



UNIVERSIDAD DE BUENOS AIRES
FACULTAD DE INGENIERÍA
Año 2018 - 1^{er} cuatrimestre

ALGORITMOS Y PROGRAMACIÓN I (95.11)

TRABAJO PRÁCTICO N.º ~~1~~

TEMA: ~~Diseño y desarrollo de una aplicación que permita interpretar y ejecutar un lenguaje de máquina inventado.~~

FECHA: ~~22/05/2018~~ los <> sáquenlos

INTEGRANTES:

Galli, Joaquín	- 99793
<taigalli@hotmail.com>	
Montilla, Delfina	- 99214
<delfinamontilla@gmail.com>	
Sobico, Carla	- 99738
<carlasobico@hotmail.com>	

¿quién se hace responsable de qué?

Índice

1. Enunciado	2
2. Alternativas consideradas	3
2.1. Estrategias adoptadas	3
3. Estructura Funcional	5
4. Problemas encontrados	6
4.1. Contador	6
4.1.1. Posibles soluciones	6
4.1.2. Solución adoptada	6
4.2. Estructura para el estado	6
4.2.1. Posibles soluciones	7
4.2.2. Solución adoptada	7
4.3. Violación de segmento	8
4.3.1. Posibles soluciones	8
4.3.2. Solución adoptada	8
4.4. Doble free o error de corrupción	8
4.4.1. Posibles soluciones	8
4.4.2. Solución adoptada	8
4.5. Nueva violación de segmento	8
4.5.1. Posibles soluciones	8
4.5.2. Solución adoptada	8
5. Resultados de ejecución	9
6. Referencias	10
7. Código fuente	11
7.1. Archivos <i>.c</i>	11
7.2. Archivos <i>.h</i>	21

1. Enunciado

Se ha de crear una “computadora” a la cual llamaremos Simpletron, que ejecuta programas escritos en un lenguaje diseñado específicamente para este problema, el Lenguaje de Máquina de la Simpletron, LMS. Este lenguaje y ejercicio brindarán un mayor entendimiento de lo que ocurre en los lenguajes de programación más cercanos al hardware.

Toda la información que utiliza la Simpletron se maneja en palabras; cada una es un número entero de cuatro dígitos con signo. Le Simpletron tiene espacio para almacenar una cantidad fija y finita de palabras, la memoria. En ella, se pueden almacenar instrucciones del programa, un dato o nada. Todas las instrucciones se deben cargar en memoria, por lo que las instrucciones son palabras. Además, deben ser siempre positivos. Los primeros dos dígitos de cada instrucción representan el código de operación, *opcode*, el cual especifica la operación a realizar. En la tabla 1.1 se resumen los códigos de operación que debe soportar el programa. Los últimos dos dígitos de la instrucción son el *operando*, el cual representa la dirección de memoria que contiene la palabra a la que se le aplica la operación.

Operación	OpCode	Descripción
<i>Op.de Entrada/Salida:</i>		
LEER	10	Lee una palabra de <i>stdin</i> a una posición de memoria
ESCRIBIR	11	Imprime por <i>stdout</i> una posición de memoria
<i>Op.de movimiento:</i>		
CARGAR	20	Carga una palabra de la memoria en el acumulador
GUARDAR	21	Guarda una palabra de la memoria en el acumulador
PCARGAR	22	Carga una palabra de la memoria en el acumulador pero el operando es <i>puntero</i>
PGUARDAR	23	Guarda una palabra de la memoria en el acumulador pero el operando es <i>puntero</i>
<i>Op.de aritméticas:</i>		
SUMAR	30	Suma una palabra al acumulador
RESTAR	31	Resta una palabra al acumulador
DIVIDIR	32	Divide el acumulador por el operando
MULTIPLICAR	33	Multiplica el acumulador por el operando
<i>Op.de movimiento:</i>		
JMP	40	Salta a una ubicación de memoria
JMPNEG	41	Idem sólo si el acumulador es negativo
JMPZERO	42	Idem sólo si el acumulador es cero
JNZ	43	Idem sólo si el acumulador NO es cero
DJNZ	44	Decrementa sólo si el acumulador NO es cero
HALT	45	Finaliza el programa

Cuadro 1.1: Códigos de operación—*opcodes*—del LMS

Además, la Simpletron posee un *acumulador*, un registro en el cual la información se coloca antes de ser utilizada por las distintas instrucciones que soporta la máquina. Antes de ejecutar un programa escrito en lenguaje LMS, el mismo debe ser cargado en memoria. Siempre, la primera instrucción de cada programa se carga en la posición 00.

Finalizada la ejecución del programa, se hará un volcado de la aplicación, mostrando todos sus parámetros y el estado de la memoria. Este volcado se conoce como *dump*.

Podrá ser configurado con algunos pocos parámetros. Mediante estos parámetros se podrá configurar la cantidad de memoria que dispone el Simpletron, el formato de salida, el archivo de salida, el archivo de entrada y el formato de la entrada. En la tabla 1.2 se detallan los argumentos que recibirá el programa.

Arg.	Opción	Descripción
-h	no posee	Muestra una ayuda
-m	N	<i>Simpletron</i> tiene una memoria de N palabras. Si no se da el argumento, por omisión tendrá 50 palabras.
-i	archivo	El programa se leerá del archivo pasado como opción, en caso contrario, de <i>stdin</i> .
-ia	bin	El archivo de entrada se entenderá como una secuencia binaria de enteros que representan las palabras que forman el programa.
	txt	El archivo de entrada se interpretará como secuencia de números, cada uno en una única línea.
-o	archivo	<i>Eldump</i> se hará en el archivo pasado como opción, si no pasa el argumento, el <u>volcado</u> se hará por <i>stdout</i> .
-of	bin	El volcado se <u>hará en</u> binario guardando cada elemento de la estructura del <i>Simpletron</i> , además de la memoria
	txt	El volcado se hará en formato de texto, imprimiendo los registros de la memoria.

Cuadro 1.2: Tabla de argumentos del programa principal

2. Alternativas consideradas

Antes de comenzar el desarrollo del código se enlistaron los posibles problemas donde se debería tomar una decisión de como ejecutarlo. Estos fueron:

1. Usar argumentos posicionales o no posicionales.
2. Cómo leer los argumentos y la implementación de los *default* de los mismos.
3. Cuál es la máxima cantidad de palabras que se pueden ingresar.
4. Qué hacer si no especifica si el archivo ingresado es binario o de texto.
5. Qué tipo de redacción se desea en los archivos ingresados.
6. La impresión final, sobrescribe o no el archivo de salida.
7. Qué hacer en caso de que pase un archivo que no coincida con el tipo indicado.
8. Cómo leer los datos ingresados (para cada una de las palabras), tomando sólo la parte necesaria de estos.
9. Cómo separar las palabras en su signo, *opcode* y *operando*.
10. Qué número establecer como cierre de ingreso de palabras por *stdin*.
11. Qué manera es más rápida en la selección de qué opcode se está usando para las operaciones.
12. Cómo imprimir en el archivo binario.
13. Cómo organizar los archivos “.h”.
14. ¿Es confuso que un argumento sea *-if*?
15. Usar o no *cat*.

2.1. Estrategias adoptadas

Los problemas enunciados anteriormente se discutieron llegando así a las siguientes decisiones, que fueron tomadas considerando que el usuario leyó previamente el archivo de ayuda y que sabe lo que hace.

1. Se decidió usar argumentos posicionales ya que esto simplicaría la lectura de los argumentos; para que el usuario no cometa algún error, esto se especificó en el archivo de ayuda.

2. Los *default* utilizados fueron los especificados en la consigna. Luego, cada argumento se analizó pasando por una serie de *else if* para poder implementar la función necesaria para cada caso.
3. Se tomó como máxima cantidad de palabras ingresadas 100, ya que la parte del operando de cada palabra sólo puede ir desde 00 a 99.
4. Si se llega a pasar un archivo y no se especifica si el mismo es de tipo binario o texto se imprimirá un error y se cerrará el programa.
5. Para una más fácil lectura se quiere que los archivos sean escritos de la forma “ $\pm XXXX ; dddd$ ”.
6. Se decidió que al imprimir en los archivos de salida estos sean sobrescritos, ya que se le avisó esto anteriormente al usuario en el archivo ayuda.
7. En el caso de que el archivo sea de un distinto formato al especificado, se continuará ejecutando el programa como si este archivo fuera del formato dicho, a pesar de no llegar al objetivo deseado.
8. 8.
9. La parte del signado sólo era importante en pocas funciones, por lo que esto se simplificó tomando el número y viendo si era mayor o menor a 0 para luego continuar. Para separar el *opcode* y el *operando*, en caso que el signado sea positivo, se dividió el número por 100, como éste era un int esta división deja unicamente lo que sería el *opcode*. Para encontrar el *operando*, que son los dos últimos dígitos del número se busca restando el número con el *opcode* multiplicado previamente por 100.
10. Se utilizó como cierre el “-99999 ” ya que éste fue sugerido en un ejemplo dado.
11. Para decidir que operación se debe utilizar según el opcode recibido, se decidió utilizar la función *Switch* teniendo cada *case* la función a utilizar para ese *opcode*. no es una función, es una estructura de control
12. Se decidió imprimir en el archivo binario la información en igual orden que para el archivo de texto y por pantalla, pero sin señalar con texto que es cada cosa y la memoria en forma de lista en vez de forma de matriz. ✓(si sale en formato binario ...)
13. Se decidió utilizar tres de estos archivos, uno para español, otro para inglés y otro para las estructuras y prototipos. Se consideró hacer de esta manera ya que las estructuras y los prototipos se deben ingresar independientemente del idioma y al no poner todo en un mismo archivo se puede hacer una actualización del programa a otro idioma de manera más simple y sin correr riesgo que se dañe otra parte del programa. Se pensó también en crear los prototipos y las estructuras en archivos separados, pero al no ser muchas se decidió dejarlo todo en uno. En caso de haber implementado más se hubieran separado.
14. Se consideró que el argumento *-if* era confuso por lo que se lo nombro *-ia* reemplazando la inicial de *file* por la inicial de *archivo*. Hay que ser consistentes, entonces cambiar of por oa.
15. Se decidió no usar *cat* ya que con los argumentos posicionales ya se pedía el archivo de entrada por línea de comando, por lo que se consideró innecesario su uso.

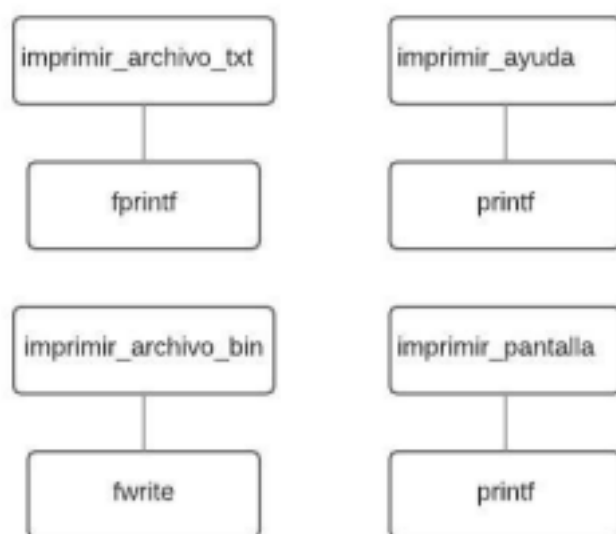


Figura 3.3: Funciones que imprimen

4. Problemas encontrados

A continuación se detallan los problemas encontrados, sus posibles soluciones, la solución adoptada y una justificación de la misma.

4.1. Contador

Se generó un problema al llegar a las operaciones de *opcode* en las que se generaba un salto en el contador. Como en primera instancia se implementó un aumento del contador al salir de la selección de operaciones, sucedía que se incrementaba en uno la posición deseada.

4.1.1. Posibles soluciones

- Modificar las funciones de salto haciendo que salga de las mismas en una posición menos de la deseada.
- Al ingresar en cada *case* aumentar el contador y en los casos de los saltos analizarlo dentro de cada *case*.

4.1.2. Solución adoptada

El problema de la primera opción era en el caso de querer ir a la posición 00. Por lo que se utilizó la segunda opción.

4.2. Estructura para el estado

En un principio, el estado de la Simpletron no había sido definido en una estructura. Es decir, el *acumulador*, *contador del programa*, *opcode*, *operando* y vector de instrucciones, *palabras*, eran variables independientes. Esto generaba que al momento de implementar una nueva función, los argumentos eran extensos. Además no se cumplía con lo pedido en la sección de Implementación acerca de la realización del trabajo práctico.



Figura 3.4: Funciones de lectura

4.2.1. Posibles soluciones

Se debía crear la estructura *estado* necesariamente.

4.2.2. Solución adoptada

Al crearse la nueva estructura, se procedió a realizar los cambios necesarios en la escritura del programa. A raíz del anterior cambio, se tuvo algunos problemas al momento de imprimir un vector en una posición especificada por un puntero, siendo los dos miembros de *estado*. Al momento de compilación, aparecía un error que marcaba que el subíndice no era un número entero; o en otro caso, «el formato

‘%i’ espera el argumento de tipo ‘int’, pero el argumento tiene el tipo ‘int *’»

4.3. Violación de segmento

Después de arreglar los errores y warnings indicados por el compilador se procedió a ejecutar el programa en su modo default; es decir, la cantidad de palabras es 50, la entrada de información se realiza por stdin y la salida por stdout. Subsecuentemente, apareció la advertencia de violación de segmenteo y se cerró el programa (el mensaje recibido se puede ver en la sección de ejecución 5), Error 1.

4.3.1. Posibles soluciones

Los punteros estado y params estaban a NULL, por lo que se debía crear una estructura a la cual apunte cada uno correspondientemente.

4.3.2. Solución adoptada

Se creó una estructura para las variables llamada Simpletron, que fue apuntada por el puntero anterior estado; y una estructura para los argumentos, llamada argumentos, con un puntero a ella, params.

4.4. Doble free o error de corrupción

Se procedió a ejecutar el programa en su modo default y el mensaje recibido se puede ver en la sección de ejecución 5. Buscamos información al respecto y los resultados se encuentran en las referencias.

4.4.1. Posibles soluciones

La causa de este error pueden ser por llamar dos veces a fclos, liberar memoria más de una vez o estar escribiendo en la memoria fuera de las regiones asignadas. Por otro lado, cerrar los archivos en funciones donde fueron pasados como parámetros no es recomendado. ✓

4.4.2. Solución adoptada

Se rehizo la implementación de las funciones free() y fclose(), evitando que cierren archivos que no fueron abiertos o que se libere memoria que no había sido creada.

4.5. Nueva violación de segmento

Se procedió a ejecutar el programa con «./main 50 suma.lms txt salida.txt txt» y el mensaje recibido se puede ver en la sección de ejecución 5, Error 4. Además notamos que no están bien validados los argumentos en caso de querer que la entrada sea por stdin o la entrada por stdout.

4.5.1. Posibles soluciones

4.5.2. Solución adoptada

¿Acá es cuando quisieron golpearme?

¿ah? No les creo que hayan implementado free() y fclose(). A lo sumo, le agregaron código alrededor.

5. Resultados de ejecución

Al compilar con `gcc -ansi -Wall -pedantic main.c -o main` y ejecutar la línea de comando `./main -h`, se obtiene la ayuda pedida. Sin embargo al momento de utilizar «./main -50 - - -», se obtiene un error de segmentación. Los resultados siguientes son los mostrados por *GNU gdb (Ubuntu 7.7.1-0ubuntu5 14.04.3) 7.7.1* al momento de ejecutarlo para encontrar el origen del problema.

Error 1:

Starting program:

```
~/GitHub/trabajopractico1/main -50 - - - -
```

Program received signal SIGSEGV, Segmentation fault.

```
0x0000000000400f84 in validar_argumentos (argc=6, argv=0x7ffffffde3c8,
params=0x0, estado=0x0, FENTRADA=0x0, FSALIDA=0x0) at main.c:163
163 params->cant_palabras = strtol(argv[ARG_POS_CANT_PALABRAS], &pc, 10);
```

Error 2:

Starting program:

```
~/GitHub/trabajopractico1/main -50 - - - -
```

ERROR: number of arguments

```
*** Error in ~/GitHub/trabajopractico1/main':
free(): invalid pointer: 0x00007ffffffde2c0 ***
```

Program received signal SIGABRT, Aborted.

```
0x00007ffffff066c37 in __GI_raise (sig=sig@entry=6) at
../nptl/sysdeps/unix/sysv/linux/raise.c:56
56 ../nptl/sysdeps/unix/sysv/linux/raise.c: No such file or directory.
```

Error 3:

Starting program:

```
~/GitHub/trabajopractico1/main 50 - - - -
```

ERROR: opening file

```
*** Error in ~/GitHub/trabajopractico1/main':
double free or corruption (out): 0x0000000000400910 ***
```

Program received signal SIGABRT, Aborted.

```
0x00007ffffff066c37 in __GI_raise (sig=sig@entry=6) at
../nptl/sysdeps/unix/sysv/linux/raise.c:56
56 ../nptl/sysdeps/unix/sysv/linux/raise.c: No such file or directory.
```

Error 4:

Starting program:

```
~/GitHub/trabajopractico1/main 50 suma.lms txt salida.txt txt
```

Program received signal SIGSEGV, Segmentation fault.

```
0x000000007020616c in ?? ()
```

¿compilan con -g para ver dónde están los errores?

6. Referencias

- [1] <https://www.lucidchart.com>
- [2] Double Free Vulnerability Basics Explained Posted by Shashi Kiran <https://www.secpod.com/blog/double-free-vulnerability-basics-explained/>
- [3] What does this error mean? double free or corruption <https://www.linuxquestions.org/questions/programming-9/what-does-this-error-mean-double-free-or-corruption-661294/>

Para empezar, no es correcta la separación que hicieron en archivos. Todo lo referido a la simpletron debe ir en simpletron.h y simpletron.c. Comentaré más, seguramente, mientras corrija el código.

Por otro lado, después de probar varias ejecuciones del programa (claramente no funciona, como anticiparon) los mensajes de error no me dieron mucha ayuda. Lo ejecuté en inglés, no sé si en español eran más claros, pero inglés es el idioma por omisión.

7. Código fuente

7.1. Archivos .c

\inputminted{c}{main.c}

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  #include "estructuras_prototipos.h"
6
7  #define LANG_ENGLISH /*elección del idioma del programa*/
8
9  #ifdef LANG_SPANISH
10 #include "LANG_SPANISH.h"
11 #endif
12 #elif defined (LANG_ENGLISH)
13 #ifdef LANG_ENGLISH
14 #include "LANG_ENGLISH.h"
15 #endif
16 falta un #else, por si no hay ningún idioma configurado: la opción por omisión.
17 int main(int argc, char *argv[])
18 {
19     estado_t simpletron;
20     estado_t *estado;
21
22     parametros_t argumentos;
23     parametros_t *params;
24     status_t st;
25     FILE *FENTRADA, *FSALIDA;
26     params=NULL;
27     estado=NULL;
28     FENTRADA=NULL;
29     FSALIDA=NULL;
30
31     estado=&simpletron;
32     params=&argumentos;
33
34     if(argc==ARGC2_MAX){
35         if((st=validar_ayuda(argc, argv))!=ST_OK){
36             return EXIT_FAILURE;
37         }
38     }
39     else {
40         if((st=validar_argumentos(argc, argv, params, estado, FENTRADA, FSALIDA))!=ST_OK){
41             return EXIT_FAILURE;
42         }
43     }
44     estado->palabras = calloc(params->cant_palabras, sizeof(params->cant_palabras));
45
46     if(estado->palabras==NULL){
47         fprintf(stderr, "%s:%s\n",MSJ_ERROR,MSJ_ERROR_NO_MEM );
48         return EXIT_FAILURE;
49     }
50     while(st!=ST_SALIR) st=operaciones(estado);
51     seleccionar_salida(argv,params,estado,FSALIDA);
52     liberar_memoria(estado);

```

main

usar memoria dinámica

estado_t * simpletron;
parametros_t argumentos;

usar minúsculas en los nombres de las variables.

dejar una línea en blanco para separar las declaraciones del resto del código.

para esa estructura no es necesario usar un puntero.

aquí deberían validar los argumentos. Si se ingresó únicamente -h, que se los informe la función que procesa los argumentos.

todavía no leí la función que valida, pero seguro que no es necesario que reciba el "estado".

&argumentos

eso es el sizeof de un entero

ejecutar_simpletron(estado);
(no pongan todo en una línea)

53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110

```

    if(FENTRADA!=NULL)
        fclose(FENTRADA);
    if(FSALIDA!=NULL)
        fclose(FSALIDA);
}
return EXIT_SUCCESS;
}

status_t leer_archivo_bin(parametros_t *params, estado_t *estado, FILE *FENTRADA, FILE *FSALIDA)
/*recibe los punteros: a la estructura de los argumentos para poder acceder al valor de cant_palabras
a la estructura de estado para cargar las instrucciones en el vector palabras, o liberar la memoria
al archivo de entrada para poder leer los datos,
y al archivo de salida (para cerrar dos archivos en caso de error)*/
{
    int i;
    int instruccion;
    instruccion = 0;
    for(i=0; i<params->cant_palabras;i++){
        if(fread(&instruccion,sizeof(int),MAX_LARGO_PALABRA,FENTRADA)!=MAX_LARGO_PALABRA){
            liberar_memoria(estado);
            fclose(FENTRADA);
            return ST_ERROR_FUERA_RANGO;
        }
        if(instruccion==FIN)
            return ST_OK;
        if(instruccion<MIN_PALABRA||instruccion>MAX_PALABRA)
            return ST_ERROR_FUERA_RANGO;
        estado->palabras[i]=instruccion;
    }
    printf("%s\n",MSJ_CARGA_COMPLETA);
    printf("%s\n",MSJ_COMIENZO_EJECUCION);
    return ST_OK;
}

status_t leer_teclado(parametros_t *params, estado_t *estado)
/*recibe los punteros: a la estructura de los argumentos para poder acceder al valor de cant_palabras
a la estructura de estado para cargar las instrucciones en el vector palabras*/
{
    int i;
    char aux[MAX_LARGO_PALABRA];
    long instruccion;
    char *pc;
    instruccion = 0;
    printf("%s\n",MSJ_BIENVENIDA);
    for(i=0; i<params->cant_palabras;i++){
        printf("%2.i ? ", i);
        if(fgets(aux,MAX_LARGO_PALABRA,stdin)==NULL){
            liberar_memoria(estado);
            return ST_ERROR_FUERA_RANGO;
        }
        instruccion = strtol(aux,&pc,10);
        if(*pc!='\0'&& *pc!='\n')
            return ST_ERROR_NO_NUMERICO;
        if(instruccion==FIN)
            return ST_OK;
        break;
    }
}

```

¿para qué recibe params? sólo necesita el tamaño. Sin embargo, pueden optar por cargar "estado" inmediatamente con el largo pedido, en cuyo caso sólo recibiría el estado y el archivo de entrada.

está de más

¿se corta algo ahí?

se encargan afuera de la fx.

instrucción es un (1) int y están queriendo leer ahí MAX_LARGO_PALABRA ints: no hay lugar suficiente. Tienen que pedir memoria para las instrucciones antes de comenzar a leer. Después puede ocurrir que haya más o menos instrucciones de las esperadas.

en el binario no debería haber. Leen hasta que se acaba el archivo o alcanzan el el tope dado por la memoria.

¡NO! ¡¡¡NO IMPRIME MENSAJES!!! La función se llama "leer_archivo_bin" ¿¿dónde dice que imprime algo?! ¡No tiene que imprimir nada!

stdin.txt

esta línea en blanco está de más.

indentar

ponemosle que es válida porque interactúa con las personas.

el ? se puede #definir en una constante.

están forzando a que haya igual cantidad de palabras que de memoria configurada. No está documentado así y no debería ser el caso.

si fuese necesario, liberar memoria

break! que muestre el mensaje de que se cargó todo

```

111         if(instruccion<MIN_PALABRA||instruccion>MAX_PALABRA)
112             return ST_ERROR_FUERA_RANGO; si fuese necesario, liberar memoria.
113         estado->palabras[i]=instruccion;
114     }
115     printf("%s\n",MSJ_CARGA_COMPLETA);
116     printf("%s\n",MSJ_COMIENZO_EJECUCION);
117     return ST_OK;
118 }
119 lo mismo que en las funciones anteriores. No es necesario recibir tanto.
120 status_t leer_archivo_txt(parametros_t *params, estado_t *estado, FILE *FENTRADA, FILE *FSALIDA)
121 /*recibe los punteros: a la estructura de los argumentos para poder acceder al valor de cant_palabr
122 a la estructura de estado para cargar las instrucciones en el vector palabras,
123 al archivo de entrada para poder leer los datos,
124 y al archivo de salida (para cerrar los dos archivos en caso de error)*/
125 {
126     size_t i;
127     char aux[MAX_LARGO_PALABRA];
128     long instruccion;
129     char *pc;
130     instruccion = 0;
131
132     for(i=0; i<params->cant_palabras;i++){
133         if(fgets(aux,MAX_LARGO_PALABRA,FENTRADA)==NULL){
134             liberar_memoria(estado);
135             fclose(FENTRADA); Esta función no abre el archivo. Esta función no lo cierra.
136             return ST_ERROR_FUERA_RANGO;
137         }
138         instruccion = strtol(aux,&pc,10); -dijeron que aceptan comentarios y no cumplen.
139         if(*pc!='\0'&& *pc!='\n')
140             return ST_ERROR_NO_NUMERICO;
141         /*validar que " " no sea un 0*/
142         if(instruccion<MIN_PALABRA||instruccion>MAX_PALABRA)
143             return ST_ERROR_FUERA_RANGO;
144         estado->palabras[i]=instruccion;
145     }
146     printf("%s\n",MSJ_CARGA_COMPLETA);
147     printf("%s\n",MSJ_COMIENZO_EJECUCION);
148     return ST_OK;
149 }
150
151 status_t validar_argumentos (int argc , char *argv[], parametros_t *params, estado_t *estado, FILE *
152 /*recibe arc y argv para realizar las validacione correspondientes a su cantidad y contenido;
153 además recibe los punteros: a la estructura de los argumentos para poder cargar el valor a cant_pal
154 a la estructura de estado para después ser pasada a la funcion de lectura correspondiente,
155 al archivo de entrada para poder leer los datos,
156 y al archivo de salida para poder escribir los datos*/
157 {
158     char *pc=NULL;
159     if(!argv){
160         fprintf(stderr, "%s: %s\n", MSJ_ERROR, MSJ_ERROR_PTR_NULO ); ¡NO! Los mensajes se
161         return ST_ERROR_PTR_NULO; imprimen en la función
162     } invocante.
163
164     if(ARGC_MAX!=argc){
165         fprintf(stderr, "%s: %s\n", MSJ_ERROR, MSJ_ERROR_CANT_ARG );
166         return ST_ERROR_CANT_ARG;
167     }
168

```

```

169     if(argv[ARG_POS_CANT_PALABRAS]==NULL){ //no puede ser NULL, a lo sumo puede ser "-" según
170         params->cant_palabras=CANT_PALABRAS_DEFAULT; su documentación.
171     }
172     else {
173         params->cant_palabras = strtol(argv[ARG_POS_CANT_PALABRAS], &pc, 10);
174         if(params->cant_palabras<0 || (*pc!='\0' && *pc!='\n') || params->cant_palabras>100){
175             fprintf(stderr, "%s: %s\n", MSJ_ERROR, MSJ_ERROR_CANT_PALABRAS);
176             return ST_ERROR_CANT_PALABRAS;
177         }
178     }
179     if(argv[ARG_POS_FENTRADA2]!=NULL){ idem, no puede ser NULL.
180         if(strcmp(argv[ARG_POS_FENTRADA2], OPCION_TXT)){ strcmp() retorna 0 cuando son iguales.
181             if((FENTRADA=fopen(argv[ARG_POS_FENTRADA1], "rt"))==NULL){
182                 fprintf(stderr, "%s: %s\n", MSJ_ERROR, MSJ_ERROR_APERTURA_ARCHIVO);
183             *FENTRADA return ST_ERROR_APERTURA_ARCHIVO;
184             (tiene que ser puntero doble a FILE)
185             leer_archivo_txt(params, estado, FENTRADA, FSALIDA); esto se hace afuera de la fx.
186         }
187         else if (strcmp(argv[ARG_POS_FENTRADA2], OPCION_BIN)){
188             if((FENTRADA=fopen(argv[ARG_POS_FENTRADA1], "rb"))==NULL){
189                 fprintf(stderr, "%s: %s\n", MSJ_ERROR, MSJ_ERROR_APERTURA_ARCHIVO);
190                 return ST_ERROR_APERTURA_ARCHIVO;
191             }
192             leer_archivo_bin(params, estado, FENTRADA, FSALIDA);
193         }
194     } todavía no leí que son ARG_POS_ENTRADA1 y 2 y no me queda claro esto..
195     else if(argv[ARG_POS_FENTRADA2]==NULL && argv[ARG_POS_FENTRADA1]!=NULL){
196         fprintf(stderr, "%s: %s\n", MSJ_ERROR, MSJ_ERROR_APERTURA_ARCHIVO);
197         return ST_ERROR_APERTURA_ARCHIVO;
198     }
199     else
200         leer_teclado(params, estado); esto se hace afuera de la fx, una vez procesados
201                                     todos los argumentos correctamente.
202     if(argv[ARG_POS_FSALIDA2]!=NULL){
203         if(strcmp(argv[ARG_POS_FENTRADA2], OPCION_TXT)){
204             if((FSALIDA=fopen(argv[ARG_POS_FSALIDA1], "wt"))==NULL){
205                 fprintf(stderr, "%s: %s\n", MSJ_ERROR, MSJ_ERROR_APERTURA_ARCHIVO);
206                 fclose(FSALIDA);
207                 return ST_ERROR_APERTURA_ARCHIVO;
208             }
209         }
210         else if (strcmp(argv[ARG_POS_FENTRADA2], OPCION_BIN)){
211             if((FSALIDA=fopen(argv[ARG_POS_FSALIDA1], "wb"))==NULL){
212                 fprintf(stderr, "%s: %s\n", MSJ_ERROR, MSJ_ERROR_APERTURA_ARCHIVO);
213                 fclose(FSALIDA);
214                 return ST_ERROR_APERTURA_ARCHIVO;
215             }
216         }
217     }
218     else if(argv[ARG_POS_FSALIDA2]==NULL && argv[ARG_POS_FSALIDA1]!=NULL){
219         fprintf(stderr, "%s: %s\n", MSJ_ERROR, MSJ_ERROR_APERTURA_ARCHIVO);
220         return ST_ERROR_APERTURA_ARCHIVO;
221     } se repiten los mismos errores que en la apertura de los archivo de entrada.
222
223     return ST_OK;
224 } NUNCA PIDEN MEMORIA PARA EL VECTOR QUE ESTÁ ADENTRO DE LA ESTRUCTURA ESTADO.
225
226 status_t validar_ayuda(int argc, char *argv[])

```

```

227  /*recibe arc y argv para verificar si el usuario ejecuto el programa con el argumento de ayuda
228  para que fuese impresa la información correspondiente*/
229  {
230
231      if(!argv){
232          return ST_ERROR_PTR_NULO;
233      }
234      if(argc!=ARGC2_MAX){
235          return ST_ERROR_CANT_ARG;
236      }
237      if(argv[ARG_POS_H]!=NULL){
238          imprimir_ayuda();
239      }
240
241      return ST_OK;
242  }
243
244  status_t imprimir_ayuda()
245  /*Imprime la información de ayuda: tabla del orden de los argumentos y
246  tabla de las operaciones validas*/
247  {
248
249      printf("%s\n", MSJ_ACLARACION_AYUDA );
250
251      printf("%s      %s      %s\n",MSJ_TITULO_ARG, MSJ_TITULO_OPC, MSJ_TITULO_DESC);
252      printf("%s      %s      %s\n",MSJ_AYUDA_ARG, MSJ_AYUDA_OPC, MSJ_AYUDA_DESC);
253      printf("%s      %s      %s\n",MSJ_MEMORIA_ARG, MSJ_MEMORIA_OPC, MSJ_MEMORIA_DESC);
254      printf("%s      %s      %s\n",MSJ_ENTRADA_ARG, MSJ_ENTRADA_OPC, MSJ_ENTRADA_DESC);
255      printf("%s      %s      %s\n",MSJ_ARCHIVO_ARG, MSJ_ARCHIVO_OPC_BIN, MSJ_ARCHIVO_DESC_BIN);
256      printf("      %s      %s\n",MSJ_ARCHIVO_OPC_TXT,MSJ_ARCHIVO_DESC_TXT);
257      printf("%s      %s      %s\n",MSJ_SALIDA_ARG, MSJ_SALIDA_OPC, MSJ_SALIDA_DESC);
258      printf("%s      %s      %s\n",MSJ_SALIDA_ARCH_ARG, MSJ_SALIDA_ARCH_BIN_OPC, MSJ_SALIDA_ARCH_BIN_DE
259      printf("      %s      %s      %s\n",MSJ_SALIDA_ARCH_TXT_OPC, MSJ_SALIDA_ARCH_TXT_DESC);
260
261      printf("%s      %s      %s\n",TAB_TITULO_OP,TAB_TITULO_CODE,TAB_TITULO_DESC);
262      printf("%s\n",TAB_ENT_SAL);
263      printf("%s      %s      %s\n",TAB_LEER_OP,TAB_LEER_CODE,TAB_LEER_DESC);
264      printf("%s      %s      %s\n",TAB_ESCRIBIR_OP,TAB_ESCRIBIR_CODE,TAB_ESCRIBIR_DESC);
265      printf("%s\n",TAB_MOV);
266      printf("%s      %s      %s\n",TAB_CARGAR_OP,TAB_CARGAR_CODE,TAB_CARGAR_DESC);
267      printf("%s      %s      %s\n",TAB_GUARDAR_OP,TAB_GUARDAR_CODE,TAB_GUARDAR_DESC);
268      printf("%s      %s      %s\n",TAB_PCARGAR_OP,TAB_PCARGAR_CODE,TAB_PCARGAR_DESC);
269      printf("%s      %s      %s\n",TAB_PGUARDAR_OP,TAB_PGUARDAR_CODE,TAB_PGUARDAR_DESC);
270      printf("%s\n",TAB_MATE);
271      printf("%s      %s      %s\n",TAB_SUMAR_OP,TAB_SUMAR_CODE,TAB_SUMAR_DESC);
272      printf("%s      %s      %s\n",TAB_RESTAR_OP,TAB_RESTAR_CODE, TAB_RESTAR_DESC);
273      printf("%s      %s      %s\n",TAB_DIVIDIR_OP,TAB_DIVIDIR_CODE,TAB_DIVIDIR_DESC);
274      printf("%s      %s      %s\n",TAB_MULT_OP,TAB_MULT_CODE,TAB_MULT_DESC);
275      printf("%s\n",TAB_CONTROL);
276      printf("%s      %s      %s\n",TAB_JUMP_OP, TAB_JUMP_CODE,TAB_JUMP_DESC);
277      printf("%s      %s      %s\n",TAB_JMPNEG_OP, TAB_JMPNEG_CODE,TAB_JMPNEG_DESC);
278      printf("%s      %s      %s\n",TAB_JMPZERO_OP, TAB_JMPZERO_CODE,TAB_JMPZERO_DESC);
279      printf("%s      %s      %s\n",TAB_JNZ_OP,TAB_JNZ_CODE,TAB_JNZ_DESC);
280      printf("%s      %s      %s\n",TAB_DJNZ_OP,TAB_DJNZ_CODE,TAB_DJNZ_DESC);
281      printf("%s      %s      %s\n",TAB_FIN_OP,TAB_FIN_CODE,TAB_FIN_DESC);
282
283      return ST_OK;
284  }

```

esto se hace en la función anterior, salvo por la impresión de la ayuda, que se haría en la función invocante.

se puede poner todo en una única línea:

```

#define TXT_AYUDA \
    "primera línea" \
    "segunda línea" \
    etc.

```


¿por qué recibe tantos datos? Requiere del estado, el formato de salida y el nombre del archivo (o un FILE * abierto)

285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342

```
status_t seleccionar_salida(char *argv[],parametros_t *params, estado_t *estado, FILE *FSALIDA)
/*recibe argv para verificar cual es el formato del archivo de salida o si se hara por stdout;
además recibe el puntero a la estructura de argumentos, de estado y al archivo de salida para poder
a las funciones que se encargaran de imprimir en el formato correspondiente.*/
{
    if(strcmp(argv[ARG_POS_FENTRADA2],OPCION_TXT)){
        imprimir_archivo_txt(params, estado, FSALIDA);
    }
    else if (strcmp(argv[ARG_POS_FENTRADA2],OPCION_BIN)){
        imprimir_archivo_bin(params, estado, FSALIDA);
    }
    else
        imprimir_pantalla(params,estado);

    return ST_OK;
}

imprimir_pantalla(params, estado) debería ser equivalente a
imprimir_archivo_txt(params, estado, stdout) (no se requiere OTRA función).
status_t imprimir_pantalla(parametros_t *params, estado_t * estado)
/*Recibe los punteros a la estructura de argumentos y a la de estado para imprimir los datos guarda
en el acumulador, en el contador del programa, la ultima instruccion ejecutada,
el ultimo opcode y operando, además de la memoria de todas las palabras, en forma de matriz*/
{
    int i,k,l;

    printf("%s\n", MSJ_REGISTRO);
    printf("%25s: %6i\n",MSJ_ACUM, estado->acumulador );
    printf("%25s: %6i\n",MSJ_CONT_PROG, estado->contador_programa );
    printf("%25s: %6i\n",MSJ_INST, estado->palabras[estado->contador_programa] );
    estado->opcode = *(estado->palabras + estado->contador_programa) /100;/*divido por 100 entonces
estado->operando = estado->palabras[estado->contador_programa] - (estado->opcode*100);/*necesito
printf("%25s: %6i\n",MSJ_OPCODE, estado->opcode );
printf("%25s: %6i\n",MSJ_OPERANDO, estado->operando );
k=0;
printf(" ");
for (l = 0; l < 10; l++)
    printf(" %i ",l) ;
for ( i = 0; i < params->cant_palabras ; i++){
    if ((i%10)==0){
        printf("\n%02i ",k); con imprimir i es suficiente. Si i%10 es 0, entonces i
        k+=10; es 0 o 10 o 20 o 30 o 40 o etc.
    }
    if(estado->palabras[i]<0)
        printf("%05i ",estado->palabras[i] );
    else
        printf("+%04i ",estado->palabras[i] );
}
printf("\n");
return ST_OK;
}

status_t imprimir_archivo_txt(parametros_t *params, estado_t *estado, FILE *FSALIDA)
/*Recibe el puntero del archivo de salida, los punteros a la estructura de argumentos y
a la de estado para imprimir los datos guardados en el acumulador, en el contador del programa,
```

printf("%+05i...

↑ obliga a la impresión del signo, sea positivo o negativo.

```

343 la ultima instruccion ejecutada, el ultimo opcode y operando,
344 además de la memoria de todas las palabras, en forma de matriz*/
345 {
346     int i,k,l;
347     fprintf(FSALIDA,"%s\n", MSJ_REGISTRO);
348     fprintf(FSALIDA, "%25s: %6d\n",MSJ_ACUM, estado->acumulador );
349     fprintf(FSALIDA, "%25s: %6d\n",MSJ_CONT_PROG, estado->contador_programa );
350     fprintf(FSALIDA, "%25s: %6d\n",MSJ_INST, estado->palabras[estado->contador_programa] );
351     estado->opcode = estado->palabras[estado->contador_programa] /100; /*divido por 100 entonces como
352     estado->operando = estado->palabras[estado->contador_programa] - (estado->opcode*100); /*necesito
353     fprintf(FSALIDA, "%25s: %6d\n",MSJ_OPCODE, estado->opcode );
354     fprintf(FSALIDA, "%25s: %6d\n",MSJ_OPERANDO, estado->operando );
355     k=0;
356     fprintf(FSALIDA,"    ");
357     for (l = 0; l < 10; l++)
358         fprintf(FSALIDA," %i ",l) ;
359     for ( i = 0; i < params->cant_palabras ; i++){
360         if ((i%10)==0){
361             fprintf(FSALIDA,"\n%02i ",k); idem
362             k+=10;
363         }
364         if(estado->palabras[i]<0)
365             fprintf(FSALIDA,"%05i ",estado->palabras[i] );
366         else idem
367             fprintf(FSALIDA,"+%04i ",estado->palabras[i] );
368         }
369     fprintf(FSALIDA,"\n");
370     return ST_OK;
371 }
372
373 status_t imprimir_archivo_bin (parametros_t *params, estado_t *estado, FILE *FSALIDA)
374 /*Recibe el puntero del archivo de salida, los punteros a la estructura de argumentos y
375 a la de estado para imprimir los datos guardados en el acumulador, en el contador del programa,
376 la ultima instruccion ejecutada, el ultimo opcode y operando, además de la memoria de todas las pal
377 {
378
379     fwrite(&estado->acumulador, sizeof(estado_t),1, FSALIDA);
380     fwrite(&estado->contador_programa, sizeof(estado_t),1, FSALIDA );
381     fwrite(&estado->palabras[estado->contador_programa], sizeof(estado_t),1, FSALIDA);
382     estado->opcode = estado->palabras[estado->contador_programa] /100; /*divido por 100 entonces como
383     estado->operando = estado->palabras[estado->contador_programa] - (estado->opcode*100); /*necesito
384     fwrite(&estado->opcode, sizeof(estado_t),1, FSALIDA);
385     fwrite(&estado->operando, sizeof(estado_t),1, FSALIDA);
386     fwrite(&estado->palabras[params->cant_palabras], sizeof(int), params->cant_palabras, FSALIDA);
387     return ST_OK;
388 }
389
390 La estructura recibida debería ser íntegramente en memoria dinámica, aunque no es
391 status_t liberar_memoria(estado_t * estado) obligatorio.
392 /*Recibe el puntero a la estructura de estado para liberar la memoria pedida*/
393 {
394     free(estado->palabras);
395     estado->palabras=NULL;
396     return ST_OK;
397 }
398
399 status_t operaciones (estado_t * estado)
400 /*Recibe el puntero a la estructura de estado para hacer un análisis de las instrucciones que se enc

```

```
401 en el vector palabras, y después se llama a una función que realiza la operación necesaria.*/
402 {
403     int salir;
404     if (estado->palabras[estado->contador_programa]<0){
405         estado->contador_programa++; ¿eh? Si se intenta ejecutar una posición negativa
406         return ST_OK_NEG;           debería ser un error.
407     }
408     else{
409         estado->opcode = estado->palabras[estado->contador_programa] /100; /*divido por 100 entonces
410         estado->operando = estado->palabras[estado->contador_programa] - (estado->opcode*100); /*nece
411         switch (estado->opcode){
412             case (LEER):
413                 op_leer(estado);
414                 estado->contador_programa++;
415                 break;
416             case (ESCRIBIR):
417                 op_escribir(estado);
418                 estado->contador_programa++;
419                 break;
420             case (CARGAR):
421                 op_cargar(estado);
422                 estado->contador_programa++;
423                 break;
424             case (GUARDAR):
425                 op_guardar(estado);
426                 estado->contador_programa++;
427                 break;
428             case (PCARGAR):
429                 op_pcargar(estado);
430                 estado->contador_programa++;
431                 break;
432             case (PGUARDAR):
433                 op_pguardar(estado);
434                 estado->contador_programa++;
435                 break;
436             case (SUMAR):
437                 op_sumar(estado);
438                 estado->contador_programa++;
439                 break;
440             case (RESTAR):
441                 op_restar(estado);
442                 estado->contador_programa++;
443                 break;
444             case (DIVIDIR):
445                 op_dividir(estado);
446                 estado->contador_programa++;
447                 break;
448             case (MULTIPLICAR):
449                 op_multiplicar(estado);
450                 estado->contador_programa++;
451                 break;
452             case (JMP):
453                 op_jmp(estado);
454                 break;
455             case (JMPNEG):
456                 if(estado->acumulador<0)
457                     op_jmp(estado);
458                 else
```

```

459         estado->contador_programa++;
460         break;
461     case(JMPZERO):
462         if(estado->acumulador==0)
463             op_jmp(estado);
464         else
465             estado->contador_programa++;
466         break;
467     case(JNZ):
468         if(estado->acumulador!=0)
469             op_jmp(estado);
470         else
471             estado->contador_programa++;
472         break;
473     case(DJNZ):
474         op_djnz(estado);
475         break;
476     case(HALT):
477         salir = -1;
478         break;
479     default:
480         break;
481 }
482 }
483 if (salir == -1){
484     return ST_SALIR;
485 }
486 else{
487     return ST_OK;
488 }
489 }
490
491 status_t op_leer (estado_t * estado)
492 /*Lee una palabra por stdin a una posicion de memoria que está indicada por el operando (miembro de
493 {
494     int * AUX; minúsculas...
495     AUX=NULL;
496     printf("%s\n", MSJ_INGRESO_PALABRA);
497     if (fgets((char*)AUX,MAX_LARGO_PALABRA,stdin)==NULL)
498         fprintf(stderr, "%s\n", MSJ_ERROR_PALABRA_NULA );
499     return ST_ERROR_PALABRA_VACIA; /*la palabra ingresada es nula*/
500     estado->palabras[estado->operando] = *AUX;
501     return ST_OK;
502 }
503
504 status_t op_escribir(estado_t * estado)
505 /*imprime por stdout el contenido de la posicion del operando(miembro de la estructura estado)*/
506 {
507     fprintf(stdout, "%s %i : %i\n", MSJ_IMPRIMIR_PALABRA,estado->operando, estado->palabras[estado->
508     return ST_OK;
509 }
510
511 status_t op_cargar (estado_t * estado)
512 /*Carga en el acumulador (miembro de la estructura estado) la posicion de memoria indicada
513 por el operando(miembro de la estructura estado)*/
514 {
515     estado->acumulador = estado->palabras[estado->operando];
516     return ST_OK;

```

¿por qué no hacer algo que itere mientras el opcode sea distinto de HALT?

No funciona correctamente. Se debe leer la cadena, procesarla, convertirla, validarla, etc.

¿qué ocurre si el operando está fuera de rango?

```
517 }
518
519
520 status_t op_pcargar (estado_t * estado)
521 /*Carga en el acumulador (miembro de la estructura estado) la posicion de memoria indicada
522 por la palabra a la que apunta el operando(miembro de la estructura estado)*/
523 {
524     estado->acumulador = estado->palabras[estado->palabras[estado->operando]];
525     return ST_OK;
526 }
527
528 status_t op_guardar (estado_t * estado)
529 /*guarda en la posicion de memoria indicada por el operando(miembro de la estructura estado)
530 lo que está en el acumulador(miembro de la estructura estado)*/{
531     estado->palabras[estado->operando] = estado->acumulador ;
532     return ST_OK;
533 }
534
535 status_t op_pguardar (estado_t * estado)
536 /*guarda en la posicion de memoria indicada por la palabra a la que apunta el operando (miembro de
537 lo que esta en el acumulador(miembro de la estructura estado)*/
538 {
539     estado->palabras[estado->palabras[estado->operando]] = estado->acumulador ;
540     return ST_OK;
541 }
542
543
544 status_t op_restar (estado_t * estado)
545 /*resta al acumulador (miembro de la estructura estado) lo guardado en la posicion de memoria indca
546 por el operando(miembro de la estructura estado)*/
547 {
548     estado->acumulador -= estado->palabras[estado->operando];
549     return ST_OK;
550 }
551
552 status_t op_dividir (estado_t * estado)
553 /*divide al acumulador (miembro de la estructura estado) por lo guardado en la posicion de memoria
554 por el operando(miembro de la estructura estado)*/{
555     estado->acumulador /= estado->palabras[estado->operando];
556     return ST_OK;
557 }
558
559 status_t op_multiplicar (estado_t * estado)
560 /*multiplica al acumulador (miembro de la estructura estado)lo guardado en la posicion de memoria i
561 {
562     estado->acumulador *= estado->palabras[estado->operando];
563     return ST_OK;
564 }
565
566 status_t op_sumar(estado_t * estado)
567 /*suma al acumulador (miembro de la estructura estado) lo guardado en la posicion de memoria indcad
568 por el operando(miembro de la estructura estado)*/
569 {
570     estado->acumulador += estado->palabras[estado->operando];
571     return ST_OK;
572 }
573
574 status_t op_jump (estado_t * estado)
```

acá es obligatorio validar que el índice se encuentre dentro del rango admitido.

```

575 /*salta a la posicion de memoria indicada por el operando(miembro de la estructura estado) menos un
576 {
577     estado->contador_programa = estado->operando;
578     return ST_OK;
579 }
580
581 status_t op_djnz (estado_t * estado)
582 /*decrementa en 1 el acumulador (miembro de la estructura estado) y salta a la posicion indicada
583 por el operando (miembro de la estructura estado) en el caso que el acumulador sea distinto de 0*/
584 {
585     estado->acumulador--;
586     if (estado->acumulador!=0){
587         estado->contador_programa = estado->operando;
588     }
589     else
590         estado->contador_programa++;
591     return ST_OK;
592 }

```

7.2. Archivos .h

estructuras y prototipos

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  #include "estructuras_prototipos.h"
6
7  #define LANG_ENGLISH /*elección del idioma del programa*/
8
9  #ifndef LANG_SPANISH
10 #include "LANG_SPANISH.h"
11 #endif
12
13 #ifndef LANG_ENGLISH
14 #include "LANG_ENGLISH.h"
15 #endif
16
17 int main(int argc, char *argv[])
18 {
19     estado_t simpletron;
20     estado_t *estado;
21
22     parametros_t argumentos;
23     parametros_t *params;
24     status_t st;
25     FILE *FENTRADA, *FSALIDA;
26     params=NULL;
27     estado=NULL;
28     FENTRADA=NULL;
29     FSALIDA=NULL;
30
31     estado=&simpletron;
32     params=&argumentos;
33
34
35     if(argc==ARGO2_MAX){
36         if((st=validar_ayuda(argc, argv))!=ST_OK){

```

Volvieron a cargar main.c

```
1: #include <stdio.h>
2:
3:
4: #ifndef LANG_ENGLISH__H
5: #define lang_english__H 1
6:
7: #define ARGV_MAX 6
8: #define ARGV2_MAX 2 ARGV_MIN
9: #define CANT_PALABRAS_DEFAULT 50
10: #define ARG_POS_CANT_PALABRAS 1
11: #define ARG_POS_FENTRADA1 2 ARG_POS_FENTRADA_NOMBRE
12: #define ARG_POS_FENTRADA2 3 ARG_POS_FENTRADA_TIPO
13: #define ARG_POS_FSALIDA1 4
14: #define ARG_POS_FSALIDA2 5
15: #define ARG_POS_H 1
16: #define OPCION_TXT "txt"
17: #define OPCION_BIN "bin"
18: #define OPCION_STDIN "stdin"
19: #define OPCION_STDOUT "stdout"
20:
21: #define MSJ_ERROR "ERROR"
22: #define MSJ_ERROR_APERTURA_ARCHIVO "opening file"
23: #define MSJ_ERROR_PTR_NULO "null pointer"
24: #define MSJ_ERROR_CANT_ARG "number of arguments"
25: #define MSJ_ERROR_CANT_PALABRAS "number of words"
26: #define MSJ_ERROR_NO_MEM "out of memory"
27: #define MSJ_ERROR_PALABRA_NULA "the entered word is null"
28:
29:
30: #define LEER 10
31: #define ESCRIBIR 11
32: #define CARGAR 20
33: #define GUARDAR 21
34: #define PCARGAR 22
35: #define PGUARDAR 23
36: #define SUMAR 30
37: #define RESTAR 31
38: #define DIVIDIR 32
39: #define MULTIPLICAR 33
40: #define JMP 40
41: #define JMPNEG 41
42: #define JMPZERO 42
43: #define JNZ 43
44: #define DJNZ 44
45: #define HALT 45
46: #define MSJ_IMPRIMIR_PALABRA "Content of the position"
47: #define MAX_LARGO_PALABRA 5 /*maximum number of characters in a word*/
48: #define MSJ_INGRESO_PALABRA "Enter a word:"
49:
```

Estas 2 etiquetas son distintas, por lo que la protección contra inclusiones múltiples no sirve

Esto no es parte del idioma. No va en ningún archivo que defina idiomas. En los archivos de idiomas van las traducciones de los mensajes, va lo que cambia de un idioma a otro.

typedef enum {...} opcode_t;

va en simpletron.h

```

50: #define MSJ_ACUM "accumulator"
51: #define MSJ_CONT_PROG "program counter"
52: #define MSJ_INST "instruction"
53: #define MSJ_OPCODE "opcode"
54: #define MSJ_OPERANDO "operand"
55:
56:
57: #define MSJ_BIENVENIDA "Welcome to Simpletron! Please enter your program one instruction (or data word) at a time. I will
11 type the location number and a question mark (?). You then type the word for that location. Type the sentinel -99999 to stop
p entering your program."
58: #define FIN -99999
59: #define MIN_PALABRA -9999
60: #define MAX_PALABRA 9999
61: #define MSJ_CARGA_COMPLETA "*** Program upload complete ***"
62: #define MSJ_COMIENZO_EJECUCION "*** Program execution starts ***"
63:
64: #define MSJ_ARG_POSICIONALES "The arguments are positional"
65:
66: #define MSJ_TITULO_ARG "Arg."
67: #define MSJ_TITULO_OPC "Option"
68: #define MSJ_TITULO_DESC "Description"
69:
70: #define MSJ_AYUDA_ARG "-h"
71: #define MSJ_AYUDA_OPC "doesn't have"
72: #define MSJ_AYUDA_DESC "Shows help."
73:
74: #define MSJ_MEMORIA_ARG "-m"
75: #define MSJ_MEMORIA_OPC "N"
76: #define MSJ_MEMORIA_DESC "Simpletron has a memory for N words. If no arguments is provided, by default it'll have 50 words."
77:
78: #define MSJ_ENTRADA_ARG "-i"
79: #define MSJ_ENTRADA_OPC "file"
80: #define MSJ_ENTRADA_DESC "The program will be read from the file provided, otherwise, from stdin."
81:
82: #define MSJ_ARCHIVO_ARG "-ia"
83: #define MSJ_ARCHIVO_OPC_BIN "bin"
84: #define MSJ_ARCHIVO_DESC_BIN "The input file will be understood as a binary sequence of integers that represent the words
ds that make up the program."
85: #define MSJ_ARCHIVO_OPC_TXT "txt"
86: #define MSJ_ARCHIVO_DESC_TXT "The input file will be interpreted as a sequence of numbers, each one in a single line"
87:
88: #define MSJ_SALIDA_ARG "-o"
89: #define MSJ_SALIDA_OPC "archivo"
90: #define MSJ_SALIDA_DESC "The dump will be done in the file provided, otherwise, the dump will be done by stdout."
91:
92: #define MSJ_SALIDA_ARCH_ARG "-of"
93:
94: #define MSJ_SALIDA_ARCH_BIN_OPC "bin"

```



```
95: #define MSJ_SALIDA_ARCH_BIN_DESC "The dump will be done in binary saving every element of the Simpletron structure, in ↗
addition to the memory."
96: #define MSJ_SALIDA_ARCH_TXT_OPC "txt"
97: #define MSJ_SALIDA_ARCH_TXT_DESC "The dump will be made in text format by printing the registers and the memory."
98:
99: #define MSJ_ACLARACION_AYUDA "To enter these arguments must be placed in order. If you want to leave the field empty yo↗
u must put '-'."
100:
101: #define TAB_TITULO_OP "Operation"
102: #define TAB_TITULO_CODE "OpCode"
103: #define TAB_TITULO_DESC "Description"
104:
105: #define TAB_ENT_SAL "Input/Output Op.:"
106: #define TAB_LEER_OP "READ"
107: #define TAB_LEER_CODE "10"
108: #define TAB_LEER_DESC "Reads a word from stdin to a position of memory"
109:
110: #define TAB_ESCRIBIR_OP "WRITE"
111: #define TAB_ESCRIBIR_CODE "11"
112: #define TAB_ESCRIBIR_DESC "Prints by stdout a position of memory"
113:
114: #define TAB_MOV "Movement Op.:"
115: #define TAB_CARGAR_OP "LOAD"
116: #define TAB_CARGAR_CODE "20"
117: #define TAB_CARGAR_DESC "Loads a word from the memory to the accumulator"
118:
119: #define TAB_GUARDAR_OP "SAVE"
120: #define TAB_GUARDAR_CODE "21"
121: #define TAB_GUARDAR_DESC "Saves a word from the accumulator to the memory"
122:
123: #define TAB_PCARGAR_OP "PLOAD"
124: #define TAB_PCARGAR_CODE "22"
125: #define TAB_PCARGAR_DESC "Same as LOAD but the operand is pointer"
126:
127: #define TAB_PGUARDAR_OP "PSAVE"
128: #define TAB_PGUARDAR_CODE "23"
129: #define TAB_PGUARDAR_DESC "Same a SAVE but the operand is pointer"
130:
131: #define TAB_MATE "Arithmetic Op.:"
132: #define TAB_SUMAR_OP "ADDITION"
133: #define TAB_SUMAR_CODE "30"
134: #define TAB_SUMAR_DESC "Adds a word to the accumulator"
135:
136: #define TAB_RESTAR_OP "SUBTRACTION"
137: #define TAB_RESTAR_CODE "31"
138: #define TAB_RESTAR_DESC "Subtract a word to the accumulator"
139:
140: #define TAB_DIVIDIR_OP "DIVIION"
141: #define TAB_DIVIDIR_CODE "32"
```

```
142: #define TAB_DIVIDIR_DESC "Divides the accumulator by the operand"
143:
144: #define TAB_MULT_OP "MULTIPLY"
145: #define TAB_MULT_CODE "33"
146: #define TAB_MULT_DESC "Multiplies the accumulator by the operand"
147:
148: #define TAB_CONTROL "Control Op.:"
149: #define TAB_JUMP_OP "JMP"
150: #define TAB_JUMP_CODE "40"
151: #define TAB_JUMP_DESC "Jumps to a memory location"
152:
153: #define TAB_JMPNEG_OP "JMPNEG"
154: #define TAB_JMPNEG_CODE "41"
155: #define TAB_JMPNEG_DESC "Idem only if the accumulator is negative"
156:
157: #define TAB_JMPZERO_OP "JMPZERO"
158: #define TAB_JMPZERO_CODE "42"
159: #define TAB_JMPZERO_DESC "Idem only if the accumulator is zero"
160:
161: #define TAB_JNZ_OP "JNZ"
162: #define TAB_JNZ_CODE "43"
163: #define TAB_JNZ_DESC "Idem only if the accumulator is NOT zero"
164:
165: #define TAB_DJNZ_OP "DJNZ"
166: #define TAB_DJNZ_CODE "44"
167: #define TAB_DJNZ_DESC "Decreases the accumulator and jump if it is NOT zero"
168:
169: #define TAB_FIN_OP "HALT"
170: #define TAB_FIN_CODE "45"
171: #define TAB_FIN_DESC "Finishes the program"
172:
173: #define MSJ_REGISTRO "Registers:"
174:
175: #endif
```

```
1: #include <stdio.h>
2:
3:
4: #ifndef LANG_SPANISH__H
5: #define lang_spanish__H 1
6:
7: #define ARGV_MAX 6
8: #define ARGV2_MAX 2
9: #define CANT_PALABRAS_DEFAULT 50
10: #define ARG_POS_CANT_PALABRAS 1
11: #define ARG_POS_FENTRADA1 2
12: #define ARG_POS_FENTRADA2 3
13: #define ARG_POS_FSALIDA1 4
14: #define ARG_POS_FSALIDA2 5
15: #define ARG_POS_H 1
16: #define OPCION_TXT "txt"
17: #define OPCION_BIN "bin"
18: #define OPCION_STDIN "stdin"
19: #define OPCION_STDOUT "stdout"
20:
21:
22: #define MSJ_ERROR "ERROR"
23: #define MSJ_ERROR_APERTURA_ARCHIVO "apertura de archivo"
24: #define MSJ_ERROR_PTR_NULO "puntero nulo"
25: #define MSJ_ERROR_CANT_ARG "cantidad de argumentos"
26: #define MSJ_ERROR_CANT_PALABRAS "cantidad de palabras"
27: #define MSJ_ERROR_NO_MEM "no hay memoria"
28: #define MSJ_ERROR_PALABRA_NULA "la palabra ingresada es nula"
29:
30:
31:
32: #define LEER 10
33: #define ESCRIBIR 11
34: #define CARGAR 20
35: #define GUARDAR 21
36: #define PCARGAR 22
37: #define PGUARDAR 23
38: #define SUMAR 30
39: #define RESTAR 31
40: #define DIVIDIR 32
41: #define MULTIPLICAR 33
42: #define JMP 40
43: #define JMPNEG 41
44: #define JMPZERO 42
45: #define JNZ 43
46: #define DJNZ 44
47: #define HALT 45
48: #define MSJ_IMPRIMIR_PALABRA "Contenido de la posicion"
49: #define MAX_LARGO_PALABRA 5 /*maxima cantidad caracteres en una palabra*/
```

idemn anterior. La capitalización es distinta.

idem, no es idioma.

va en simpletron.h

```
50: #define MSJ_INGRESO_PALABRA "Ingrese una palabra:"
51:
52: #define MSJ_ACUM "acumulador"
53: #define MSJ_CONT_PROG "contador del programa"
54: #define MSJ_INST "instruccion"
55: #define MSJ_OPCODE "opcode"
56: #define MSJ_OPERANDO "operando"
57:
58:
59: #define MSJ_BIENVENIDA "Bienvenide a le Simpletron! Por favor, ingrese su programa una instruccion (o dato) a la vez.
Yo escribir la ubicacion y un signo de pregunta (?) .Luego usted ingrese la palabra para esa ubicacion. Ingrese -99999 para
finalizar."
60: #define FIN -99999
61: #define MIN_PALABRA -9999
62: #define MAX_PALABRA 9999
63: #define MSJ_CARGA_COMPLETA "*** Carga del programa completa ***"
64: #define MSJ_COMIENZO_EJECUCION "*** Comienza la ejecucion del programa ***"
65:
66:
67: #define MSJ_ARG_POSICIONALES "Los argumentos son posicionales. En caso de querer que entrada sea por stdin se debe escribir 'entrada stdin' y en caso de querer que salida sea por stdout 'salida stdout'."
68:
69: #define MSJ_TITULO_ARG "Arg."
70: #define MSJ_TITULO_OPC "Opcion"
71: #define MSJ_TITULO_DESC "Descripcion"
72:
73: #define MSJ_AYUDA_ARG "-h"
74: #define MSJ_AYUDA_OPC "no posee"
75: #define MSJ_AYUDA_DESC "Muestra una ayuda."
76:
77: #define MSJ_MEMORIA_ARG "-m"
78: #define MSJ_MEMORIA_OPC "N"
79: #define MSJ_MEMORIA_DESC "Simpletron tiene una memoria de N palabras. Si no se da el argumento, por omision tendra; 50 palabras."
80:
81: #define MSJ_ENTRADA_ARG "-i"
82: #define MSJ_ENTRADA_OPC "archivo"
83: #define MSJ_ENTRADA_DESC "El programa se leer; del archivo pasado como opcion, en caso contrario, de stdin."
84:
85: #define MSJ_ARCHIVO_ARG "-ia"
86:
87: #define MSJ_ARCHIVO_OPC_BIN "bin"
88: #define MSJ_ARCHIVO_DESC_BIN "El archivo de entrada se entender; como una secuencia binaria de enteros que representan las palabras que forman el programa."
89:
90: #define MSJ_ARCHIVO_OPC_TXT "txt"
91: #define MSJ_ARCHIVO_DESC_TXT "El archivo de entrada se interpretar; como secuencia de numeros, cada uno en una linea."
92:
```

```
93: #define MSJ_SALIDA_ARG "-o"
94: #define MSJ_SALIDA_OPC "archivo"
95: #define MSJ_SALIDA_DESC "El dump se har ; en el archivo pasado como opci n, si no pasa el argumento, el volcado se har ;
 ; por stdout."
96:
97: #define MSJ_SALIDA_ARCH_ARG "-of"
98:
99: #define MSJ_SALIDA_ARCH_BIN_OPC "bin"
100: #define MSJ_SALIDA_ARCH_BIN_DESC "El volcado se har ; en binario guardando cada elemento de la estructura del Simpletro ;
n, adem s de la memoria."
101:
102: #define MSJ_SALIDA_ARCH_TXT_OPC "txt"
103: #define MSJ_SALIDA_ARCH_TXT_DESC "El volcado se har ; en formato de texto imprimiendo los registros y la memoria."
104:
105: #define MSJ_ACLARACION_AYUDA "Para ingresar estos argumentos se deber ; colocar en el orden mostrado a continuaci n.  
En caso de querer dejar el campo vacio se deber ; colocar '-' . "
106:
107: #define TAB_TITULO_OP "Operaci n"
108: #define TAB_TITULO_CODE "OpCode"
109: #define TAB_TITULO_DESC "Descripci n"
110:
111: #define TAB_ENT_SAL "Op. de Entrada/Salida:"
112: #define TAB_LEER_OP "LEER"
113: #define TAB_LEER_CODE "10"
114: #define TAB_LEER_DESC "Lee una palabra de stdin a una posici n de memoria"
115:
116: #define TAB_ESCRIBIR_OP "ESCRIBIR"
117: #define TAB_ESCRIBIR_CODE "11"
118: #define TAB_ESCRIBIR_DESC "Imprime por stdout una posici n de memoria"
119:
120: #define TAB_MOV "Op. de movimiento:"
121: #define TAB_CARGAR_OP "CARGAR"
122: #define TAB_CARGAR_CODE "20"
123: #define TAB_CARGAR_DESC "Carga una palabra de la memoria en el acumulador"
124:
125: #define TAB_GUARDAR_OP "GUARDAR"
126: #define TAB_GUARDAR_CODE "21"
127: #define TAB_GUARDAR_DESC "Guarda una palabra del acumulador a la memoria"
128:
129: #define TAB_PCARGAR_OP "PCARGAR"
130: #define TAB_PCARGAR_CODE "22"
131: #define TAB_PCARGAR_DESC "Idem CARGAR pero el operando es puntero"
132:
133: #define TAB_PGUARDAR_OP "PGUARDAR"
134: #define TAB_PGUARDAR_CODE "23"
135: #define TAB_PGUARDAR_DESC "Idem GUARDAR pero el operando es puntero"
136:
137: #define TAB_MATE "Op. aritm ticas:"
138: #define TAB_SUMAR_OP "SUMAR"
```

```
139: #define TAB_SUMAR_CODE "30"
140: #define TAB_SUMAR_DESC "Suma una palabra al acumulador"
141:
142: #define TAB_RESTAR_OP "RESTAR"
143: #define TAB_RESTAR_CODE "31"
144: #define TAB_RESTAR_DESC "Resta una palabra al acumulador"
145:
146: #define TAB_DIVIDIR_OP "DIVIDIR"
147: #define TAB_DIVIDIR_CODE "32"
148: #define TAB_DIVIDIR_DESC "Divide el acumulador por el operando"
149:
150: #define TAB_MULT_OP "MULTIPLICAR"
151: #define TAB_MULT_CODE "33"
152: #define TAB_MULT_DESC "Multiplica el acumulador por el operando"
153:
154: #define TAB_CONTROL "Op. control:"
155: #define TAB_JUMP_OP "JMP"
156: #define TAB_JUMP_CODE "40"
157: #define TAB_JUMP_DESC "Salta a una ubicaci3n de memoria"
158:
159: #define TAB_JMPNEG_OP "JMPNEG"
160: #define TAB_JMPNEG_CODE "41"
161: #define TAB_JMPNEG_DESC "Idem s3lo si el acumulador es negativo"
162:
163: #define TAB_JMPZERO_OP "JMPZERO"
164: #define TAB_JMPZERO_CODE "42"
165: #define TAB_JMPZERO_DESC "Idem s3lo si el acumulador es cero"
166:
167: #define TAB_JNZ_OP "JNZ"
168: #define TAB_JNZ_CODE "43"
169: #define TAB_JNZ_DESC "Idem s3lo si el acumulador NO es cero"
170:
171: #define TAB_DJNZ_OP "DJNZ"
172: #define TAB_DJNZ_CODE "44"
173: #define TAB_DJNZ_DESC "Decrementa el acumulador y salta si NO es cero"
174:
175: #define TAB_FIN_OP "HALT"
176: #define TAB_FIN_CODE "45"
177: #define TAB_FIN_DESC "Finaliza el programa"
178:
179:
180: #define MSJ_REGISTRO "Registros:"
181:
182: #endif
```

```

1: #include <stdio.h>
2:
3:
4: #ifndef ESTRUCTURAS_PROTOTIPOS__H
5: #define estructuras_prototipos__H 1
6:
7: typedef struct estado
8: {
9:     int acumulador; /*posicion de memoria del acumulador*/
10:    int contador_programa; /*cuenta el numero de paso y de memoria que se encuentra el programa*/
11:    int *palabras; /*vector donde estan guardadas las palabras*/
12:    int opcode; /*el codigo de operacion, que especifica la operaci3n a realizar*/
13:    int operando; /*represeta la direcci3n de memoria que contiene la palabra a la que se le aplica la operaci3n*/
14:
15: }estado_t; /*una estructura para almacenar el estado del Simpletron*/
16:
17: typedef struct parametros
18: {
19:     int cant_palabras; /*la cantidad de palabras que han sido asignadas en memoria para las instrucciones*/
20:     char i; /*argumento que indica que el programa se leera del archivo pasado como opcion*/
21:     char ia; /*argumento que indica que el archivo de entrada se leera con el formato especificado como opcion*/
22:     char o; /*argumento que indica que el dump se hara en el archivo pasado como opcion*/
23:     char of; /*argumento que indica que el dump se hara en el formato especificado como opcion*/
24: }parametros_t; /*estructura con los argumentos que son pasados al momento de ejecucion*/
25:
26: typedef enum
27: {
28:     ST_OK,
29:     ST_ERROR_PTR_NULO,
30:     ST_ERROR_CANT_ARG,
31:     ST_ERROR_FUERA_RANGO,
32:     ST_OK_NEG,
33:     ST_ERROR_PALABRA_VACIA,
34:     ST_ERROR_APERTURA_ARCHIVO,
35:     ST_ERROR_CANT_PALABRAS,
36:     ST_ERROR_NO_NUMERICO,
37:     ST_SALIR
38: }status_t;
39:
40:
41: status_t validar_ayuda(int argc, char *argv[]);
42: status_t imprimir_ayuda();
43: status_t validar_argumentos (int argc , char *argv[], parametros_t *params, estado_t * estado, FILE * FENTRADA, FILE *F
SALIDA);
44:
45: status_t leer_archivo_txt(parametros_t *params, estado_t * estado,FILE *FENTRADA, FILE *FSALIDA);
46: status_t leer_archivo_bin(parametros_t *params, estado_t * estado, FILE *FENTRADA, FILE *FSALIDA);
47: status_t leer_teclado(parametros_t *params, estado_t * estado);
48:

```

las capitalizaciones son distintas.

esto va en simpletron.h

const char * nombre...;

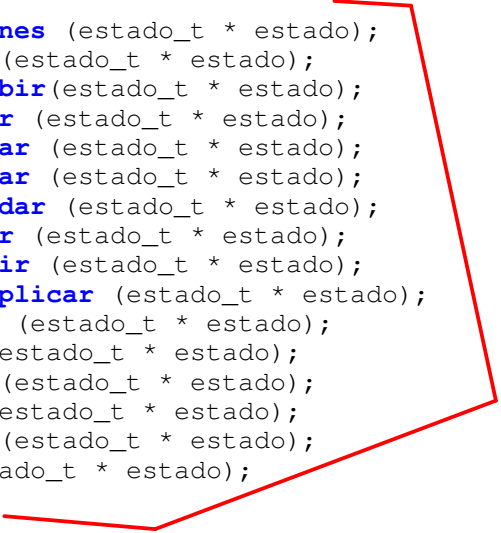
...

/* en la fx. de validaci3n */

params->nombre_... = argv[ARG_POS_....]

status.h, error.h, estado.h, etc.

```
49: status_t seleccionar_salida(char *argv[],parametros_t *params, estado_t *estado, FILE *FSALIDA);
50: status_t imprimir_pantalla(parametros_t *params, estado_t * estado);
51: status_t imprimir_archivo_bin (parametros_t *params, estado_t * estado, FILE *FSALIDA);
52: status_t imprimir_archivo_txt (parametros_t *params, estado_t * estado, FILE *FSALIDA);
53: status_t liberar_memoria(estado_t *estado);
54:
55: status_t operaciones (estado_t * estado);
56: status_t op_leer (estado_t * estado);
57: status_t op_escribir(estado_t * estado);
58: status_t op_cargar (estado_t * estado);
59: status_t op_pcargar (estado_t * estado);
60: status_t op_guardar (estado_t * estado);
61: status_t op_pguardar (estado_t * estado);
62: status_t op_restar (estado_t * estado);
63: status_t op_dividir (estado_t * estado);
64: status_t op_multiplicar (estado_t * estado);
65: status_t op_sumar (estado_t * estado);
66: status_t op_jump (estado_t * estado);
67: status_t op_djnz (estado_t * estado);
68: status_t jmpneg (estado_t * estado);
69: status_t jmpzero (estado_t * estado);
70: status_t jnz (estado_t * estado);
71:
72: #endif
```



simpletron.h/simpletron.c