiente comando, que se fijará en la siguiente iteración. Si no hay pipe se pone a -1 el valor de **tuberiaAnterior**, pues de lo contrario podría haber problemas si aparece algún pipe después de punto y coma.

```
            // Tubería de salida
            if (ordenActual->pipe && !(ordenActual->fsalidaActual || ordenActual->fanexarActual)) {
                        pipe(tuberia);
                        salidaAnterior = dup(STDOUT_FILENO);
                        dup2(tuberia[1],STDOUT_FILENO);
                        close(tuberia[1]);
                        tuberiaAnterior = tuberia[0];
            }else tuberiaAnterior = -1;
```

A continuación, se procede al tratamiento de las redirecciones. En primer lugar se tiene le redireccionamiento de salida que se hará si la orden tiene fichero de salida, lo cual se mira y toma del campo **fsalidaActual** que es el efectivo para el redireccionamiento.

Como el redireccionamiento de salida permite el redireccionamiento desde un descriptor de fichero, se duplica el fichero cuyo descriptor será **fdActual** (campo de la estructura **ordenActual**). Luego, éste se salvará en **salidaAnterior**.

La nueva **salida** será el descriptor del fichero **fsalidaActual** que se abrirá con permiso de escritura (si no existe se creará); su contenido se trunca, es decir, no se anexa.

Finalmente se cierra el descriptor de fichero que es la salida actual para una determinada funcionalidad, como puede ser el redireccionamiento de errores en el descriptor de fichero 2. Luego se hace efectivo el cambio de la salida con **dup2** haciendo que ahora sea **salida**.

```
            // Redireccionamiento de salida
            if (ordenActual->fsalidaActual) {
                        salidaAnterior = dup(o->fdActual);
                        if ((salida = open(ordenActual->fsalidaActual, O_WRONLY | O_TRUNC | O_CREAT, S_IRUSR | S_IWUSR)) == -1) {
                                    printf("No se puede abrir %s para escritura\n", ordenActual->fsalidaActual);
```

```
                    return -2;
                }
                close(o->fdActual);
                dup2(salida,o->fdActual);
            }
```

Para el redireccionamiento de anexar el proceso es el mismo, salvo que ahora se abre el fichero anexar en modo **anexar** (**O_APPEND**). Además, las variables varían, pues ahora se tiene **anexarAnterior** para el descriptor de fichero anterior y **anexar** para el fichero. Además, se mira si hay que anexar en el campo **fanexarActual**.

```
            // Redireccionamiento de anexar
            if (ordenActual->fanexarActual) {
                anexarAnterior = dup(o->fdActual);
                if ((anexar = open(ordenActual->fanexarActual, O_WRONLY | O_APPEND |
O_CREAT, S_IRUSR | S_IWUSR)) == -1) {
                    printf("No se puede abrir %s para anexar\n", ordenActual-
>fanexarActual);
                    return -2;
                }
                close(o->fdActual);
                dup2(anexar,o->fdActual);
            }
```

Finalmente, para el redireccionamiento de entrada se procede de forma similar. La diferencia es que ahora la entrada a cambiar siempre es la misma (**STDIN_FILENO**), de forma que se salva en **entradaAnterior**. Luego se abre como lectura el fichero **entrada**. Luego se cierra la entrada estándar **STDIN_FILENO** y realiza el cambio de la entrada, con **dup2**, pasándola a **entrada**.

```
            // Redireccionamiento de entrada
            if (ordenActual->fentradaActual) {
                entradaAnterior = dup(STDIN_FILENO);
                if ((entrada = open (ordenActual->fentradaActual, O_RDONLY)) == -1) {
                    printf("No se puede abrir %s para lectura\n", ordenActual-
>fentradaActual);
                    return -2;
                }
                close(STDIN_FILENO);
                dup2(entrada,STDIN_FILENO);
```

```
        }
```

Ya se puede realizar la ejecución, pero existen comandos que son propios del shell, de forma que no existen programas externos que hagan la tarea de dichos comandos. Por ello hay que introducir código para ellos. Es el caso del comando **cd** usado para cambiar de directorio. Si el comando es **cd** se analizarán sus argumentos para ejecutar el comando correctamente.

Para conocer el directorio actual se usa la función **getcwd**, que permite conocer el **directorioAnterior** y el **directorio** actual. Para realizar el cambio de directorio se usa la función **chdir**. Por lo general, el argumento del comando **cd** indica el directorio al que ir, pero adicionalmente es posible indicar argumentos especiales cuya semántica se comenta en la siguiente tabla.

Argumento	*Significado*
~	Ir al directorio personal (**HOME**)
--	Ir al directorio personal (**HOME**)
-	Ir al directorio anterior

Para ir al directorio personal, se toma el mismo a partir de la variable de entorno **HOME** mediante la función **getenv**. Si dicha variable no estuviera fijada se irá al directorio raíz /.

```
            // Ejecución de un comando con argumentos
            if (strcmp(ordenActual->comando,"cd") == 0) {
                // Comandos no existentes como programas
                // cd (Cambiar de directorio). Si hay más de un argumento, se ignora
                char *dir = (char *)malloc(sizeof(char)*strlen(directorioAnterior));
strcpy(dir,directorioAnterior);
                getcwd(directorioAnterior,MAX_DIR);
                // Ir a directorio personal (Si no hay directorio personal, se va a la raíz)
                if (ordenActual->argumentos == NULL ||
                    (strcmp(ordenActual->argumentos->valor,"~") == 0) ||
                    (strcmp(ordenActual->argumentos->valor,"--") == 0))
chdir(getenv("HOME")!=NULL?getenv("HOME"):"/");
                // Ir a directorio anterior
                else if (strcmp(ordenActual->argumentos->valor,"-") == 0) chdir(dir);
                // Ir al directorio indicado
                else chdir(ordenActual->argumentos->valor);
                ...
```

Si el comando es **exit**, no se ejecutará, sino que se abandonará la ejecución y saldrá del programa directamente.

```
}else if (strcmp(ordenActual->comando,"exit") == 0) {
    // exit (Cerrar el shell)
    return -1;
```

Si el comando es un **killall shell**, es decir, se intenta matar al shell desde él mismo, se ignora, para evitar que dicho suceso ocurra; no obstante, existen otras forma de matar al shell que no se controlan, lo cual también ocurre en el Bash.

```
}else if ((strcmp(ordenActual->comando,"killall") == 0) &&
        (strcmp(ordenActual->argumentos->valor,"shell") == 0)) {
    // Se intenta matar al shell (killall shell), se ignora
```

Si no se da uno de los comandos anteriores, se procederá a la ejecución del comando. Dicha ejecución la realizará un proceso hijo que se creará con **fork()**. El hijo simplemente ejecuta el comando llamando a **ejecutarComando**, mientras que el padre (el propio shell) esperará a que el hijo termine, a no ser que el comando se lanza en segundo plano o modo tanto, en cuyo caso se el padre hace el **waitpid** con la opción **WNOHANG**.

```
}else {
    // No se hace nada si no se indicó ningún comando ("")
    // Comandos existentes como programas
    switch(pid = fork()){
        case -1:        // Fallo
            printf("Fallo al crear un nuevo proceso con fork()\n");
            return -2;
        case 0:         // Hijo
            ejecutarComando(ordenActual);
        default:        // Padre
            waitpid(pid,&estado,ordenActual->segundoplano?WNOHANG:0);
    }
}
```

Tras la ejecución del comando, se procede a la restauración de la entrada y salida estándar. En primer lugar se restaura la entrada si fue modificada en el caso de un pipe (por ello **entradaAnterior** no valdrá -1, pero no debe haber fichero de entrada **fentradaActual**). Para ello se hace que la **entradaAnterior** pase a ser la entrada estándar (**STDIN_FILENO**), con **dup2** y luego

se resetea a -1 la **entradaAnterior**.

```
if (entradaAnterior != -1 && !ordenActual->fentradaActual) {
    dup2(entradaAnterior,STDIN_FILENO);
    entradaAnterior = -1;
}
```

Si hay **pipe** y no hay redirección de salida ni anexar, se procede a restuarar la **salidaAnterior** a la salida estándar (**STDOUT_FILENO**), con **dup2**.

```
if (ordenActual->pipe && !(ordenActual->fsalidaActual || ordenActual->fanexarActual)) {
    dup2(salidaAnterior,STDOUT_FILENO);
}
```

Seguidamente se restauran las salidas en función de los redireccionamientos. En todos los casos se cierra el descriptor de salida o entrada actual, que en el caso de la salida y anexar puede ser cualquier descriptor de fichero (almacenado en **fdActual**), y en el caso de la entrada es **STDIN_FILENO**. Luego se cierra la **salida**, el fichero de **anexar** o el de **entrada**, según el caso, para finalmente, con **dup2**, restaurar la salida o entrada.

```
if (ordenActual->fsalidaActual) {
    close(o->fdActual);
    close(salida);
    dup2(salidaAnterior,o->fdActual);
}
if (ordenActual->fanexarActual) {
    close(o->fdActual);
    close(anexar);
    dup2(anexarAnterior,o->fdActual);
}
if (ordenActual->fentradaActual) {
    close(STDIN_FILENO);
    close(entrada);
    dup2(entradaAnterior,STDIN_FILENO);
}
```

El siguiente paso consiste en evitar bloqueos de comandos concatenados con pipe, pero que en el comando previo se hace redireccionamiento de salida. Dicho comando previo, en realidad es el actual, y para evitar el bloqueo se procede de la siguiente forma: se crea un pipe, cerrando la

tubería de salida **tuberia[1]** y poniendo en **tuberiaAnterior** la de entrada (**tuberia[0]**).

```
            if(ordenActual->pipe && (ordenActual->fsalidaActual || ordenActual->fanexarActual)) {
                        pipe(tuberia);
                        close(tuberia[1]);
                        tuberiaAnterior = tuberia[0];
            }
```

Para terminar la iteración del bucle, se avanza a la siguiente orden o comando:

```
ordenActual = ordenActual->siguiente;
```

lexico.l

Las librerías a utilizar y el código en C que se requiera se pone entre "%{" y "}%".

```
%{
//#define DEBUG_LEXICO
#include "sintactico.tab.h"
#include <unistd.h>
%}
```

DEBUG_LEXICO indica si queremos que se muestren mensajes de depuración para el léxico. A lo largo del código del analizador léxico hemos puesto varios #IFDEF DEBUG_LEXICO tal que en caso de estar activado se muestren mensajes de depuración para cada caso.

A continuación tenemos definidas las variables o directivas especiales, que son las declaraciones preliminares. Son una ayuda que proporciona flex a la hora de definir las expresiones regulares. Los normal es definir directivas especiales simplemente, pero también se puede poner indicadores especiales.

En nuestro caso hemos definido lo que sería una variable, con el nombre NUMERO, para definir la expresión regular de un número entero. Allí donde se use, en las expresiones regulares, se expanderá su valor.

```
NUMERO ([1-9][0-9]*)|0
```

Podemos controlar el tipo de implementación de la variable que almacena el texto que se toma de la entrada. Dicha variable la define flex com yytext. Se tratará de un vector de caracteres. Podemos indicar si se debe implementar como un vector estático o dinámico, es decir, como un puntero. Queremos que sea un puntero, para que se admita cualquier tamaño para la entrada (la entrada se hace en el momento en que se pulsa ENTER al introducir todo un comando al shell). Lo

indicamos de la siguiente forma:

```
%pointer
```

Finalmente, para expresiones regulares especialmente complejas, se requiere el uso de estados, como si se tratara de un autómata. Es el caso de las ristras, que al admitir secuencias de escape, que deben traducirse por el valor que representan, se requiere ir pasando de un estado a otro para poder realizar dicha tarea. Para ello se tienen estados que representa la reducción a una expresión regular, que se hará por trozos. Se tendrá una entrada y un fin, y pasos intermedios; esto se verá más adelante.

Como hay tres tipos de ristra, se requerirán tres estados, y adicionalmente otro para los identificadores, pues también permite ciertos caracteres de escape como para el carácter blanco (\ ).

```
%x comillasdobles
%x comillassimples
%x comillassimplesansic
%x identificador
```

Para separar las declaraciones preliminares que hemos hecho de las reglas usamos los símbolos %%.

```
%%
```

A continuación tenemos las reglas, cuya estructura general es:

```
expresión_regular { código en C a ejecutar si se cumple la expresión regular }
```

Además, tenemos en yylval.valor el valor que ha disparado la regla, tal que lo podemos almacenar en la estructura oportuna.

Iremos comentando las expresiones regulares una por una y el código C que se genera para construir y devolver los tokens al analizador sintáctico.

Expresión Regular para ristras con Comillas Dobles

Como admite secuencias de escape, se requiere el uso de un estado por el que se pasará por diversos subestados, para poder convertir las secuencias de escape en los valores ASCII que representan.

Las ristras de comillas dobles comenzarán por comillas dobles: **"**.

```
\"              {
```

```
yylval.valor = (char *)malloc(sizeof(char));
strcpy(yylval.valor,"");
BEGIN(comillasdobles);
}
```

Al detectarlas se entra en el estado **comillasdobles**, con la instrucción **BEGIN(comillasdobles);**. Además, se inicia el proceso de almacenamiento de la ristra en yylval.valor, que se inicializa a la ristra vacía: **""**.

Para la secuencia de escape de las comillas dobles (**"**) y la del salto de línea (**SALTO_DE_LÍNEA**) se usa la siguiente expresión regular.

```
<comillasdobles>\\(\"|\n) {
if(yylval.valor) yylval.valor = (char *)realloc(yylval.valor,sizeof(char)*(strlen(yylval.valor)+2));
else yylval.valor = (char *)malloc(sizeof(char)*2);
strcat(yylval.valor,++yytext);
}
```

Por secuencia de escape del salto de línea se permite que se ponga la barra \ y luego se pulse **ENTER** para escribir un comando en varias líneas. Se almacenan las comillas dobles (**"**) y el salto de línea (\n), respectivamente.

El resto de secuencias de escape no son tales, es decir, no se hace ninguna conversión, sino que se ignora la barra. Esto quiere decir, que lo que se hace es almacenar tanto la barra como lo que le precede. La expresión regular es \\. porque indica la aparación de una barra \ y cualquier caracter (**.**).

```
<comillasdobles>\\. {
if(yylval.valor) yylval.valor = (char *)realloc(yylval.valor,sizeof(char)*(strlen(yylval.valor)+3));
else yylval.valor = (char *)malloc(sizeof(char)*3);
strcat(yylval.valor,yytext);
}
```

Si apararece un salto de línea se almacena tal cual, pues se permite que una ristra tenga saltos de línea y se termine en la siguiente línea.

```
<comillasdobles>\n {
if(yylval.valor) yylval.valor = (char *)realloc(yylval.valor,sizeof(char)*(strlen(yylval.valor)+2));
else yylval.valor = (char *)malloc(sizeof(char)*2);
strcat(yylval.valor,yytext);
```

```
}
```

Cuando se encuentren comillas dobles (**"**) de nuevo, se terminará la ristra, de modo que se termina el estado y se vuelve al estado inicial o por defecto: **BEGIN(INITIAL);**. Además, se devuelve el token, que será un **IDENTIFICADOR**.

```
<comillasdobles>\" {
strcat(yylval.valor,"\0");
#ifdef DEBUG_LEXICO
        printf("Ristra de comillas dobles: %s\n",yylval.valor);
#endif
BEGIN(INITIAL);
return IDENTIFICADOR;
}
```

Cualquier otro caracter se almacena como parte de la ristra. Esto quiere decir, cualquier otro caracter salvo la barra \, el salto de línea **n** y las comillas dobles **"**, que se representa como **[**^**n****"]**, en la expresión regular.

```
<comillasdobles>[^\\\n\"]+ {
if(yylval.valor) yylval.valor = (char *)realloc(yylval.valor,sizeof(char)*(strlen(yylval.valor)+yyleng+1));
else yylval.valor = (char*)malloc(sizeof(char)*(yyleng+1));
strcat(yylval.valor, yytext);
}
```

Expresión Regular para ristras con Comillas Simples

Las secuencias de escape no se admiten para la comillas simples, ni tampoco los saltos de línea. No obstante, también se hace uso de un estado, de modo que al leerse un comilla simple **'** se entra en el estado **comillassimples**.

```
\' {
yylval.valor = (char *)malloc(sizeof(char));
strcpy(yylval.valor,"");
BEGIN(comillassimples);
}
```

Cuando se vea otro comilla simple se devuelve la ristra almacenada y se pasa al estado **INITIAL**. Y se devuelve el token **IDENTIFICADOR**.

```
<comillassimples>\' {
```

```
strcat(yylval.valor,"\0");
#ifdef DEBUG_LEXICO
		printf("Ristra de comillas simples: %s\n",yylval.valor);
#endif
BEGIN(INITIAL);
return IDENTIFICADOR;
}
```

El resto de caracteres, salvo la comilla simple, se almacenan en la ristra.

```
<comillassimples>[^']+ {
if(yylval.valor) yylval.valor = (char *)realloc(yylval.valor,sizeof(char)*(strlen(yylval.valor)+yyleng+1));
else yylval.valor = (char*)malloc(sizeof(char)*(yyleng+1));
strcat(yylval.valor,yytext);
}
```

Expresión Regular para ristras ANSI-C

Las ristras ANSI-C son las más versátiles en la conversión y admisión de secuencias de escape. Se requerirá un estado llamado **comillassimlesansic**, que se inicia con **$'**.

```
\$\' {
yylval.valor = (char *)malloc(sizeof(char));
strcpy(yylval.valor,"");
BEGIN(comillassimplesansic);
}
```

Las secuencias de escape de números octales se tratan admitiendo números octales de 1 a 3 cifras (cada cifra toma un valor de 0 a 7). Con la función sscanf se convierte a octal la ristra que representa el octal y luego se almacena en la ristra.

```
<comillassimplesansic>\\[0-7]{1,3} {
int aux; sscanf(yytext+1,"%o",&aux);
yylval.valor = (char *)realloc(yylval.valor,sizeof(char)*(strlen(yylval.valor)+2));
int l = strlen(yylval.valor); yylval.valor[l] = aux; yylval.valor[l+1] = '\0';
}
```

Las secuencias de escape de números hexadecimales se tratan admitiendo números hexadecimales de 1 a 2 cifras (cada cifra toma un valor de 0 a 9, de la a a la f ó de la A a la F). Con la función sscanf se convierte a hexadecimal la ristra que representa el hexadecimal y luego se

almacena en la ristra.

```
<comillassimplesansic>\\x[0-9a-fA-F]{1,2} {
int aux; sscanf(yytext+2,"%x",&aux);
yylval.valor = (char *)realloc(yylval.valor,sizeof(char)*(strlen(yylval.valor)+2));
int l = strlen(yylval.valor); yylval.valor[l] = aux; yylval.valor[l+1] = '\0';
}
```

El resto de secuencias de escape se tratan sin expresión regular por su mayor comodidad. Se trata de las secuencias de escape especiales: **n**, **t**, **r**, **b** y **f**. El resto de secuencias de escape, que no serían tales, se dejan sin modificación, almacenándose la barra y el valor que le sigue.

```
<comillassimplesansic>\\. {
if(yylval.valor) yylval.valor = (char *)realloc(yylval.valor,sizeof(char)*(strlen(yylval.valor)+2));
else yylval.valor = (char *)malloc(sizeof(char)*2);
if (strcmp(yytext,"\\n") == 0) strcat(yylval.valor,"\n");
else if(strcmp(yytext,"\\t") == 0) strcat(yylval.valor,"\t");
else if(strcmp(yytext,"\\r") == 0) strcat(yylval.valor,"\r");
else if(strcmp(yytext,"\\b") == 0) strcat(yylval.valor,"\b");
else if(strcmp(yytext,"\\f") == 0) strcat(yylval.valor,"\f");
else strcat(yylval.valor,yytext);
}
```

Para la secuencia de escape de las comillas simples \', se toma la comilla simple.

```
<comillassimplesansic>[^\\']+ {
if(yylval.valor) yylval.valor = (char *)realloc(yylval.valor,sizeof(char)*(strlen(yylval.valor)+yyleng+1));
else yylval.valor = (char*)malloc(sizeof(char)*(yyleng+1));
strcat(yylval.valor,yytext);
}
```

Cuando llega la comilla simple, se finaliza la ristra y se pasa al estado **BEGIN**. Se devuelve el token **IDENTIFICADOR**.

```
<comillassimplesansic>\' {
strcat(yylval.valor,"\0");
#ifdef DEBUG_LEXICO
        printf("Ristra de comillas simples ANSI-C: %s\n",yylval.valor);
#endif BEGIN(INITIAL);
return IDENTIFICADOR;
}
```

Expresión Regular para espacios blancos y tabuladores

Se trata de ignorar los espacios en blanco y tabuladores (que pueden ser múltiples).

```
[ \t]+ {
#ifdef DEBUG_LEXICO
		printf("Blanco o Tabulador\n");
#endif
}
```

Expresión Regular para Redireción de Anexar

Se permite que con >> se indique el redireccionamiento de anexar, pero además se permite que lo que se anexe venga desde un descriptor de fichero indicado como un número entero (**NUMERO**).

Para saber si aparece algún número de descriptor de fichero se mira la longitud de lo leído en **yytext**, que se puede mirar directamente en **yyleng**. Si es mayor que dos, tomaremos su valor entero (usando la función **atoi**), que es el descriptor de fichero desde el que se hace la redirección de anexar. Si es menor que dos, el redireccionamiento es desde la salida estándar (**STDOUT_FILENO**). Tanto el uno como el otro se almacenan en el campo ficheroRedireccion de **yylval**. Además, se devuelve el token **REDIRECCIONANEXAR**.

```
{NUMERO}?\>> {
#ifdef DEBUG_LEXICO
		Oprintf("Signo: %s\n",yytext);
#endif
if(yyleng > 2){
		yylval.valor = (char*)malloc(sizeof(char)*yyleng-1);		// Se ignora el signo '>>'
		yytext[yyleng-2] = 0;
		yylval.ficheroRedireccion = atoi(yytext);
}else yylval.ficheroRedireccion = STDOUT_FILENO;
return REDIRECCIONANEXAR;
}
```

Expresión Regular para Redireción de Salida

Ocurre lo mismo que con la redirección de anexar, sólo que ahora se redirecciona con >. Además, ahora el token a devolver es **REDIRECCIONSALIDA**.

```
{NUMERO}?\> {
#ifdef DEBUG_LEXICO
printf("Signo: %s\n",yytext);
#endif
if(yyleng > 1){
		yylval.valor = (char*)malloc(sizeof(char)*yyleng); // Se ignora el signo '>'
		yytext[yyleng-1] = 0;
		yylval.ficheroRedireccion = atoi(yytext);
}else yylval.ficheroRedireccion = STDOUT_FILENO;
return REDIRECCIONSALIDA;
}
```

Expresión Regular para Redireción de Entrada

El redireccionamiento de entrada, con **<**, no admite la redirección desde un descriptor de fichero. En este caso simplemene se devuelve el token, que es **REDIRECCIONENTRADA**.

```
\< {
#ifdef DEBUG_LEXICO
		printf("Signo: <\n");
#endif
return REDIRECCIONENTRADA;
}
```

Expresión Regular para Pipe

El pipe (|) devolverá el token **PIPE**.

```
\| {
#ifdef DEBUG_LEXICO
		printf("Signo: |\n");
#endif return PIPE;
}
```

Expresión Regular para Segundo Plano

El segundo plano (**&**) devolverá el token **SEGUNDOPLANO**.

```
\& {
#ifdef DEBUG_LEXICO
		printf("Signo: &\n");
```

```
#endif
return SEGUNDOPLANO;
}
```

Expresión Regular para Punto y Coma

El punto y coma (**;**) devolverá el token **PUNTOCOMA**.

```
\; {
#ifdef DEBUG_LEXICO
		printf("Signo: ;\n");
#endif
return PUNTOCOMA;
}
```

Expresión Regular para Salto de Línea

El salto de línea es analizado para devolver el token **SALTOLINEA** y el valor **"\n"** de forma que permite el comportamiento correcto ante errores sintácticos.

```
\n {
#ifdef DEBUG_LEXICO
		printf("Salto de línea\n");
#endif
// Permite comportamiento correcto ante errores sintácticos al final de la orden
yylval.valor = "\n";
return SALTOLINEA;
}
```

Expresión Regular para Identificador

Los identificadores deben tratarse con cuidado para admitir ciertas secuencias de escape, como la del espacio blanco. Se hará uso del estado **identificador**. El identificador puede empezar de dos formas. La primera de ellas consiste en que empieza por barra \, es decir, por una secuencia de escape. Se inicia el estado **identificador** y se almacena lo que venga después de la barra \, ya que en realidad no son secuencias de escape propiamente dichas.

```
\\. {
yylval.valor = (char*)malloc(sizeof(char)*2);
strcpy(yylval.valor,++yytext);
```

```
BEGIN(identificador);
}
```

La segunda forma de inicio de un identificador se produce cuando se empieza por un caracter distinto de blanco, tabulador (**t**), salto de línea (**n**), comillas dobles (**"**), comillas simples (\'), pipe (\|), punto y coma (**;**), redireccionamiento de entrada (\<), redireccionamiento de salida (\>), segundo plano (**&**), barra (\\).

```
[^ \t\n\"\'\|\;\<\>\&\\] {
yylval.valor = (char*)malloc(sizeof(char)*2);
strcpy(yylval.valor,yytext);
BEGIN(identificador);
}
```

Cualquier otro caracter de escape dentro del identificador se trata como el inicial, es decir, se toma lo que sigue a la barra.

```
<identificador>\\. {
yylval.valor = (char *)realloc(yylval.valor,sizeof(char)*(strlen(yytext)+yyleng));
strcat(yylval.valor,++yytext);
}
```

El resto de caracteres, salvo el blanco, tabulador (**t**), salto de línea (**n**), comillas dobles (**"**), comillas simples (\'), pipe (\|), punto y coma (**;**), redireccionamiento de entrada (\<), redireccionamiento de salida (\>), segundo plano (**&**), barra (\\), se toman tal cual.

```
<identificador>[^ \t\n\"\'\|\;\<\>\&\\]+ {
yylval.valor = (char*)realloc(yylval.valor,sizeof(char)*(strlen(yytext)+yyleng+1));
strcat(yylval.valor,yytext);
}
```

Cuando llegue alguno de los siguientes símbolos: blanco, tabulador (**t**), salto de línea (**n**), comillas dobles (**"**), comillas simples (\'), pipe (\|), punto y coma (**;**), redireccionamiento de entrada (\<), redireccionamiento de salida (\>), segundo plano (**&**), barra (\\), se terminará el identificador, devolviendo el token **IDENTIFICADOR**. Además, se pasa al estado **INITIAL**.

```
<identificador>[ \t\n\"\'\|\;\<\>\&\\] {
#ifdef DEBUG_LEXICO
        printf("Identificador: %s\n",yylval.valor);
#endif yyless(0);
BEGIN(INITIAL);
```

```
return IDENTIFICADOR;
}
```

Por último, si no se ha disparado antes ninguna regla, pillamos los demás casos, ilegales en nuestro léxico, y mostramos un mensaje de error (en realidad, no suele darse este caso, pero es una buena práctica de programación ponerlo).

```
. {
printf("CarÃicter %s ilegal\n",yytext);
}
```

Para terminar la sección de expresiontes, usamos los caracteres %%. Después podría ponerse código completamente escrito en lenguaje C, como podría ser el **main**, pero se pondrá en el analizador sintáctico hecho en Bison.

```
%%
```

Makefile

El Makefile está bien comentado por lo que es de fácil comprensión. Simplemente destacar que, en caso de no especificar regla en el make, se disparará la primera, que en nuestro caso es "shell", la cual compila la shell y la ejecuta (shell -> build_shell -> clean build-sintactico build-lexico). La última regla es "install", que copia nuestra shell en /bin/, aunque necesitamos los privilegios adecuados para ello.

sintactico.y

Su estructura es similar a la del fichero "lexico.l". Las librerías a utilizar y el código en C que se requiera se pone entre "%{" y "}%".

```
%{
#include <librería>
#define
declaraciones
}%
```

En nuestro caso se declaran las siguientes variables (algunos de ellas como **extern** porque se encuentran en otro módulos).

```
extern FILE *yyin;      // Declarado en lexico.l
```

```
extern char *yytext;   // Declarado en lexico.l
int yydebug = 1;              // Se habilita el modo debug si se usa -t
struct Orden *orden;   // Orden a ejecutar (analizada léxica y sintácticamente)
struct Alias *alias;   // Lista de alias
int valorError = 0;    // Indica si ha habido error de sintaxis
char *directorio;             // Directorios actual,
char *directorioAnterior;      // y anterior
int pid = 0;                  // pid del hijo actual, por si hay que enviarle alguna señal
```

El siguiente paso consiste en definir como será la variable **yylval** que el léxico podrá rellenar con información para el sintáctico. Su forma permite indicar el **valor** del token, y en caso de ser un redireccionamiento de salida o anexar, indicar desde que fichero se redirecciona, pues permite hacerlo desde cualquier descriptor de fichero.

```
%union {
        char *valor;
        int ficheroRedireccion;
}
```

A continuación definimos los tokens (símbolos terminales) a utilizar que tendrán que serán los que nos devuelva el léxico. Los tokens definidos son:

```
%token REDIRECCIONANEXAR
%token REDIRECCIONSALIDA
%token REDIRECCIONENTRADA
%token PIPE
%token SEGUNDOPLANO
%token PUNTOCOMA
%token SALTOLINEA
%token IDENTIFICADOR
```

Además, los tipos definidos en el union para **yylval** puede usarse para asignárselos a los token, de forma que podrán manejarse como si fueran de los tipos antes indicados. A continuación se indica que el token **IDENTIFICADOR** es de tipo **valor**, que en realidad es el tipo **char ***. Esto permite tratarlo como una ristra al estilo C. En el cas ode **REDIRECCIONANEXAR** y **REDIRECCIONSALIDA**, serán de tipo **ficheroRedireccion**, que es un **int**.

```
%type <valor> IDENTIFICADOR
%type <ficheroRedireccion> REDIRECCIONANEXAR
```

```
%type <ficheroRedireccion> REDIRECCIONSALIDA
```

Para separar las declaraciones preliminares que hemos hecho de las reglas usamos los símbolos %%.

```
%%
```

A continuación tenemos las reglas, que son del tipo:

```
NT: {NT | T} ['{' Código en C a ejecutar '}'] {| {NT | T} ['{' Código en C a ejecutar '}']} ';'
```

donde NT es un símbolo No Terminal y T es un símbolo Terminal definido en los tokens. Esta es una gramática independiente del contexto, que corresponde con el tipo 3 de la jerarquía de gramáticas de Chomsky.

Las reglas de la gramática se explican y muestran a continuación. En primer lugar se tendrá el símbolo no terminal principal (axioma), que es el que recoge toda la expresión que se introduzca en el shell. Se trata de una expresión terminada en un salto de línea. Dicha expresión será un comando (entendiéndolo como todo lo que hay que ejecutar) o bien nada, es decir, se admite que sólo se pulse **ENTER**.

```
expresion: SALTOLINEA {return -2;} | comando SALTOLINEA {return 0;}
;
```

Un salto de línea devuelve -2 para tratarlo de forma especial (no se ejecuta nada), mientras que el comando sí requiere su ejecución (devuelve 0).

Un **comando** puede reducir (es decir, tener un valor sintáctico en cuanto a reglas) de dos formas. Puede ser un **mandato_argumentos** (el nombre de un programa o comando y sus argumentos) y el **resto** (otro mando y sus argumentos después de un elemento de concatenación de comandos como el pipe, punto y coma, etc.), o bien, tener el indicador de **SEGUNDOPLANO** (**&**) antes del resto. En el caso de haber segundo plano, se indica en el campo **segundoplano** de la **ordenActual**, poniéndo su valor a 1 (true).

```
comando:
        mandato_argumentos resto
        | mandato_argumentos SEGUNDOPLANO {
                ordenActual(orden)->segundoplano = 1;
                #ifdef DEBUG_SINTACTICO
                        printf("Segundo plano (&)\n");
                #endif
```

```
            } resto
;
```

El **mandato_argumentos** puede reducirse de tres formas distintas:

- Un comando simple sin argumentos.
- Un comando con argumentos.
- Un comando con redirección.

En realidad, esto lo que permite es generar una recursividad de argumentos, pues se reduce a sí misma con **mandato_argumentos**. De esta forma, cuando se llega a **IDENTIFICADOR** se tiene el comando y se inserta un comando en la **ordenActual**. Como se admiten caracters comodín, es necesario tomar el comando **cmd** y expandir si fuera necesario. En caso de que haya expansión se toma la primera de ellas como hace el Bash (en tal caso **cmd->valor** se trata como true). Si no la hay, se toma simplemente el comando indicado. Para ello se usa **$1**, es decir, el primer símbolo de la opción, que en este caso es **IDENTIFICADOR**.

La forma en que se toman los símbolos terminales o no, en Bison, es como si fuera un pila, de modo que **$$** es el símbolo no terminal de la regla (**mandato_argumentos** en este caso) y **$1**, **$2**, etc., son los símbolos no terminales y terminales de cada opción (que se separan por |). Así, si se les asignó un tipo en las declaraciones preliminares, se pueden manejar como tales, referenciándolos con el formato indicado.

Al **insertarComando** se crea un nuevo comando para la orden en general, pues puede componerse de múltiples comandos.

En el caso de la segunda opción: **mandato_argumentos IDENTIFICADOR**, también se tiene en cuenta la expansión para caracteres especiales. En este caso, si hay expansión se toman todos sus valores y no sólo el primero, de modo que se tiene un bucle para la inserción de todos los argumentos: **insertarArgumento**. Si no hay expansión (no hay caracteres comodín), se inserta el argumento pasado en **IDENTIFICADOR**, que en este caso es **$2** porque está en la segunda posición.

Finalmente, la tercera opción permitirá que haya redirección: **mandato_argumentos redireccion**. La redirección se tratará por separado en otra regla de la gramática.

```
mandato_argumentos:
```

```
        IDENTIFICADOR {
                struct Palabra *cmd = (struct Palabra *)malloc(sizeof(struct Palabra));
                inicializarPalabra(cmd);
                expandirPalabra(cmd,$1);
                if(cmd->valor) insertarComando(ordenActual(orden),cmd->valor); // Se toma la
1ª expansión por orden alfabético
                else insertarComando(ordenActual(orden),$1);
                #ifdef DEBUG_SINTACTICO
                        printf("Comando: %s\n",$1);
                #endif
        }
        | mandato_argumentos IDENTIFICADOR {
                struct Palabra *arg = (struct Palabra *)malloc(sizeof(struct Palabra));
                inicializarPalabra(arg);
                expandirPalabra(arg,$2);
                if(arg->valor){
                        while(arg){
                                insertarArgumento(ordenActual(orden),arg->valor);
                                arg = arg->siguiente;
                        }
                }else insertarArgumento(ordenActual(orden),$2);
                #ifdef DEBUG_SINTACTICO
                        printf("Argumento: %s\n",$2);
                #endif
        }
        | mandato_argumentos redireccion
;
```

La regla **resto** es la que permite la concatenación de varios comandos, lo cual se puede hacer múltiples veces y de dos formas: usando pipe (|) o punto y coma (**;**). Además, puede que no haya concatenación, por lo que se admite que resto sea nulo, es decir, existe una primera opción nula (por ello aparece | antes de la "primera" opción escrita, que de hecho es la segunda opción).

El formato es similar para el pipe y el punto y coma, pues les sigue el siguiente **comando**. En el caso del **PIPE** se indica que la **ordenActual** tiene pipe (poniendo el campo **pipe** a 1, que se entiende como true) y se inserta una nueva orden (**insertarOrden**), es decir, se abre hueco para insertar el siguiente **comando**. Para el punto y coma ocurre igual, con la única diferencia que es el

campo **puntocoma** el que se pone a 1 (true).

Además, también se admite que sea simplemente **PUNTOCOMA** para permitir que las instrucciones puedan terminar con punto y coma. En el caso de poner simplemente un punto y coma como expresión, se considerará un error sintáctico.

```
resto:
		| PIPE {
				ordenActual(orden)->pipe = 1;
				insertarOrden(orden);
				#ifdef DEBUG_SINTACTICO
					printf("Pipe (|)\n");
				#endif
		} comando
		| PUNTOCOMA {
				ordenActual(orden)->puntocoma = 1;
				insertarOrden(orden);
				#ifdef DEBUG_SINTACTICO
					printf("Punto y coma (;)\n");
				#endif
		} comando
		| PUNTOCOMA
;
```

Finalmente, se tiene la regla para la redireccion. Se trata por separado cada tipo de redirección: **REDIRECCIONANEXAR**, **REDIRECCIONSALIDA** y **REDIRECCIONENTRADA**. A todas ellas les sigue un **IDENTIFICADOR** que es el fichero al que se redirecciona. La posibilidad de múltiple redireccionamiento ya se conseguía gracias a la regla **comando** de la gramática, pues la redirección es como un argumento, es decir, puede ponerse en cualquier posición a la hora de indicar el comando.

En el caso de la redireción de anexar y de salida el procedimiento es similar y consiste en almacenar en la ordenActual el fichero al que se redirecciona (**$2**, pues lo proporciona **IDENTIFICADOR**) y el descriptor de fichero desde el que se redirecciona (**$1**, pues lo roporciona **REDIRECCIONANEXAR**), ya que se permite y el léxico devuelve que descriptor se usará. La única diferencia es que en el caso de anexar se hace con **insertarFanexar**, mientras que en el caso de salida se usa **insertarFsalida**.

Para la entrada es más simple, pues se usa **insertarFentrada**, que no requiere la indicación del descriptor de fichero desde el que se redirecciona, sino sólo el fichero al que se redirecciona; todo ello con el sentido que tiene al escribir, otra cosa es el flujo de los datos en el redireccionamiento. Por este motivo se indica simplemente el fichero de entrada de redireccionamiento, que será **$2**, pues se tiene en **IDENTIFICADOR**.

```
redireccion:
        REDIRECCIONANEXAR IDENTIFICADOR {
                insertarFanexar(ordenActual(orden),$2,$1);
                #ifdef DEBUG_SINTACTICO
                        printf("Anexar (");
                        if($1 != STDOUT_FILENO) printf("%i",$1);
                        printf(">> %s)\n",$2);
                #endif
        }
        | REDIRECCIONSALIDA IDENTIFICADOR {
                insertarFsalida(ordenActual(orden),$2,$1);
                #ifdef DEBUG_SINTACTICO
                        printf("Salida (");
                        if($1 != STDOUT_FILENO) printf("%i",$1);
                        printf("> %s)\n",$2);
                #endif
        }
        | REDIRECCIONENTRADA IDENTIFICADOR {
                insertarFentrada(ordenActual(orden),$2);
                #ifdef DEBUG_SINTACTICO
                        printf("Entrada (< %s)\n",$2);
                #endif
        }
;
```

A continuación separamos con %% la declaración de reglas de la declaración de funciones de C.

```
%%
```

Por último declaramos todas las funciones en C que queremos que se añadan al fichero de código C que creará Bison (sintactico.tab.c).

Concretamente nosotros definimos una función a ejecutar en caso de error, una función para matar a un hijo y la función main general por donde se empezará a ejecutar nuestro programa.

```
yyerror(char *s);
void matarHijo();
int main(int argc, char *argv[]);
```

El hecho de usar una función de error es absolutamente necesario, por obligación del analizador sintáctico de Bison, y debe llamarse obligatoriamente **yyerror**. Se lanzará en el caso de que haya un error sintáctico. Esto se produce cuando la expresión escrita no tiene la forma correcta de acuerdo a la gramática (p. ej., una expresión que tenga el símbolo de redirección de salida > y luego no se indica un fichero).

El control de errores consiste en indicar el error de sintaxis y limpiar la entrada que se tiene en **yytext** para que no se produzcan errores espúreos o dobles. Ademas se pone la variable **valorError** a 1 (true) para que no se ejecute nada, se limpia la estructura de almacenamiento de la orden y se solicite una nueva expresión. Además, para las expresiones que sólo son salto de línea no se hace nada para evitar errores en el tratamiento de **yytext**. En el resto de casos se limpia **yytext** llamando a **yylex()** hasta que se llegue al **SALTOLINEA**.

```
yyerror(char *s) {
        valorError = 1;
        if(strcmp(s,"syntax error") == 0){
                fprintf(stderr, "Error de sintaxis");
                if(yylval.valor && (strcmp(yylval.valor,"\n") != 0)) printf(" en %s\n",yytext);
                else printf("\n");
        }
        else fprintf(stderr,"%s\n",s);
        // Para vaciar el buffer de entrada en caso de error
        if(!yylval.valor || (strcmp(yylval.valor,"\n") != 0)) while(yylex() != SALTOLINEA);
}
```

La función **matarHijo** se limita a mandar la señal **SIGINT**, que se produce al hacer Ctrl+C, al hijo, siempre que éste exista (su **pid** se almacena para ello; cuando no hay ningún hijo vale 0).

```
void matarHijo(){
        if(pid) kill(pid,SIGINT);
}
```

Finalmente se tiene el main. El motivo de poner la función main aquí es arbitrario, ya que se

podría haber situado en otro fichero aparte. La estructura de la función main se corresponde con el diagrama general de bloques expuesto en la sección anterior. A continuación comentamos lo más importante de su código.

En primer lugar, en las inicializaciones, se crea una lista para contener todos los alias de comandos que se creen.

```
alias = (struct Alias *)malloc(sizeof(struct Alias)); inicializarAlias(alias);
```

Además, se controla el directorio actual (también se toma sólo la carpeta y no toda la ruta) y anterior, así como el **hostname**. Más adelante se tomarán los valores apropiados de cada uno para mostrarlos en el prompt.

```
directorio = (char *)malloc(sizeof(char)*MAX_DIR);
directorioAnterior = (char *)malloc(sizeof(char)*MAX_DIR);
getcwd (directorio,MAX_DIR); getcwd (directorioAnterior,MAX_DIR);
char *hostname = (char *)malloc(sizeof(char)*MAX_HOSTNAME);
char *carpeta = (char *)malloc(sizeof(char)*MAX_DIR);
```

Por otro lado se activa el control de la señal **SIGINT**, que lo hará la función **matarHijo**.

```
signal(SIGINT,matarHijo);
```

Ya se está listo para entrar en el bucle del intérprete de comandos (shell). Se inicializa a 0 la variable r que indica el retorno y permite salir del shell al introducirse el comando **exit**.

```
int r = 0;
while(1) { ... }
```

El contenido del bucle, lo primero que hace es inicializar la estructura **orden**. En ella se tendrá la orden a ejecutar, después del análisis sintáctico.

```
orden = (struct Orden *)malloc(sizeof(struct Orden));
inicializarOrden(orden);
```

Seguidamente se construye el prompt, que usa el usuario actual (**getlogin()**), el **hostname** y la **carpeta**, previamente calculada.

```
...
printf("[michele::%s@%s %s]$ ",getlogin(),hostname,carpeta);
```

Se realiza el análisis sintáctico, que devuelve r un valor que indica casos especiales, como saltos de línea sin comando, etc. Además, ya se tiene la **orden** lista para ejecutar. Antes de llamar a **yyparse** se resetea a **NULL** el valor de **yylval.valor**, de forma que los errores pueden controlarse

siempre bien, evitando valores espúreos o viejos.

```
yylval.valor = NULL;
r = yyparse();
```

El salto de línea sin comando devuelve -2, de modo que se limpia la orden y se vuelve a iterar el bucle:

```
if(r == -2){
		liberarOrden(orden);
		continue;
}
```

Si no ha habido error (sintáctico) se ejecuta la orden; además, se pasan los alias, por si lo que hay que ejecutar es un alias. En caso de haber error no se ejecuta nada pero se resetea la variable **valorError** poniéndola a 0, indicando que no hay error.

```
if(!valorError) r = ejecutar(orden,alias);
else valorError = 0;
```

Finalmente se libera la **orden**, para que se vuelve a llenar correctamente en la siguiente iteración, que leerá otra expresión introducida por el usuario del shell.

```
liberarOrden(orden);
```

Si la ejecución devolvió -1 quiere decir que el comando **exit** aparecía en la ejecución. Es responsabilidad de la ejecución y no del análisis sintáctico, porque es la única forma de permitir que se ponga **exit** concatenado o junto con otros comandos dentro de una misma expresión. De lo contrario, no se ejecutaría ninguna si aparece el exit. Es la ejecución a que para el ver el **exit** y devuelve -1 para que se salga del bucle.

```
if(r == -1) break;
```

A la salida del bucle se libera la memoria tomada para las variables usadas para el prompt, y se termina el programa.

structs.h

Aquí definimos básicamente las estructuras que se usarán en nuestro programa, así como la declaración prototipo de las funciones que usamos para visualizar dichas estructuras.

[DESDE AKI]

A continuación definimos la esctructura "Palabra", que representa una secuencia de palabras,

concretamente una palabra (**valor**) y un puntero a la **siguiente** palabra. La secuencia terminará cuando dicho puntero sea NULL.

```
struct Palabra {
	char *valor;
	struct Palabra *siguiente;
};
```

Ahora definimos la estructura "Orden", que representa la orden que vamos a ejecutar. Básicamente se usa para que el analizador sintáctico vaya construyéndola a medida que va parseando la sentencia. En caso de que su sintaxis sea correcta, se le pasa a la función ejecutar que la recorrerá e irá ejecutando los comandos correspondientes. Esta estructura está comentada con la semántica de cada campo.

```
struct Orden {
	char *comando;			// Comando
	struct Palabra *argumentos;	// Argumentos
	struct Palabra *fanexar;	// Ficheros de salida en modo anexar (>>)
	struct Palabra *fsalida;		// Ficheros de salida (>)
	struct Palabra *fentrada;	// Ficheros de entrada (<); no es necesario almacenarlos
	char *fanexarActual;		// Fichero de salida en modo anexar que se usará; es excluyente con fsalidaActual
	char *fsalidaActual;		// Fichero de salida que se usará; es excluyente con fanexarActual
	char *fentradaActual;		// Fichero de entrada que se usará
	int fdActual;			// Descriptor de fichero (fd) de la entrada para los redireccionamientos fd> y fd>>
	int puntocoma;			// Hay punto y coma (no hay pipe)?
	int pipe;			// Hay pipe (no hay punto y coma)?
	int segundoplano;		// Hay segundo plano?
	struct Orden *siguiente;	// Siguiente orden, si hay punto y coma o pipe
};
```

La estructura "Orden" permite almacenar el comando y sus argumentos y todos los ficheros a que se redireccione (como "Palabra"), si bien también se mantiene el último o actual, que es el verdaderamente usado en la redirección. Ocurre lo mismo con el descriptor de fichero actual, referido al descriptor de fichero desde el que se redirecciona para anexar o salida. Finalmente se puede tener otra orden **siguiente** en cuyo caso se indica si la actual tiene **pipe** o **puntocoma**. También se permite que cada orden puede ejecutarse en **segundoplano**.

Luego definimos la estructura "Alias" que almacenará los alias que hayamos definido. Básicamente se compone del alias a sustituir (**valor**), la secuencia sustituta (**comando**, que se almacena como ristra y en el momento de reemplazar al alias, se analizará sintácticamente de forma local) y un puntero al **siguiente** alias. En caso de ser NULL este último puntero, no habrán más alias. Hay que destacar la necesidad de que la orden a la que equivale el alias no puede guardarse como una estructura "Orden" porque las órdenes pueden depender del momento en que se lancen y no del momento en que se crea el alias. Esto queda claramente patente en el caso de usar caracteres comodín, que requerirán un proceso de expansión dependiente del momento en que se haga.

```
struct Alias {
    char *valor;                // Alias
    char *comando;              // Orden (como ristra)
    struct Alias *siguiente;            // Siguiente alias
};
```

Por último definimos las declaraciones protoipo de las funciones que usamos para visualizar las estructuras expuestas, que se comentarán en el siguiente apartado junto con las funciones no exportadas en la interfaz del módulo **structs**.

En el caso de la estructura Palabra, tenemos los siguientes procedimientos:

```
void imprimirPalabras(struct Palabra *p,char *titulo);
struct Palabra* palabraActual(struct Palabra *p);
void insertarValor(struct Palabra *p,char *v);
void inicializarPalabra(struct Palabra *p);
void insertarPalabra(struct Palabra *p);
void copiarPalabra(struct Palabra *p1,struct Palabra *p2);
void liberarPalabra(struct Palabra *p);
struct Palabra* expandirPalabra(struct Palabra *p,char *r);
```

Por otro lado, para la estructura Orden tenemos:

```
void imprimirOrden(struct Orden *o);
struct Orden* ordenActual(struct Orden *o);
void inicializarOrden(struct Orden *o);
void insertarComando(struct Orden *o,char *cmd);
void insertarArgumento(struct Orden *o,char *arg);
void insertarFanexar(struct Orden *o,char *f, int fd);
void insertarFsalida(struct Orden *o,char *f, int fd);
```

```
void insertarFentrada(struct Orden *o,char *f);
void insertarOrden(struct Orden *o);
void copiarOrden(struct Orden *o1,struct Orden *o2);
void fusionarOrden(struct Orden *o1,struct Orden *o2);
void liberarOrden(struct Orden *o);
char **vectorArgv(struct Orden *o,struct Palabra *argAdicional);
void crearFicherosOrden(struct Orden *o);
```

Por último, para la estructura Alias tenemos:

```
void imprimirAlias(struct Alias *a);
struct Alias* aliasActual(struct Alias *a);
void inicializarAlias(struct Alias *a);
void insertarAliasValor(struct Alias *a,char *alias);
void insertarAliasComando(struct Alias *a,char *cmd);
void insertarAlias(struct Alias *a,char *alias,char *cmd);
void eliminarAlias(struct Alias *a,char *alias);
void liberarAlias(struct Alias *a);
struct Alias* buscarAlias(struct Alias *a,char *alias);
```

structs.c

En este archivo definimos todas los procedimientos a usar con las estructuras referidas en el punto anterior. Incluímos tanto procedimientos de inicialización, liberación, manejo y depuración.

En el caso de la estructura Palabra, tenemos los siguientes procedimientos, de los cuales se comentan aquellos verdaderamente interesantes, pues muchos de ellos son meras agrupaciones de código para su mejor manejo, o bien funciones para imprimir la estructura, a modo de utilidad de depuración.

Para imprimir la estructura **Palabra** se dispone de la función **imprimirPalabras**, con la posibilidad indicar un **titulo** que las preceda; es útil para depurar.

```
void imprimirPalabras(struct Palabra *p,char *titulo) {...}
```

Para avanzar hasta la última palabra almacenada en la estructura se usa **palabraActual**, que devuelve un puntero a la última palabra. Es útil para inserciones al final o acceso a los campos de la última palabra insertada en la estructura **Palabra**.

```
struct Palabra* palabraActual(struct Palabra *p) {...}
```

Para insertar una palabra, o mejor dicho su valor, se indica con v a la función **insertarValor**, que toma memoria y lo inserta en el campo **valor**.

```
void insertarValor(struct Palabra *p,char *v) {...}
```

Al crear una palabra, a parte de tomar memoria para ella, se podrá inicializar con **inicializarPalabra**, que pone a NULL todos los campos.

```
void inicializarPalabra(struct Palabra *p) {...}
```

Para insertar una palabra se recorre la estructura hasta llegar a la última palabra contenida y se abre hueco para una nueva palabra y se inicializa con **inicializarPalabra**. Todo ello lo hace **insertarPalabra**.

```
void insertarPalabra(struct Palabra *p){
	// La Palabra no debe ser nula
	// (de lo contrario, las modificaciones no se verían externamente)
	if(p->siguiente) insertarPalabra(p->siguiente);
	else{
		p->siguiente = (struct Palabra *)malloc(sizeof(struct Palabra));
		inicializarPalabra(p->siguiente);
	}
}
```

Para poder copiar estructuras **Palabra** se usa la función **copiarPalabra**, de modo que recorre las estructuras Palabra **p1** y **p2**, donde la primera es una palabra vacía pero inicializada, que se irá rellenando con **p2**.

```
void copiarPalabra(struct Palabra *p1,struct Palabra *p2){
	if(!p2) p1 = NULL; // p2 no debería ser nulo
	else{
		if(p2->valor){
			p1->valor = (char *)malloc(sizeof(char)*(strlen(p2->valor)+1));
			strcpy(p1->valor,p2->valor);
		}else p1->valor = NULL;
		if(p2->siguiente){
			p1->siguiente = (struct Palabra *)malloc(sizeof(struct Palabra));
			copiarPalabra(p1->siguiente,p2->siguiente);
		}else p1->siguiente = NULL;
	}
}
```

Cuando una palabra ya no se vaya a usar más, se podrá liberar la memoria que ocupa llamando a **liberarPalabra**.

```
void liberarPalabra(struct Palabra *p) {...}
```

Para poder tratar la expansión de caracteres comodín se hace uso de expresiones regulares del estándar POSIX. A la hora de expandir una palabra, se trata de realizar los siguientes pasos:

1. Obtener los ficheros del directorio actual, pues ellos son los valores posibles a los que puede expandirse una palabra. Al obener los ficheros del directorio con la función **scandir** se indica que se use la función **alphasort** para que se tomen en orden alfabético.
2. Si exiten caracteres comodín (? o *) se procede a comparar cada fichero con la expresión regular de la ristra r que representa la palabra que tiene caracteres comodín.

```
struct Palabra* expandirPalabra(struct Palabra *p,char *r){
	// Obtener los ficheros del directorio actual
	struct dirent **ficheros;
	int n = scandir(".",&ficheros, 0, alphasort); // alphasort --> Orden alfabético

	// Rellenar estructura Palabra con las coincidencias
	if(index(r,'?') || index(r,'*')){
		int i;
		for(i=0; i<n; i++){
			if(compararExpresionRegular(ficheros[i]->d_name,expresionRegular(r))){
				if(p->valor) insertarPalabra(p);
				insertarValor(palabraActual(p),ficheros[i]->d_name);
			}
		}
	}
	return p;
}
```

Se hace uso de dos funciones auxiliares para la conversión a expresión regular y la comparación con expresiones regulares. La conversión a expresión regular consiste básicamente en sustituir los caracteres comodín por sus equivalentes en la notación de expresiones regulares POSIX. Dicha conversión se muestra en la siguiente **Tabla1**.

Caracter Comodín	*Equivalente POSIX*	*Significado*
?	.	Un caracter cualquiera
*	.*	Ningún o cualquier número de caracter cualqueira

Tabla 1: Conversión de Caracteres Comodín

La función **expresionRegular** simplemente realiza la búsqueda y sustitución de los caracteres comodín por los equivalente POSIX para poder aplicar luego la comparación con expresiones regulares POSIX.

```
char* expresionRegular(char *r){
	int i = 0, k = 0;
	char *er = (char *)malloc(sizeof(char));
	er[k++] = '^';
	for(; i < strlen(r); i++){
		er = (char *)realloc(er,sizeof(char)*(k+1));
		if	(r[i] == '?') er[k++] = '.';
		else if(r[i] == '*'){er[k++] = '.';  er = (char
*)realloc(er,sizeof(char)*(k+1)); er[k++] = '*';}
		else if(r[i] == '.'){er[k++] = '\\'; er = (char
*)realloc(er,sizeof(char)*(k+1)); er[k++] = '.';}
		else			er[k++] = r[i];
	}
	er = (char *)realloc(er,sizeof(char)*(k+2));
	er[k++] = '$'; er[k] = '\0';
	return er;
}
```

La comparación de expresiones regulares se hace con **compararExpresionRegular**, usando variables de tipo **regex_t** (tipo regular expresion, es decir, expresión regular). Lo primero es compilarla con **regcomp** y luego ejecutarla con **regexec**. La expresión regular se obtiene de la función antes comentadas y ahora se conoce como **patron**. Tras compilar, la ejecución devuelve 0 si hay coincidencia, de forma que se niega para que se devuelva 1 (true) en caso de coincidencia.

```
int compararExpresionRegular(char *r, char *patron){
	regex_t er;
	if (regcomp(&er,patron,0) != 0) return -1;	// Error compilando la expresión regular
	return !regexec(&er,r,(size_t)0,NULL,0);	// Devuelve 1 si hay coincidencia con la expresión
regular
```

```
}
```

Por otro lado, para la estructura Orden tenemos inicialmente otra función de impresión de la estructura (**imprimirOrden**), otra para recorrer la estructura para llegar a la última orden o actual (**ordenActual**), otra para inicializar la estructura (**inicializarOrden**), que son similares a las de **Palabra**, variando lo justo para adecuarse a la estructura **Orden**.

```
void imprimirOrden(struct Orden *o) {...}
struct Orden* ordenActual(struct Orden *o) {...}
void inicializarOrden(struct Orden *o) {...}
```

Seguidamente se define una lista de funciones para insertar valores a los distintos campos de la estructura. Se trata de insertar comando con **insertarComando**, insertar argumento con **insertarArgumento**, insertar fichero de redirección de anexar (**insertarFanexar**), de salida (**insertarFsalida**) y de entrada (**insertarFentrada**). En el caso de las funciones de inserción de ficheros de anexar, se actualiza el fichero de anexar, salida y entrada actual, pues sólo en uno se hace efectiva la redirección. Además, en el caso de anexar y salida se indica el descriptor del fichero desde el que se redirecciona.

```
void insertarComando(struct Orden *o,char *cmd) {...}
void insertarArgumento(struct Orden *o,char *arg) {...}
void insertarFanexar(struct Orden *o,char *f, int fd) {...}
void insertarFsalida(struct Orden *o,char *f, int fd) {...}
void insertarFentrada(struct Orden *o,char *f) {...}
```

Como en el caso de **Palabra**, para la **Orden** también se dispone de una función para abrir hueco para una orden e inicializarla, mediante **insertarOrden**. También se tiene una función para la copia de una orden en orden, con **copiarOrden**.

```
void insertarOrden(struct Orden *o) {...}
void copiarOrden(struct Orden *o1,struct Orden *o2) {...}
```

Se dispone de una función llamada **fusionarOrden** necesaria para el manejo de los alias. Dicha función tiene el propósito de que los alias puedan tener a su vez argumentos, de forma que tras la expansión de los mismos, se tiene la orden del propio comando que es el alias y del alias como comando que puede tener argumentos adicionales, entre otras cosas. La fusión trata de añadir el resto de órdenes en el campo **siguiente** y se requiere un control especial en el resto de campos de la última orden de la estructura **Orden** que se forma con el comando que representa el alias, que es al que se le añade el resto. Para estos campos, el tratamiento consiste en añadir argumentos y

actualizar las redirecciones, así como marcar el tipo de concatenación si la hubiere, que puede ser de pipe o punto y coma, además de indicar si se ejecutará en segundo plano.

```
void fusionarOrden(struct Orden *o1,struct Orden *o2){
    if(o2){
        struct Orden *oActual = (struct Orden *)malloc(sizeof(struct Orden));
        oActual = ordenActual(o1);
        struct Palabra *aux = (struct Palabra *)malloc(sizeof(struct Palabra));
        if(o2->argumentos) aux = o2->argumentos;
        else aux = NULL;
        while(aux){
            insertarArgumento(oActual,aux->valor);
            aux = aux->siguiente;
        }
        if(o2->fanexar) aux = o2->fanexar;
        else aux = NULL;
        while(aux){
            insertarFanexar(oActual,aux->valor,o2->fdActual);
            aux = aux->siguiente;
        }
        if(o2->fsalida) aux = o2->fsalida;
        else aux = NULL;
        while(aux){
            insertarFsalida(oActual,aux->valor,o2->fdActual);
            aux = aux->siguiente;
        }
        if(o2->fentrada) aux = o2->fentrada;
        else aux = NULL;
        while(aux){
            insertarFentrada(oActual,aux->valor);
            aux = aux->siguiente;
        }
        if(o2->fanexarActual){
            oActual->fanexarActual = (char *)malloc(sizeof(char)*(strlen(o2-
>fanexarActual)+1));
            strcpy(oActual->fanexarActual,o2->fanexarActual);
        }
        if(o2->fsalidaActual){
```

```
                oActual->fsalidaActual = (char *)malloc(sizeof(char)*(strlen(o2->fsalidaActual)+1));
                strcpy(oActual->fsalidaActual,o2->fsalidaActual);
            }
            if(o2->fentradaActual){
                oActual->fentradaActual = (char *)malloc(sizeof(char)*(strlen(o2->fentradaActual)+1));
                strcpy(oActual->fentradaActual,o2->fentradaActual);
            }
            oActual->fdActual = o2->fdActual;
            oActual->puntocoma = o2->puntocoma;
            oActual->pipe = o2->pipe;
            oActual->segundoplano = o2->segundoplano;
            if(o2->siguiente){
                oActual->siguiente = (struct Orden *)malloc(sizeof(struct Orden));
                inicializarOrden(oActual->siguiente);
                copiarOrden(oActual->siguiente,o2->siguiente);
            }
        }
}
```

Cuando ya no se vaya a usar más la orden, se podrá liberar su espacio con **liberarOrden**.

```
void liberarOrden(struct Orden *o) {...}
```

Adicionalmente, se requiere una función para construir el vector de argumentos que se pasa a la función de ejecutar, de la familia de funciones **exec**. Dicho vector de argumentos debe ser de tipo vector de ristras en nuestro caso, pues se usa la función **execvp** que requiere un vector para los parámetros. Esta función permite que dicho vector tenga el nombre del programa o comando y luego todos los parámetros, que es el formato que se requiere.

También se permite añadir más parámetros de forma interna, indicados en **argAdicionales**, lo cual permite que determinados comandos se ejecuten de forma especial, controlados por el propio shell. Es el caso del comando ls, que se ejecuta con colo, añadiendo el argumento **--color**.

```
char **vectorArgv(struct Orden *o, struct Palabra *argAdicionales){
    struct Palabra *aux = o->argumentos;
    int i;
    char **arg;
```

```
    for(i = 0; aux != NULL; aux = aux->siguiente, i++);
    aux = argAdicionales;
    for(; aux != NULL; aux = aux->siguiente, i++);
    arg = (char **)malloc(sizeof(char *)*(i+2));    // Argumentos + Comando + NULL
    arg[0] = o->comando;
    aux = argAdicionales;
    for(i = 1; aux != NULL; aux = aux->siguiente, i++) arg[i] = aux->valor;
    aux = o->argumentos;
    for (; aux != NULL; aux = aux->siguiente, i++) arg[i] = aux->valor;
    arg[i] = NULL;

    return arg;
}
```

Finalmente, para tener un comportamiento idéntico al del Bash, cuando la redirección de anexar o salida se hace desde múltiples ficheros, solo el último será el efectivo (además, sólo uno incluyendo ambos tipos, no uno para cada tipo de redirección, pues la redirección es de un único fichero, sea del tipo que sea; sin tener en cuenta la de entrada, que no interfiere en las de anexar y salida), pero el resto de ficheros se crean vacíos. La función **crearFicherosOrden** se encarga de hacer esto llamando a la función auxiliar **crearFicheros**.

```
void crearFicherosOrden(struct Orden *o){
    crearFicheros(o->fanexar);
    crearFicheros(o->fsalida);
}
```

La función **crearFicheros** recorre las estructuras **Palabra** que contienen los ficheros y los va abriendo, lo que permite que funciona para los ficheros de anexar, o creando si no existen, funcionando también para los de salida.

```
void crearFicheros(struct Palabra *p){
    if(p){
        int fd;
        if((fd = open(p->valor, O_CREAT, S_IRUSR | S_IWUSR)) == -1)
            printf("No se puede abrir %s para anexar o salida\n",p->valor);
        close(fd);
        crearFicheros(p->siguiente);
    }
```

```
}
```

Por último, para la estructura Alias tenemos también las funciones para imprimir la estructura **Alias** con **imprimirAlias**, llegar al alias actual o último con **aliasActual**, iniciliazar la estructura con **inicializarAlias**, así como las funciones **insertarAliasValor** e **insertarAliasComando** para insertar el nombre del alias y el comando al que representa en los campos de la estructura. También se tienen la función **insertarAlias** para abrir hueco e inicializar un alias.

```
void imprimirAlias(struct Alias *a) {...}
struct Alias* aliasActual(struct Alias *a) {...}
void inicializarAlias(struct Alias *a) {...}
void insertarAliasValor(struct Alias *a,char *alias) {...}
void insertarAliasComando(struct Alias *a,char *cmd) {...}
```

Para mayor versatilidad, se permite insertar y eliminar alias de la estructura, para lo que se dispone de las funciones **insertarAlias** y **eliminarAlias**. La inserción se hace de forma que no se produzcan duplicados, es decir, no habrá dos alias con el mismo nombre.

```
void insertarAlias(struct Alias *a,char *alias,char *cmd){
	// El Alias no debe ser nulo
	// (de lo contrario, las modificaciones no se verían externamente)
	if(!a->valor){insertarAliasValor(a,alias); insertarAliasComando(a,cmd);}
	else if(strcmp(a->valor,alias) == 0) insertarAliasComando(a,cmd);
	else if(a->siguiente) insertarAlias(a->siguiente,alias,cmd);
	else{
		a->siguiente = (struct Alias *)malloc(sizeof(struct Alias));
		inicializarAlias(a->siguiente);
		insertarAliasValor(a->siguiente,alias); insertarAliasComando(a->siguiente,cmd);
	}
}

void eliminarAlias(struct Alias *a,char *alias){
	if(strcmp(a->valor,alias) == 0){
		if(a->siguiente){
			insertarAliasValor(a,a->siguiente->valor); free(a->siguiente->valor);
			insertarAliasComando(a,a->siguiente->comando); free(a->siguiente-
>comando);
			a->siguiente = a->siguiente->siguiente;
```

```
            }else{
                free(a->valor); free(a->comando); inicializarAlias(a);
            }
        }else if(a->siguiente){
            if(strcmp(a->siguiente->valor,alias) == 0){
                free(a->siguiente->valor);
                free(a->siguiente->comando);
                a->siguiente = a->siguiente->siguiente;
            }else eliminarAlias(a->siguiente,alias);
        }
}
```

Cuando ya no se vaya a usar más la estructura de alias se liberará su espacio con **liberarAlias**.

```
void liberarAlias(struct Alias *a) {...}
```

Finalmente, también se implementa la función **buscarAlias** que permite la búsqueda de una **alias** dentro de la estructura, para poder llegar al alias concreto si existe y tomar el comando que representa. La búsqueda es una simple búsqueda recursiva secuencial por toda la estructura **Alias**.

```
struct Alias* buscarAlias(struct Alias *a,char *alias){
    if(!a) return NULL;
    if(a->valor && (strcmp(a->valor,alias) == 0)) return a;
    return buscarAlias(a->siguiente,alias);
}
```

Código fuente con comentarios

ejecucion.h

```
#ifndef EJECUCION_H
#define EJECUCION_H

#include "structs.h"

int ejecutar(struct Orden *o,struct Alias *alias);

#endif
```

ejecucion.c

```
#include <unistd.h>
#include <signal.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include "ejecucion.h"
#include "structs.h"

#define MAX_STRING 256
#define MAX_DIR MAX_STRING
#define FICHEROAUXILIAR "_orden_auxiliar_"

extern FILE *yyin;
extern char *directorioAnterior;
extern int pid;

int ejecutarComando(struct Orden *o) {
    // EjecuciÃ³n del comando (se busca en las rutas del sistema)
```

```
    int i = 0;
    struct Orden *aux = (struct Orden *)malloc(sizeof(struct Orden)); inicializarOrden(aux);
    if(strcmp(o->comando,"ls") == 0) insertarArgumento(aux,"--color");
    execvp(o->comando,vectorArgv(o,aux->argumentos));
    printf("Comando %s no encontrado\n",o->comando);
    exit(1);
}

int ejecutar(struct Orden *o,struct Alias *a){
    int estado;
    struct Orden *ordenActual;
    int salida, entrada, anexar;
    int salidaAnterior, entradaAnterior, anexarAnterior;
    int tuberia[2];
    int tuberiaAnterior = -1;

    // Muestreo de todos los alias o CreaciÃ³n de un alias
    if(strcmp(o->comando,"alias") == 0){
        if(o->argumentos){
            if((!o->argumentos->siguiente) ||
               (!o->argumentos->siguiente->siguiente) ||
               (strcmp(o->argumentos->siguiente->valor,"=") != 0)) printf("DeclaraciÃ³n de alias
incorrecta\n");
            else insertarAlias(a,o->argumentos->valor,strcat(o->argumentos->siguiente-
>siguiente->valor,"\n"));
        }else imprimirAlias(a);
        return 0;
    }

    // EliminaciÃ³n de un alias
    if(strcmp(o->comando,"unalias") == 0){
        if(o->argumentos) eliminarAlias(a,o->argumentos->valor);
```

```
    return 0;
}

// ExpansiÃ³n de un alias (para construir la orden y ejecutarla)
struct Alias *aliasAux = (struct Alias *)malloc(sizeof(struct Alias));
if(aliasAux = buscarAlias(a,o->comando)){
    struct Orden *ordenAux = (struct Orden *)malloc(sizeof(struct Orden));
    copiarOrden(ordenAux,o);
    liberarOrden(o); o = (struct Orden *)malloc(sizeof(struct Orden)); inicializarOrden(o);

    FILE *yyinAnterior = yyin;
    yyin = fopen(FICHEROAUXILIAR,"w+"); // Se escribe la orden en FICHEROAUXILIAR
    fprintf(yyin,aliasAux->comando);
    fseek(yyin,0,SEEK_SET);          // Se parsea la orden del FICHEROAUXILIAR
    yyparse();
    fclose(yyin);
    remove(FICHEROAUXILIAR);
    yyin = yyinAnterior;             // Se restaura la entrada de parseo

    fusionarOrden(o,ordenAux); liberarOrden(ordenAux);
}

ordenActual = o;
while (ordenActual != NULL) {
    // Crear ficheros de salida y anexar (Ãºtil cuando son mÃ¡s de uno)
    crearFicherosOrden(ordenActual);

    // TuberÃa de entrada
    if (tuberiaAnterior != -1 && !ordenActual->fentradaActual) {
        entradaAnterior = dup(STDIN_FILENO);
        dup2(tuberiaAnterior,STDIN_FILENO);
        close(tuberiaAnterior);
```

```
        }

        // TuberÃa de salida
        if (ordenActual->pipe && !(ordenActual->fsalidaActual || ordenActual->fanexarActual)) {
            pipe(tuberia);
            salidaAnterior = dup(STDOUT_FILENO);
            dup2(tuberia[1],STDOUT_FILENO);
            close(tuberia[1]);
            tuberiaAnterior = tuberia[0];
        }else tuberiaAnterior = -1;

        // Redireccionamiento de salida
        if (ordenActual->fsalidaActual) {
            salidaAnterior = dup(o->fdActual);
            if ((salida = open(ordenActual->fsalidaActual, O_WRONLY | O_TRUNC | O_CREAT, S_IRUSR
| S_IWUSR)) == -1) {
                printf("No se puede abrir %s para escritura\n", ordenActual->fsalidaActual);
                return -2;
            }
            close(o->fdActual);
            dup2(salida,o->fdActual);
        }

        // Redireccionamiento de anexar
        if (ordenActual->fanexarActual) {
            anexarAnterior = dup(o->fdActual);
            if ((anexar = open(ordenActual->fanexarActual, O_WRONLY | O_APPEND | O_CREAT,
S_IRUSR | S_IWUSR)) == -1) {
                printf("No se puede abrir %s para anexar\n", ordenActual->fanexarActual);
                return -2;
            }
            close(o->fdActual);
```

```
            dup2(anexar,o->fdActual);
        }

        // Redireccionamiento de entrada
        if (ordenActual->fentradaActual) {
            entradaAnterior = dup(STDIN_FILENO);
            if ((entrada = open (ordenActual->fentradaActual, O_RDONLY)) == -1) {
                printf("No se puede abrir %s para lectura\n", ordenActual->fentradaActual);
                return -2;
            }
            close(STDIN_FILENO);
            dup2(entrada,STDIN_FILENO);
        }

        // EjecuciÃ³n de un comando con argumentos
        if (strcmp(ordenActual->comando,"cd") == 0) {
            // Comandos no existentes como programas
            // cd (Cambiar de directorio). Si hay mÃ¡s de un argumento, se ignora
            char *dir = (char *)malloc(sizeof(char)*strlen(directorioAnterior));
strcpy(dir,directorioAnterior);
            getcwd(directorioAnterior,MAX_DIR);
            // Ir a directorio personal (Si no hay directorio personal, se va a la raÃz)
            if (ordenActual->argumentos == NULL ||
                (strcmp(ordenActual->argumentos->valor,"~") == 0) ||
                (strcmp(ordenActual->argumentos->valor,"--") == 0))
chdir(getenv("HOME")!=NULL?getenv("HOME"):"/");
            // Ir a directorio anterior
            else if (strcmp(ordenActual->argumentos->valor,"-") == 0) chdir(dir);
            // Ir al directorio indicado
            else chdir(ordenActual->argumentos->valor);
        }else if (strcmp(ordenActual->comando,"exit") == 0) {
            // exit (Cerrar el shell)
```

```
        return -1;
    }else if ((strcmp(ordenActual->comando,"killall") == 0) &&
            (strcmp(ordenActual->argumentos->valor,"shell") == 0)) {
        // Se intenta matar al shell (killall shell), se ignora
    }else {
        // No se hace nada si no se indicÃ³ ningÃºn comando ("")
        // Comandos existentes como programas
        switch(pid = fork()){
            case -1:    // Fallo
                printf("Fallo al crear un nuevo proceso con fork()\n");
                return -2;
            case 0:     // Hijo
                ejecutarComando(ordenActual);
            default:    // Padre
                waitpid(pid,&estado,ordenActual->segundoplano?WNOHANG:0);
        }
    }

    // Reestablecemos los flujos de entrada y salida
    if (entradaAnterior != -1 && !ordenActual->fentradaActual) {
        dup2(entradaAnterior,STDIN_FILENO);
        entradaAnterior = -1;
    }
    if (ordenActual->pipe && !(ordenActual->fsalidaActual || ordenActual->fanexarActual)) {
        dup2(salidaAnterior,STDOUT_FILENO);
    }
    if (ordenActual->fsalidaActual) {
        close(o->fdActual);
        close(salida);
        dup2(salidaAnterior,o->fdActual);
    }
```

```
        if (ordenActual->fanexarActual) {
            close(o->fdActual);
            close(anexar);
            dup2(anexarAnterior,o->fdActual);
        }
        if (ordenActual->fentradaActual) {
            close(STDIN_FILENO);
            close(entrada);
            dup2(entradaAnterior,STDIN_FILENO);
        }
        // Si hay redireccionamiento de salida o de anexar, se abre pipe (aunque no se escribirÃ¡ nada en Ã©l)
        if(ordenActual->pipe && (ordenActual->fsalidaActual || ordenActual->fanexarActual)) {
            pipe(tuberia);
            close(tuberia[1]);
            tuberiaAnterior = tuberia[0];
        }
        ordenActual = ordenActual->siguiente;
    }
    return 0;
}
```

lexico.l

```
%{
//#define DEBUG_LEXICO
#include "sintactico.tab.h"
#include <unistd.h>
%}

NUMERO ([1-9][0-9]*)|0

%pointer

%x comillasdobles
%x comillassimples
%x comillassimplesansic
%x identificador

%%
```

```
\"              {
yylval.valor = (char *)malloc(sizeof(char));
strcpy(yylval.valor,"");
BEGIN(comillasdobles);
}

<comillasdobles>\\(\"|\n) {
if(yylval.valor) yylval.valor = (char *)realloc(yylval.valor,sizeof(char)*(strlen(yylval.valor)+2));
else            yylval.valor = (char *)malloc(sizeof(char)*2);
strcat(yylval.valor,++yytext);
}

<comillasdobles>\\. {
if(yylval.valor) yylval.valor = (char *)realloc(yylval.valor,sizeof(char)*(strlen(yylval.valor)+3));
else            yylval.valor = (char *)malloc(sizeof(char)*3);
strcat(yylval.valor,yytext);
}

<comillasdobles>\n {
if(yylval.valor) yylval.valor = (char *)realloc(yylval.valor,sizeof(char)*(strlen(yylval.valor)+2));
else            yylval.valor = (char *)malloc(sizeof(char)*2);
strcat(yylval.valor,yytext);
}

<comillasdobles>\"      {
strcat(yylval.valor,"\0");
#ifdef DEBUG_LEXICO
                printf("Ristra de comillas dobles: %s\n",yylval.valor);
#endif
BEGIN(INITIAL);
return IDENTIFICADOR;
}

<comillasdobles>[^\\\n\"]+ {
if(yylval.valor) yylval.valor = (char *)realloc(yylval.valor,sizeof(char)*(strlen(yylval.valor)+yyleng+1));
else            yylval.valor = (char*)malloc(sizeof(char)*(yyleng+1));
strcat(yylval.valor, yytext);
}

\'              {
yylval.valor = (char *)malloc(sizeof(char));
strcpy(yylval.valor,"");
BEGIN(comillassimples);
}

<comillassimples>\'     {
strcat(yylval.valor,"\0");
#ifdef DEBUG_LEXICO
                printf("Ristra de comillas simples: %s\n",yylval.valor);
#endif
BEGIN(INITIAL);
return IDENTIFICADOR;
}
```

```
<comillassimples>[^']+ {
if(yylval.valor) yylval.valor = (char *)realloc(yylval.valor,sizeof(char)*(strlen(yylval.valor)+yyleng+1));
else            yylval.valor = (char*)malloc(sizeof(char)*(yyleng+1));
strcat(yylval.valor,yytext);
}

\$\' {
yylval.valor = (char *)malloc(sizeof(char));
strcpy(yylval.valor,"");
BEGIN(comillassimplesansic);
}

<comillassimplesansic>\\[0-7]{1,3} {
int aux; sscanf(yytext+1,"%o",&aux);
yylval.valor = (char *)realloc(yylval.valor,sizeof(char)*(strlen(yylval.valor)+2));
int l = strlen(yylval.valor); yylval.valor[l] = aux; yylval.valor[l+1] = '\0';
}

<comillassimplesansic>\\x[0-9a-fA-F]{1,2} {
int aux; sscanf(yytext+2,"%x",&aux);
yylval.valor = (char *)realloc(yylval.valor,sizeof(char)*(strlen(yylval.valor)+2));
int l = strlen(yylval.valor); yylval.valor[l] = aux; yylval.valor[l+1] = '\0';
}

<comillassimplesansic>\\. {
if(yylval.valor) yylval.valor = (char *)realloc(yylval.valor,sizeof(char)*(strlen(yylval.valor)+2));
else            yylval.valor = (char *)malloc(sizeof(char)*2);
if     (strcmp(yytext,"\\n") == 0) strcat(yylval.valor,"\n");
else if(strcmp(yytext,"\\t") == 0) strcat(yylval.valor,"\t");
else if(strcmp(yytext,"\\r") == 0) strcat(yylval.valor,"\r");
else if(strcmp(yytext,"\\b") == 0) strcat(yylval.valor,"\b");
else if(strcmp(yytext,"\\f") == 0) strcat(yylval.valor,"\f");
else strcat(yylval.valor,yytext);
}

<comillassimplesansic>[^\\']+ {
if(yylval.valor) yylval.valor = (char *)realloc(yylval.valor,sizeof(char)*(strlen(yylval.valor)+yyleng+1));
else            yylval.valor = (char*)malloc(sizeof(char)*(yyleng+1));
strcat(yylval.valor,yytext);
}

<comillassimplesansic>\' {
strcat(yylval.valor,"\0");
#ifdef DEBUG_LEXICO
                printf("Ristra de comillas simples ANSI-C: %s\n",yylval.valor);
#endif
BEGIN(INITIAL);
return IDENTIFICADOR;
}

[ \t]+ {
#ifdef DEBUG_LEXICO
                printf("Blanco o Tabulador\n");
```

```
#endif
}

{NUMERO}?\>>            {
#ifdef DEBUG_LEXICO
                printf("Signo: %s\n",yytext);
#endif
if(yyleng > 2){
                yylval.valor = (char*)malloc(sizeof(char)*yyleng-1);                // Se ignora el signo '>>'
                yytext[yyleng-2] = 0;
                yylval.ficheroRedireccion = atoi(yytext);
}else yylval.ficheroRedireccion = STDOUT_FILENO;
return REDIRECCIONANEXAR;
}

{NUMERO}?\> {
#ifdef DEBUG_LEXICO
                printf("Signo: %s\n",yytext);
#endif
if(yyleng > 1){
                yylval.valor = (char*)malloc(sizeof(char)*yyleng);  // Se ignora el signo '>'
                yytext[yyleng-1] = 0;
                yylval.ficheroRedireccion = atoi(yytext);
}else yylval.ficheroRedireccion = STDOUT_FILENO;
return REDIRECCIONSALIDA;
}

\< {
#ifdef DEBUG_LEXICO
                printf("Signo: <\n");
#endif
return REDIRECCIONENTRADA;
}

\| {
#ifdef DEBUG_LEXICO
                printf("Signo: |\n");
#endif
return PIPE;
}

\& {
#ifdef DEBUG_LEXICO
                printf("Signo: &\n");
#endif
return SEGUNDOPLANO;
}

\; {
#ifdef DEBUG_LEXICO
                printf("Signo: ;\n");
#endif
return PUNTOCOMA;
}
```

```
\n {
#ifdef DEBUG_LEXICO
		printf("Salto de lÃnea\n");
#endif
// Permite comportamiento correcto ante errores sintÃicticos al final de la orden
yylval.valor = "\n";
return SALTOLINEA;
}

\\. {
yylval.valor = (char*)malloc(sizeof(char)*2);
strcpy(yylval.valor,++yytext);
BEGIN(identificador);
}

[^ \t\n\"\'\|\;\<\>\&\\] {
yylval.valor = (char*)malloc(sizeof(char)*2);
strcpy(yylval.valor,yytext);
BEGIN(identificador);
}

<identificador>\\. {
yylval.valor = (char *)realloc(yylval.valor,sizeof(char)*(strlen(yytext)+yyleng));
strcat(yylval.valor,++yytext);
}

<identificador>[^ \t\n\"\'\|\;\<\>\&\\]+ {
yylval.valor = (char*)realloc(yylval.valor,sizeof(char)*(strlen(yytext)+yyleng+1));
strcat(yylval.valor,yytext);
}

<identificador>[ \t\n\"\'\|\;\<\>\&\\] {
#ifdef DEBUG_LEXICO
		printf("Identificador: %s\n",yylval.valor);
#endif
yyless(0);
BEGIN(INITIAL);
return IDENTIFICADOR;
}

. {
printf("CarÃicter %s ilegal\n",yytext);
}

%%
```

Makefile

```
#############
# Variables #
#############
```

```
COMPILADOR = gcc
OPCIONES-COMPILADOR = -lfl
FLEX = flex
# -I --> Lee un caracter mÃis (lookahead) aunque no sea necesario
# -Cem --> Grado de compresiÃ³n de la tabla de sÃmbolos
OPCIONES-FLEX = -I -Cem
LEXICO = lexico.l
ANALIZADOR-LEXICO = lex.yy.c
BISON = bison
# -d --> Genera tabla de sÃmbolos terminales en el fichero sintactico.tab.h
# -v --> Genera fichero sintactico.output de explicaciÃ³n del autÃ³mata
# -t --> Activa el modo depuraciÃ³n
OPCIONES-BISON = -d -v #-t
SINTACTICO = sintactico.y
ANALIZADOR-SINTACTICO = sintactico.tab.c
TABLA-SIMBOLOS = sintactico.tab.h
STRUCTS = structs.c
EJECUCION = ejecucion.c



##########
# Reglas #
##########

shell: build-shell
   clear; ./shell

build-shell: clean build-sintactico build-lexico
   $(COMPILADOR) -o shell $(ANALIZADOR-SINTACTICO) $(ANALIZADOR-LEXICO) $(STRUCTS)
$(EJECUCION) $(OPCIONES-COMPILADOR)

build-lexico:
   $(FLEX) $(LEXICO)
```

```
build-sintactico:
	$(BISON) $(OPCIONES-BISON) $(SINTACTICO)

clean:
	rm -f shell $(ANALIZADOR-SINTACTICO) $(TABLA-SIMBOLOS) $(ANALIZADOR-LEXICO)

install:
	cp shell /bin/
```

sintactico.y

```
%{
#include <stdio.h>
#include <sys/types.h>
#include <signal.h>
#include <unistd.h>
#include "structs.h"
#include "ejecucion.h"

//#define DEBUG_SINTACTICO
//#define DEBUG_STRUCT
//#define DEBUG_ALIAS
#define MAX_STRING 256
#define MAX_DIR MAX_STRING
#define MAX_HOSTNAME MAX_STRING

extern FILE *yyin;                              // Declarado en lexico.l
extern char *yytext;                    // Declarado en lexico.l
int yydebug = 1;                                // Se habilita el modo debug si se usa -t
struct Orden *orden;                    // Orden a ejecutar (analizada lÃ©xica y sintÃicticamente)
struct Alias *alias;                    // Lista de alias
int valorError = 0;                             // Indica si ha habido error de sintaxis
char *directorio;                               // Directorios actual,
char *directorioAnterior; // y anterior
int pid = 0;                                    // pid del hijo actual, por si hay que enviarle alguna
seÃ±al
%}

%union {
        char *valor;
        int ficheroRedireccion;
}

%token REDIRECCIONANEXAR
%token REDIRECCIONSALIDA
%token REDIRECCIONENTRADA
%token PIPE
```

```
%token SEGUNDOPLANO
%token PUNTOCOMA
%token SALTOLINEA
%token IDENTIFICADOR

%type <valor> IDENTIFICADOR
%type <ficheroRedireccion> REDIRECCIONANEXAR
%type <ficheroRedireccion> REDIRECCIONSALIDA

%%

expresion: SALTOLINEA {return -2;} | comando SALTOLINEA {return 0;}
;

comando:
		mandato_argumentos resto
		| mandato_argumentos SEGUNDOPLANO {
			ordenActual(orden)->segundoplano = 1;
			#ifdef DEBUG_SINTACTICO
				printf("Segundo plano (&)\n");
			#endif
		} resto
;

mandato_argumentos:
		IDENTIFICADOR {
			struct Palabra *cmd = (struct Palabra *)malloc(sizeof(struct Palabra));
inicializarPalabra(cmd);
			expandirPalabra(cmd,$1);
			if(cmd->valor) insertarComando(ordenActual(orden),cmd->valor); // Se toma la
1Âª expansiÃ³n por orden alfabÃ©tico
			else insertarComando(ordenActual(orden),$1);
			#ifdef DEBUG_SINTACTICO
				printf("Comando: %s\n",$1);
			#endif
		}
		| mandato_argumentos IDENTIFICADOR {
			struct Palabra *arg = (struct Palabra *)malloc(sizeof(struct Palabra));
inicializarPalabra(arg);
			expandirPalabra(arg,$2);
			if(arg->valor){
				while(arg){
					insertarArgumento(ordenActual(orden),arg->valor);
					arg = arg->siguiente;
				}
			}else insertarArgumento(ordenActual(orden),$2);
			#ifdef DEBUG_SINTACTICO
				printf("Argumento: %s\n",$2);
			#endif
		}
		| mandato_argumentos redireccion
;

resto:
```

```
		| PIPE {
			ordenActual(orden)->pipe = 1;
			insertarOrden(orden);
			#ifdef DEBUG_SINTACTICO
				printf("Pipe (|)\n");
			#endif
		} comando
		| PUNTOCOMA {
			ordenActual(orden)->puntocoma = 1;
			insertarOrden(orden);
			#ifdef DEBUG_SINTACTICO
				printf("Punto y coma (;)\n");
			#endif
		} comando
		| PUNTOCOMA
;

redireccion:
		REDIRECCIONANEXAR IDENTIFICADOR {
			insertarFanexar(ordenActual(orden),$2,$1);
			#ifdef DEBUG_SINTACTICO
				printf("Anexar (");
				if($1 != STDOUT_FILENO) printf("%i",$1);
				printf(">> %s)\n",$2);
			#endif
		}
		| REDIRECCIONSALIDA IDENTIFICADOR {
			insertarFsalida(ordenActual(orden),$2,$1);
			#ifdef DEBUG_SINTACTICO
				printf("Salida (");
				if($1 != STDOUT_FILENO) printf("%i",$1);
				printf("> %s)\n",$2);
			#endif
		}
		| REDIRECCIONENTRADA IDENTIFICADOR {
			insertarFentrada(ordenActual(orden),$2);
			#ifdef DEBUG_SINTACTICO
				printf("Entrada (< %s)\n",$2);
			#endif
		}
;

%%

yyerror(char *s) {
		valorError = 1;
		if(strcmp(s,"syntax error") == 0){
			fprintf(stderr, "Error de sintaxis");
			if(yylval.valor && (strcmp(yylval.valor,"\n") != 0)) printf(" en %s\n",yytext);
			else printf("\n");
		}
		else fprintf(stderr,"%s\n",s);
		// Para vaciar el buffer de entrada en caso de error
```

```
		if(!yylval.valor || (strcmp(yylval.valor,"\n") != 0)) while(yylex() != SALTOLINEA);
}

void matarHijo(){
		if(pid) kill(pid,SIGINT);
}

int main(int argc, char *argv[]) {
		++argv, --argc;  // Se salta el nombre del programa
		if (argc) yyin = fopen(argv[0], "r");
		else	yyin = stdin;

		// Lista de Alias, manejada por el shell
		alias = (struct Alias *)malloc(sizeof(struct Alias)); inicializarAlias(alias);

		// Control del directorio (actual) y directorio anterior, y del hostname (nombre del equipo)
		directorio = (char *)malloc(sizeof(char)*MAX_DIR);
		directorioAnterior = (char *)malloc(sizeof(char)*MAX_DIR);
		getcwd (directorio,MAX_DIR); getcwd (directorioAnterior,MAX_DIR);
		char *hostname = (char *)malloc(sizeof(char)*MAX_HOSTNAME);
		char *carpeta = (char *)malloc(sizeof(char)*MAX_DIR);

		// Control de seÃ±ales
		signal(SIGINT,matarHijo);

		int r = 0;
		while(1) {
			// InicializaciÃ³n de la estructura Orden
			orden = (struct Orden *)malloc(sizeof(struct Orden));
			inicializarOrden(orden);

			// Prompt
			gethostname(hostname,MAX_HOSTNAME);
			getcwd(directorio,MAX_DIR);
			strcpy(carpeta,rindex(directorio,'/')+1);
			if(!strlen(carpeta)) strcpy(carpeta,"/"); // Si no es el directorio '/', se quita la barra
'/'
			printf("[michele::%s@%s %s]$ ",getlogin(),hostname,carpeta);

			// AnÃ¡lisis lÃ©xico, sintÃ¡ctico y construcciÃ³n de la estructura Orden
			yylval.valor = NULL;
			r = yyparse();

			// Tratamiento de sÃmbolos especiales
			// Salto de lÃnea (no hay instrucciÃ³n)
			if(r == -2){
				liberarOrden(orden);
				continue;
			}

			// EjecuciÃ³n de la estructura Orden
			if(!valorError) r = ejecutar(orden,alias);
			else valorError = 0;
```

```
                        // Mostrar ordenes y alias en modo depuraciÃ³n
                        #ifdef DEBUG_STRUCT
                                imprimirOrden(orden);
                        #endif
                        #ifdef DEBUG_ALIAS
                                imprimirAlias(alias);
                        #endif

                        // LiberaciÃ³n de la estructura Orden
                        liberarOrden(orden);

                        // Si ejecutar devuelve r = -1 ==> exit
                        if(r == -1) break;
                }
                free(directorio); free(directorioAnterior);
                free(hostname); free(carpeta);
                return 0;
}
```

structs.h

```
#ifndef STRUCTS_H
#define STRUCTS_H

#define MAX_ARGS 256

struct Palabra {
    char *valor;
    struct Palabra *siguiente;
};

struct Orden {
    char *comando;              // Comando
    struct Palabra *argumentos; // Argumentos
    struct Palabra *fanexar;    // Ficheros de salida en modo anexar (>>)
    struct Palabra *fsalida;    // Ficheros de salida (>)
    struct Palabra *fentrada;   // Ficheros de entrada (<); no es necesario almacenarlos
    char *fanexarActual;        // Fichero de salida en modo anexar que se usarÃ¡; es excluyente con
fsalidaActual
    char *fsalidaActual;        // Fichero de salida que se usarÃ¡; es excluyente con fanexarActual
```

```
    char *fentradaActual;       // Fichero de entrada que se usarÃ¡
    int fdActual;               // Descriptor de fichero (fd) de la entrada para los redireccionamientos fd> y fd>>
    int puntocoma;              // Hay punto y coma (no hay pipe)?
    int pipe;                   // Hay pipe (no hay punto y coma)?
    int segundoplano;           // Hay segundo plano?
    struct Orden *siguiente;    // Siguiente orden, si hay punto y coma o pipe
};

struct Alias {
    char *valor;                // Alias
    char *comando;              // Orden (como ristra)
    struct Alias *siguiente;    // Siguiente alias
};

void imprimirPalabras(struct Palabra *p,char *titulo);
struct Palabra* palabraActual(struct Palabra *p);
void insertarValor(struct Palabra *p,char *v);
void inicializarPalabra(struct Palabra *p);
void insertarPalabra(struct Palabra *p);
void copiarPalabra(struct Palabra *p1,struct Palabra *p2);
void liberarPalabra(struct Palabra *p);
struct Palabra* expandirPalabra(struct Palabra *p,char *r);

void imprimirOrden(struct Orden *o);
struct Orden* ordenActual(struct Orden *o);
void inicializarOrden(struct Orden *o);
void insertarComando(struct Orden *o,char *cmd);
void insertarArgumento(struct Orden *o,char *arg);
void insertarFanexar(struct Orden *o,char *f, int fd);
void insertarFsalida(struct Orden *o,char *f, int fd);
void insertarFentrada(struct Orden *o,char *f);
void insertarOrden(struct Orden *o);
```

```
void copiarOrden(struct Orden *o1,struct Orden *o2);
void fusionarOrden(struct Orden *o1,struct Orden *o2);
void liberarOrden(struct Orden *o);
char **vectorArgv(struct Orden *o,struct Palabra *argAdicional);
void crearFicherosOrden(struct Orden *o);

void imprimirAlias(struct Alias *a);
struct Alias* aliasActual(struct Alias *a);
void inicializarAlias(struct Alias *a);
void insertarAliasValor(struct Alias *a,char *alias);
void insertarAliasComando(struct Alias *a,char *cmd);
void insertarAlias(struct Alias *a,char *alias,char *cmd);
void eliminarAlias(struct Alias *a,char *alias);
void liberarAlias(struct Alias *a);
struct Alias* buscarAlias(struct Alias *a,char *alias);

#endif
```

structs.c

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <dirent.h>
#include <regex.h>
#include <stdio.h>
#include "structs.h"

// Funciones para la estructura Palabra
void imprimirPalabras(struct Palabra *p, char *titulo){
   static int i = 1;
   if(p){
```

```
        if(p->valor) printf("%s %i = %s\n",titulo,i++,p->valor);
        imprimirPalabras(p->siguiente,titulo);
    }else i = 1;
}

struct Palabra* palabraActual(struct Palabra *p){
    if(p == NULL) return p; // La estructura Palabra no deberÃa ser nula
    if(p->siguiente == NULL) return p;
    return palabraActual(p->siguiente);
}

void insertarValor(struct Palabra *p, char *v){
    p->valor = (char *)malloc(sizeof(char)*(strlen(v)+1));
    strcpy(p->valor,v);
}

void inicializarPalabra(struct Palabra *p){
    p->valor = NULL;
    p->siguiente = NULL;
}

void insertarPalabra(struct Palabra *p){
    // La Palabra no debe ser nula
    // (de lo contrario, las modificaciones no se verÃan externamente)
    if(p->siguiente) insertarPalabra(p->siguiente);
    else{
        p->siguiente = (struct Palabra *)malloc(sizeof(struct Palabra));
        inicializarPalabra(p->siguiente);
    }
}

void copiarPalabra(struct Palabra *p1,struct Palabra *p2){
    if(!p2) p1 = NULL;  // p2 no deberÃa ser nulo
```

```
    else{
        if(p2->valor){
            p1->valor = (char *)malloc(sizeof(char)*(strlen(p2->valor)+1));
            strcpy(p1->valor,p2->valor);
        }else p1->valor = NULL;
        if(p2->siguiente){
            p1->siguiente = (struct Palabra *)malloc(sizeof(struct Palabra));
            copiarPalabra(p1->siguiente,p2->siguiente);
        }else p1->siguiente = NULL;
    }
}

void liberarPalabra(struct Palabra *p){
    if(p){
        free(p->valor);
        liberarPalabra(p->siguiente);
        free(p);
    }
}

int compararExpresionRegular(char *r, char *patron){
   regex_t er;
   if (regcomp(&er,patron,0) != 0) return -1;  // Error compilando la expresiÃ³n regular
   return !regexec(&er,r,(size_t)0,NULL,0);    // Devuelve 1 si hay coincidencia con la expresiÃ³n regular
}

char* expresionRegular(char *r){
    int i = 0, k = 0;
    char *er = (char *)malloc(sizeof(char));
    er[k++] = '^';
    for(; i < strlen(r); i++){
        er = (char *)realloc(er,sizeof(char)*(k+1));
```

```
        if     (r[i] == '?') er[k++] = '.';
        else if(r[i] == '*'){er[k++] = '.';  er = (char *)realloc(er,sizeof(char)*(k+1)); er[k++] = '*';}
        else if(r[i] == '.'){er[k++] = '\\'; er = (char *)realloc(er,sizeof(char)*(k+1)); er[k++] = '.';}
        else                 er[k++] = r[i];
    }
    er = (char *)realloc(er,sizeof(char)*(k+2));
    er[k++] = '$'; er[k] = '\0';
    return er;
}

struct Palabra* expandirPalabra(struct Palabra *p,char *r){
    // Obtener los ficheros del directorio actual
    struct dirent **ficheros;
    int n = scandir(".", &ficheros, 0, alphasort);  // alphasort --> Orden alfabÃ©tico

    // Rellenar estructura Palabra con las coincidencias
    if(index(r,'?') || index(r,'*')){
        int i;
        for(i=0; i<n; i++){
            if(compararExpresionRegular(ficheros[i]->d_name,expresionRegular(r))){
                if(p->valor) insertarPalabra(p);
                insertarValor(palabraActual(p),ficheros[i]->d_name);
            }
        }
    }
    return p;
}

// Funciones para la estructura Orden
void imprimirOrden(struct Orden *o){
    static int i = 1;
    if(o){
```

```
        printf("Comando %i = %s\n",i++,o->comando);
        imprimirPalabras(o->argumentos,"Argumento");
        imprimirPalabras(o->fanexar,"Fichero de Salida en Modo Anexar");
        imprimirPalabras(o->fsalida,"Fichero de Salida");
        imprimirPalabras(o->fentrada,"Fichero de Entrada");
        if(o->fanexarActual) printf("Fichero de Salida en Modo Anexar Actual = %s\n",o->fanexarActual);
        if(o->fsalidaActual) printf("Fichero de Salida Actual = %s\n",o->fsalidaActual);
        if(o->fentradaActual) printf("Fichero de Entrada Actual = %s\n",o->fentradaActual);
        if(o->fdActual != STDOUT_FILENO) printf("Entrada para redireccionamiento Anexar/Salida = %i\n",o->fdActual);
        if(o->puntocoma) printf("Orden terminada en Punto y Coma\n");
        if(o->pipe) printf("Orden con Pipe\n");
        if(o->segundoplano) printf("Orden en Segundo Plano\n");
        printf("\n");
        imprimirOrden(o->siguiente);
    }else i = 1;
}

struct Orden* ordenActual(struct Orden *o){
    if(o == NULL) return o; // La estructura Orden no deberÃa ser nula
    if(o->siguiente == NULL) return o;
    return ordenActual(o->siguiente);
}

void inicializarOrden(struct Orden *o){
    o->comando = NULL;
    o->argumentos = NULL;
    o->fanexar = NULL;
    o->fsalida = NULL;
    o->fentrada = NULL;
    o->fanexarActual = NULL;
```

```
    o->fsalidaActual = NULL;
    o->fentradaActual = NULL;
    o->fdActual = 1;    // stdin
    o->puntocoma = 0;
    o->pipe = 0;
    o->segundoplano = 0;
    o->siguiente = NULL;
}

void insertarComando(struct Orden *o, char *cmd){
    o->comando = (char *)malloc(sizeof(char)*(strlen(cmd)+1));
    strcpy(o->comando,cmd);
}

void insertarArgumento(struct Orden *o, char *arg){
    if(!o->argumentos){ // Primer argumento
        o->argumentos = (struct Palabra *)malloc(sizeof(struct Palabra));
        inicializarPalabra(o->argumentos);
        insertarValor(o->argumentos,arg);
    }else{              // Resto de argumentos
        insertarPalabra(o->argumentos);
        insertarValor(palabraActual(o->argumentos),arg);
    }
}

void insertarFanexar(struct Orden *o, char *f, int fd){
    if(!o->fanexar){    // Primer fichero de anexar
        o->fanexar = (struct Palabra *)malloc(sizeof(struct Palabra));
        inicializarPalabra(o->fanexar);
        insertarValor(o->fanexar,f);
    }else{              // Resto de ficheros de anexar
        insertarPalabra(o->fanexar);
```

```
        insertarValor(palabraActual(o->fanexar),f);
    }
    if(o->fanexarActual) free(o->fanexarActual);
    o->fanexarActual = (char *)malloc(sizeof(char)*(strlen(f)+1));
    strcpy(o->fanexarActual,f);
    if(o->fsalidaActual) {free(o->fsalidaActual); o->fsalidaActual = NULL;}
    o->fdActual = fd;
}

void insertarFsalida(struct Orden *o, char *f, int fd){
    if(!o->fsalida){    // Primer fichero de salida
        o->fsalida = (struct Palabra *)malloc(sizeof(struct Palabra));
        inicializarPalabra(o->fsalida);
        insertarValor(o->fsalida,f);
    }else{              // Resto de ficheros de salida
        insertarPalabra(o->fsalida);
        insertarValor(palabraActual(o->fsalida),f);
    }
    if(o->fsalidaActual) free(o->fsalidaActual);
    o->fsalidaActual = (char *)malloc(sizeof(char)*(strlen(f)+1));
    strcpy(o->fsalidaActual,f);
    if(o->fanexarActual) {free(o->fanexarActual); o->fanexarActual = NULL;}
    o->fdActual = fd;
}

void insertarFentrada(struct Orden *o, char *f){
    if(!o->fentrada){   // Primer fichero de entrada
        o->fentrada = (struct Palabra *)malloc(sizeof(struct Palabra));
        inicializarPalabra(o->fentrada);
        insertarValor(o->fentrada,f);
    }else{              // Resto de ficheros de entrada
        insertarPalabra(o->fentrada);
```

```
      insertarValor(palabraActual(o->fentrada),f);
   }
   if(o->fentradaActual) free(o->fentradaActual);
   o->fentradaActual = (char *)malloc(sizeof(char)*(strlen(f)+1));
   strcpy(o->fentradaActual,f);
}

void insertarOrden(struct Orden *o){
   // La Orden no debe ser nula
   // (de lo contrario, las modificaciones no se verÃan externamente)
   if(o->siguiente) insertarOrden(o->siguiente);
   else{
      o->siguiente = (struct Orden *)malloc(sizeof(struct Orden));
      inicializarOrden(o->siguiente);
   }
}

void copiarOrden(struct Orden *o1,struct Orden *o2){
   if(!o2) o1 = NULL;  // o2 no deberÃa ser nulo
   else{
      if(o2->comando){
         o1->comando = (char *)malloc(sizeof(char)*(strlen(o2->comando)+1));
         strcpy(o1->comando,o2->comando);
      }else o1->comando = NULL;
      if(o2->argumentos){
         o1->argumentos = (struct Palabra *)malloc(sizeof(struct Palabra));
         copiarPalabra(o1->argumentos,o2->argumentos);
      }else o1->argumentos = NULL;
      if(o2->fanexar){
         o1->fanexar = (struct Palabra *)malloc(sizeof(struct Palabra));
         copiarPalabra(o1->fanexar,o2->fanexar);
      }else o1->fanexar = NULL;
```

```
    if(o2->fsalida){
        o1->fsalida = (struct Palabra *)malloc(sizeof(struct Palabra));
        copiarPalabra(o1->fsalida,o2->fsalida);
    }else o1->fsalida = NULL;
    if(o2->fentrada){
        o1->fentrada = (struct Palabra *)malloc(sizeof(struct Palabra));
        copiarPalabra(o1->fentrada,o2->fentrada);
    }else o1->fentrada = NULL;
    if(o2->fanexarActual){
        o1->fanexarActual = (char *)malloc(sizeof(char)*(strlen(o2->fanexarActual)+1));
        strcpy(o1->fanexarActual,o2->fanexarActual);
    }else o1->fanexarActual = NULL;
    if(o2->fsalidaActual){
        o1->fsalidaActual = (char *)malloc(sizeof(char)*(strlen(o2->fsalidaActual)+1));
        strcpy(o1->fsalidaActual,o2->fsalidaActual);
    }else o1->fsalidaActual = NULL;
    if(o2->fentradaActual){
        o1->fentradaActual = (char *)malloc(sizeof(char)*(strlen(o2->fentradaActual)+1));
        strcpy(o1->fentradaActual,o2->fentradaActual);
    }else o1->fentradaActual = NULL;
    o1->fdActual = o2->fdActual;
    o1->puntocoma = o2->puntocoma;
    o1->pipe = o2->pipe;
    o1->segundoplano = o2->segundoplano;
    if(o2->siguiente){
        o1->siguiente = (struct Orden *)malloc(sizeof(struct Orden));
        copiarOrden(o1->siguiente,o2->siguiente);
    }else o1->siguiente = NULL;
  }
}
```

```
void fusionarOrden(struct Orden *o1,struct Orden *o2){
   if(o2){
      struct Orden *oActual = (struct Orden *)malloc(sizeof(struct Orden));
      oActual = ordenActual(o1);
      struct Palabra *aux = (struct Palabra *)malloc(sizeof(struct Palabra));
      if(o2->argumentos) aux = o2->argumentos;
      else aux = NULL;
      while(aux){
         insertarArgumento(oActual,aux->valor);
         aux = aux->siguiente;
      }
      if(o2->fanexar) aux = o2->fanexar;
      else aux = NULL;
      while(aux){
         insertarFanexar(oActual,aux->valor,o2->fdActual);
         aux = aux->siguiente;
      }
      if(o2->fsalida) aux = o2->fsalida;
      else aux = NULL;
      while(aux){
         insertarFsalida(oActual,aux->valor,o2->fdActual);
         aux = aux->siguiente;
      }
      if(o2->fentrada) aux = o2->fentrada;
      else aux = NULL;
      while(aux){
         insertarFentrada(oActual,aux->valor);
         aux = aux->siguiente;
      }
      if(o2->fanexarActual){
         oActual->fanexarActual = (char *)malloc(sizeof(char)*(strlen(o2->fanexarActual)+1));
```

```
            strcpy(oActual->fanexarActual,o2->fanexarActual);
        }
        if(o2->fsalidaActual){
            oActual->fsalidaActual = (char *)malloc(sizeof(char)*(strlen(o2->fsalidaActual)+1));
            strcpy(oActual->fsalidaActual,o2->fsalidaActual);
        }
        if(o2->fentradaActual){
            oActual->fentradaActual = (char *)malloc(sizeof(char)*(strlen(o2->fentradaActual)+1));
            strcpy(oActual->fentradaActual,o2->fentradaActual);
        }
        oActual->fdActual = o2->fdActual;
        oActual->puntocoma = o2->puntocoma;
        oActual->pipe = o2->pipe;
        oActual->segundoplano = o2->segundoplano;
        if(o2->siguiente){
            oActual->siguiente = (struct Orden *)malloc(sizeof(struct Orden));
            inicializarOrden(oActual->siguiente);
            copiarOrden(oActual->siguiente,o2->siguiente);
        }
    }
}

void liberarOrden(struct Orden *o){
    if(o){
        free(o->comando);
        liberarPalabra(o->argumentos);
        liberarPalabra(o->fanexar);
        liberarPalabra(o->fsalida);
        liberarPalabra(o->fentrada);
        free(o->fanexarActual);
        free(o->fsalidaActual);
```

```
        free(o->fentradaActual);
        liberarOrden(o->siguiente);
        free(o);
    }
}

// Construye el vector argv, que se compone del comando seguido de los argumentos (si los hubiere)
// Se permite aÃ±adir argumentos adicionales, no indicados por el usuario
char **vectorArgv(struct Orden *o, struct Palabra *argAdicionales){
    struct Palabra *aux = o->argumentos;
    int i;
    char **arg;

    for(i = 0; aux != NULL; aux = aux->siguiente, i++);
    aux = argAdicionales;
    for(; aux != NULL; aux = aux->siguiente, i++);
    arg = (char **)malloc(sizeof(char *)*(i+2));    // Argumentos + Comando + NULL
    arg[0] = o->comando;
    aux = argAdicionales;
    for(i = 1; aux != NULL; aux = aux->siguiente, i++) arg[i] = aux->valor;
    aux = o->argumentos;
    for (; aux != NULL; aux = aux->siguiente, i++) arg[i] = aux->valor;
    arg[i] = NULL;

    return arg;void copiarOrden(struct Orden *o1,struct Orden *o2);
}

void crearFicheros(struct Palabra *p){
    if(p){
        int fd;
        if((fd = open(p->valor, O_CREAT, S_IRUSR | S_IWUSR)) == -1)
            printf("No se puede abrir %s para anexar\n",p->valor);
```

```
        close(fd);
        crearFicheros(p->siguiente);
    }
}

void crearFicherosOrden(struct Orden *o){
    crearFicheros(o->fanexar);
    crearFicheros(o->fsalida);
}

// Funciones para la estructura Alias
void imprimirAlias(struct Alias *a){
    static int i = 1;
    if(a){
        if(a->valor && a->comando) printf("Alias %i: %s = %s",i++,a->valor,a->comando);
        imprimirAlias(a->siguiente);
    }else i = 1;
}

struct Alias* aliasActual(struct Alias *a){
    if(a == NULL) return a; // La estructura Alias no deberÃa ser nula
    if(a->siguiente == NULL) return a;
    return aliasActual(a->siguiente);
}

void inicializarAlias(struct Alias *a){
    a->valor = NULL;
    a->comando = NULL;
    a->siguiente = NULL;
}

void insertarAliasValor(struct Alias *a,char *alias){
    a->valor = (char *)malloc(sizeof(char)*(strlen(alias)+1));
```

```
    strcpy(a->valor,alias);
}

void insertarAliasComando(struct Alias *a,char *cmd){
    a->comando = (char *)malloc(sizeof(char)*(strlen(cmd)+1));
    strcpy(a->comando,cmd);
}

void insertarAlias(struct Alias *a,char *alias,char *cmd){
    // El Alias no debe ser nulo
    // (de lo contrario, las modificaciones no se verÃan externamente)
    if(!a->valor){insertarAliasValor(a,alias); insertarAliasComando(a,cmd);}
    else if(strcmp(a->valor,alias) == 0) insertarAliasComando(a,cmd);
    else if(a->siguiente) insertarAlias(a->siguiente,alias,cmd);
    else{
        a->siguiente = (struct Alias *)malloc(sizeof(struct Alias));
        inicializarAlias(a->siguiente);
        insertarAliasValor(a->siguiente,alias); insertarAliasComando(a->siguiente,cmd);
    }
}

void eliminarAlias(struct Alias *a,char *alias){
    if(strcmp(a->valor,alias) == 0){
        if(a->siguiente){
            insertarAliasValor(a,a->siguiente->valor); free(a->siguiente->valor);
            insertarAliasComando(a,a->siguiente->comando); free(a->siguiente->comando);
            a->siguiente = a->siguiente->siguiente;

        }else{
            free(a->valor); free(a->comando); inicializarAlias(a);
        }
    }else if(a->siguiente){
```

```
        if(strcmp(a->siguiente->valor,alias) == 0){
            free(a->siguiente->valor);
            free(a->siguiente->comando);
            a->siguiente = a->siguiente->siguiente;
        }else eliminarAlias(a->siguiente,alias);
    }
}

void liberarAlias(struct Alias *a){
    if(a){
        free(a->valor);
        free(a->comando);
        liberarAlias(a->siguiente);
        free(a);
    }
}

struct Alias* buscarAlias(struct Alias *a,char *alias){
    if(!a) return NULL;
    if(a->valor && (strcmp(a->valor,alias) == 0)) return a;
    return buscarAlias(a->siguiente,alias);
}
```