The Simpletron[*]

Machine-Language Programming

Algoritmos y Programación I (95.11/75.02)

7 de mayo de 2018

1. Objetivo

El objetivo del presente trabajo es el diseño y desarrollo de una aplicación que permita interpretar y ejecutar un lenguaje de máquina inventado. A este intérprete lo llamaremos *Simpletron*. Utilizando el lenguaje inventando y el *Simpletron* se puede escribir cualquier programa.

La aplicación debe ser escriba en lenguaje ANSI-C89.

Nota: El trabajo práctico está basado en un ejercicio del capítulo de punteros del libro de Deitel & Deitel, mencionado en la bibliografía de la materia. Se recomienda leer dicho ejercicio.

2. Alcance

Mediante el presente TP se busca que el/la estudiante aplique, conocimientos sobre los siguientes temas:

- Programas en modo consola
- Directivas al preprocesador C
- Juego de caracteres ASCII
- Tipos enumerativos
- Control de flujo
- Salida de datos con formato
- Funciones
- Cadenas de caracteres
- Arreglos/Vectores
- Memoria dinámica
- Argumentos en línea de órdenes (CLA)
- Estructuras
- Archivos
- Modularización

Fecha de entrega: 22 de mayo de 2018

[*]basado en un ejercicio del libro *C How to program* de Deitel & Deitel

3. Introducción

Se ha de crear una "computadora" a la cual llamaremos *Simpletron*[1]. La *Simpletron* ejecuta programas escritos en un lenguaje diseñado específicamente para este problema, el *Lenguaje de Máquina de la* Simpletron, LMS. También llamado el *assembly* de la *Simpletron*. Este lenguaje y este ejercicio brindarán un mayor entendimiento de lo que ocurre en los lenguajes de programación más cercanos al hardware.

4. Desarrollo

4.1. *Simpletron*

Toda la información que utiliza la *Simpletron* se maneja en *palabras*. Una palabra es un número entero de cuatro dígitos *con signo*, como son `+9511`, `-7502`, `+0666`, `-2330`, etcétera.

Le *Simpletron* tiene espacio para almacenar una cantidad fija y finita de palabras, la *memoria*. En memoria 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, números de cuatro dígitos. Además, deben ser siempre positivos. Los primeros dos dígitos de cada instrucción represetan el código de operación, *opcode*, el cual especifica la operación a realizar. En la tabla 1 se resumen los códigos de operación que debe soportar el programa. Los últimos dos dígitos de la instrucción son el *operando*, el cual represeta la dirección de memoria que contiene la palabra a la que se le aplica la operación.

Operación	*OpCode*	Descripción
Op. de Entrada/Salida:		
`LEER`	10	Lee una palabra de *stdin* a una posición de memoria
`ESCRIBIR`	11	Imprime por *stdout* una posición de memoria
Op. de movimiento:		
`CARGAR`	20	Carga una palabra de la memoria en el acumulador
`GUARDAR`	21	Guarda una palabra del acumulador a la memoria
`PCARGAR`	22	Idem `CARGAR` pero el operando es *puntero*
`PGUARDAR`	23	Idem `GUARDAR` pero el operando es *puntero*
Op. aritméticas:		
`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
Op. control:		
`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

[1] dado que es la computadora, o el computador, o el sistema, o la máquina, etc. la *Simpletron* se llamará indistintamente la o el o le *Simpletron*

Operación	*OpCode*	Descripción
`JNZ`	43	Idem sólo si el acumulador NO es cero
`DJNZ`	44	Decrementa el acumulador y salta si NO es cero
`HALT`	45	Finaliza el programa

Tabla 1: Códigos de operación—*opcodes*—del LMS

Además, la *Simpletron* posee un *acumulador*, `ACC`, 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`.

4.2. Ejemplos de código LMS

suma.lms

```
0  +1009 ; LEER en la posición de memoria 09
1  +1010 ; LEER en la posición de memoria 10
2  +2009 ; CARGAR la posición de memoria 09 al acumulador
3  +3010 ; SUMAR al acumulador la posición de memoria 10
4  +2111 ; GUARDAR el acumulador en la posición de memoria 11
5  +1111 ; ESCRIBIR por pantalla el contenido de la memoria 11
6  +4500 ; HALT
7  +0000 ; nop
8  +0000 ; nop
9  +0000 ; Variable A
10 +0000 ; Variable B
11 +0000 ; Resultado
```

resta.lms

```
0  +1009
1  +1010
2  +2009
3  +3110
4  +4107 ; JMPNEG salta a la pos. 07 si el acumulador es negativo
5  +1109
6  +4008 ; JMP salta a la posición de memoria 08
7  +1110
8  +4500
9  +0000
10 +0000
11 +0000
```

maximo.lms

```
0  +1017
1  +2018
2  +3117
3  +4215
4  +2018
5  +3021
```

```
+2118
+1019
+2020
+3119
+4112
+4001
+2019
+2120
+4001
+1120
+4500
+0000
+0000
+0000
+0000
+0001
```

promedio.lms

```
+1022 ; LEER en la posición de memoria 22
+2022 ; CARGAR la posición de memoria 22 al acumulador
+3119 ; RESTAR al acumulador la posición de memoria 19
+4212 ; JMPZERO salta a la posición 12 si el acumulador es cero
+2019 ; CARGAR la posición de memoria 19 al acumulador
+3020 ; SUMAR el contenido de la pos. de memoria 20 al acumulador
+2119 ; GUARDAR el contenido del acumulador en la posición 19
+1018 ; LEER en la posición de memoria 18
+2017 ; CARGAR la posición de memoria 17
+3018 ; SUMAR la posición de memoria 18
+2117 ; GUARDAR el contenido del acumulador en la posición 17
+4001 ; JMP salta a la posición 01
+2017 ; CARGAR la posición de memoria 17
+3222 ; DIVIDIR el acumulador por el contenido de 22
+2121 ; GUARDAR el acumulador en la posición 21
+1121 ; ESCRIBIR por pantalla el contenido de 21
+4500 ; HALT
+0000 ; Pos. 17 (valor acumulado)
+0000 ; Pos. 18 (variable)
+0000 ; Pos. 19 (contador)
+0001 ; Pos. 20 (incrementos de a uno)
+0000 ; Pos. 21
+0000 ; Pos. 22 (cantidad)
```

5. Simulador del *Simpletron*

La presente sección se basa en el ejercicio 7.27 del Deitel & Deitel [1]

En base a la sección anterior, definiremos las especificaciones de un simulador del *Simpletron*. Con este simulador, podrán ejecutar el código escrito en LMS.

Siguiendo el enunciado del Deitel, el programa debe iniciar con la impresión de uno de los siguientes mensajes, según haya sido compilado para utilizar el español o el inglés.

```
*** ¡Bienvenide a le Simpletron!              ***
*** Por favor, ingrese su programa una        ***
```

```
*** instrucción (o dato) a la vez. Yo          ***
*** escribiré la ubicación y un signo de       ***
*** pregunta (?). Luego usted ingrese la       ***
*** palabra para esa ubicación. Ingrese        ***
*** -99999 para finalizar:                     ***
```

Ejemplo 1: Mensaje de inicio en español

```
*** Welcome to Simpletron!                     ***
*** Please enter your program one instruction ***
*** (or data word) at a time. I will type the ***
*** location number and a question mark (?).  ***
*** You then type the word for that location. ***
*** Type the sentinel -99999 to stop entering ***
*** your program.                              ***
```

Ejemplo 2: Mensaje de inicio en inglés

Ahora suponga que ya hemos pasado esa etapa y el simulador está corriendo, e ingresamos el código de resta.lms. Veremos algo similar a los siguiente:

```
00 ? +1009
01 ? +1010
02 ? +2009
03 ? +3110
04 ? +4107
05 ? +1109
06 ? +4008
07 ? +1110
08 ? +4500
09 ? +0000
10 ? +0000
11 ? +0000
12 ? -99999
*** Carga del programa completa                ***
*** Comienza la ejecución del programa         ***
```

Ejemplo 3: Ingreso de resta.lms con mensajes en español

El programa se encuentra cargado en la *memoria* del *Simpletron* y será ejecutado.

Como mencionamos antes, la *Simpletron* posee un *acumulador* por donde pasan **todos** los datos. La *Simpletron* deberá iterar por las distintas posiciones de memoria, por lo que será necesario contar con una variable que nos permita llevar la cuenta de dicha posición, *program counter*. Con la variable antes mencionada podemos obtener el *registro de instrucción*, que equivale a la palabra almacenada en la memoria indicada por el contador antes mencionado. De este *registro de instrucción* podemos obtener tanto el código de operación, *opcode*, como el *operando*.

Finalizada la ejecución del programa, se hará un volcado de la aplicación, mostrando todos sus parámetros y el estado de la memoria, como se muestra en el ejemplo 4. Este volcado se conoce como *dump*, por su traducción del idioma inglés.

```
********INICIO DE EJECUCION DEL SIMPLETRON*******
Ingrese una palabra: 125
Ingrese una palabra: 3695
Contenido de la posicion 10: 3695
*********FIN DE EJECUCION DEL SIMPLETRON*********

REGISTROS:
     acumulador: -3570
program counter:    08
    instruccion: +4500
         opcode:    45
       operando:    00

MEMORIA:
     0     1     2     3     4     5     6     7     8     9
 0 +1009 +1010 +2009 +3110 +4107 +1109 +4008 +1110 +4500 +0125
10 +3695 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
20 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
30 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
40 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
50 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
60 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
70 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
80 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
```

Ejemplo 4: Ejecución y *dump* final en español

5.1. Implementación

A diferencia del libro, usaremos una estructura para almacenar el estado del *Simpletron*. La misma debe contener todas las variables antes mencionadas y, además, la memoria. Como ayuda, pueden consultar el libro en este aspecto.

Además, es requisito separar la funcionalidad del *Simpletron* de la del simulador. En este caso, se recomienda consultar a los docentes sobre las ideas para llevar a cabo la modularización.

Se preferirá un programa con alta cohesión en los módulos y bajo acoplamiento entre los mismos.

5.2. Configuración

Nuestro simulador, a diferencia del propuesto en el libro, 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 2 se detallan los argumentos que recibirá el programa. Se admite que los argumentos sean posicionales, aunque en dicho caso será necesario que todos los argumentos estén presentes, incluso en el caso en que sólo se quiera configurar el formato del archivo de salida.

Opcional: considere que el carácter ; indica el comienzo de un comentario que termina al finalizar la línea. Además, todos los espacios en blanco al final

Arg.	Opción	Descripción
`-h`	no posee	Muestra una ayuda
`-m`	`N`	*Simpletron* 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 *stdin*, como se muestra en el ejemplo 3.
`-if`	`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, como se muestra en los ejemplos de la sección 4.2.
`-o`	`archivo`	El *dump* se hará en el archivo pasado como opción, si no pasa el argumento, el volcado se hará por *stdout*.
`-of`	`bin`	El volcado se hará en binario guardando cada elemento de la estructura del *Simpletron*, además de la memoria.
	`txt`	El volcado se hará en formato de texto, como se muestra en el ejemplo 4, imprimiendo los registros y la memoria.

Tabla 2: Tabla de argumentos del programa principal

de código útil son descartados, lo mismo que los comentarios. De esta forma, las líneas

```
+4415 ; Decrementa el acumulador y, si el mismo no es 0,
      ; vuelve al comienzo del ciclo
```

son válidas, y equivalentes a

```
+4415
```

5.3. Ejecución del programa

Suponiendo que el programa se llama `simpletron`, algunas variantes de ejecución son:

```
$ # Argumentos posicionales:
$ ./simpletron -h
$ cat ej.bin | ./simpletron -m 25 -i - -if bin -o dump.bin -of bin
$ ./simpletron -m 25 -i ej.bin -if bin -o dump.bin -of bin
$ ./simpletron -m 40 -i ej2.lms -if txt -o dump.txt -of txt
$ # Argumentos no posicionales:
$ ./simpletron -h
$ cat ej.bin | ./simpletron -m 25 -if bin -o dump.bin -of bin
$ ./simpletron -m 25 -i ej.bin -if bin
$ ./simpletron -m 40 -i ej2.lms -if txt -of txt
```

IMPORTANTE: Tenga en cuenta que si el programa se pasa por *stdin* utilizando `cat`, el flujo de entrada se cierra al finalizar el archivo. Si el programa LMS interactúa con el usuario y pretende leer una palabra de *stdin*, no podrá leer nunca y entrará en un bucle infinito. Si se desea realizar una prueba de este

modo el archivo debe contener, luego del programa, los parámetros que se desea pasar durante la ejecución.

6. Testing

Para probar la correcta ejecución del programa se deben ejecutar los códigos LMS que se encuentran en este enunciado. Algunos de los programas se encuentran documentados, otros no: documentarlos.

6.1. Fugas de memoria

Las fugas de memoria de la aplicación pueden ser advertidas utilizando la aplicación *valgrind*. Para que la misma indique dónde se producen fugas de memoria, es necesario compilar nuestra aplicación en modo debug.

Una vez compilada la aplicación, *valgrind* se ejecuta de la siguiente manera:

```
valgrind ./mi_aplicacion
valgrind ./mi_aplicacion argumento1 argumento2 ...
```

Al ejecutar correctamente el comando **y** si el programa no tiene fallas, se ve una leyenda similar a la siguiente:

```
==23390== HEAP SUMMARY:
==23390==     in use at exit: 0 bytes in 0 blocks
==23390==   total heap usage: 89,408 allocs, 89,408 frees, 5,057,256
    bytes allocated
==23390==
==23390== All heap blocks were freed -- no leaks are possible
==23390==
==23390== For counts of detected and suppressed errors, rerun with:
   -v
==23390== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0
   from 0)
```

Ejemplo 5: Resultado de ejecución de valgrind sin fugas de memoria

Si en cambio el programa tiene fugas de memoria, el mensaje es:

```
==25854== HEAP SUMMARY:
==25854==     in use at exit: 456,722 bytes in 3,932 blocks
==25854==   total heap usage: 89,408 allocs, 85,476 frees, 5,057,256
    bytes allocated
==25854==
==25854== LEAK SUMMARY:
==25854==    definitely lost: 456,722 bytes in 3,932 blocks
==25854==    indirectly lost: 0 bytes in 0 blocks
==25854==      possibly lost: 0 bytes in 0 blocks
==25854==    still reachable: 0 bytes in 0 blocks
==25854==         suppressed: 0 bytes in 0 blocks
==25854== Rerun with --leak-check=full to see details of leaked
   memory
==25854==
==25854== For counts of detected and suppressed errors, rerun with:
   -v
```

```
==25854== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0
    from 0)
```

Ejemplo 6: Resultado de ejecución de valgrind con fugas de memoria

7. Restricciones

La realización de los programas pedidos está sujeta a las siguientes restricciones:

- Debe realizarse en grupos de **3 (tres)** integrantes.
- No está permitida la utilización de `scanf()`, `gets()`[2], `fflush(stdin)`[3],la biblioteca `conio.h`[4] (`#include <conio.h>`), etc.
- Debe recurrirse a la utilización de funciones mediante una adecuada parametrización.
- No está permitido en absoluto tener hard-codings:

```
...
int opcode;
...
if (opcode == 40)                /* ¡¡hard-coded!!*/
    printf("%s","xxxxxxxx");     /* ¡¡hard-coded!!*/
else if(opcode == 41)            /* ¡¡hard-coded!!*/
    printf("%s","yyyyyyyy");     /* ¡¡hard-coded!!*/
...
```

  sino que debe recurrirse al uso de `ETIQUETAS`, `CONSTANTES SIMBÓLICAS`, `MACROS`, etc.

  Los ejemplos no son exhaustivos, sino que existen otros hard-codings y tampoco son aceptados.

- Hay ciertas cuestiones que no han sido especificadas intencionalmente en este Requerimiento, para darle al/la desarrollador/a la libertad de elegir implementaciones que, según su criterio, resulten más convenientes en determinadas situaciones. Por lo tanto, se debe explicitar cada una de las decisiones adoptadas, y el o los fundamentos considerados para las mismas.

8. Entrega

La fecha límite de aprobación del trabajo práctico es el 5 de junio de 2018.

No se requiere entrega en papel, pero se acepta. Deberá realizarse una entrega digital, a través del campus de la materia, de un único archivo cuyo nombre debe seguir el siguiente formato:

[2] obsoleta en C99 [3], eliminada en C11 [4] por fallas de seguridad en su uso.
[3] comportamiento indefinido para flujos de entrada ([3],[4]). Definida en estándar POSIX.
[4] biblioteca no estándar, con diferentes implementaciones y licencias, y no siempre disponible.

```
YYYYMMDD_apellido1-apellido2-apellido3-N.tar.gz
```

donde `YYYY` es el año (2018), `MM` el mes y `DD` el día en que uno de los integrantes sube el archivo, `apellido-1a3` son los apellidos de los integrantes ordenados alfabéticamente, `N` indica el número de vez que se envía el trabajo (`1`, `2`, etc.), y `.tar.gz` es la extensión, que no necesariamente es `.tar.gz`.

El archivo comprimido debe contener los siguientes elementos:

- La correspondiente documentación de desarrollo del TP (en formato pdf), siguiendo la numeración siguiente, incluyendo:
  1. Carátula del TP. Incluir una dirección de correo electrónico.
  2. Enunciado del TP.
  3. Estructura funcional de los programas desarrollados.
  4. Explicación de cada una de las alternativas consideradas y las estrategias adoptadas.
  5. Resultados de la ejecución (corridas) de los programas, captura de las pantallas, bajo condiciones normales e inesperadas de entrada.
  6. **Archivos de prueba utilizados.**
  7. Reseña sobre los problemas encontrados en el desarrollo de los programas y las soluciones implementadas para subsanarlos.
  8. Bibliografía (ver sección 9).
  9. Indicaciones sobre la compilación de lo entregado para generar la aplicación.
     NOTA: Si la compilación del código fuente presenta mensajes de aviso (warning), notas o errores, los mismos deben ser comentados en un apartado del informe.

  NOTA: El Informe deberá ser redactado en *correcto* idioma castellano.
- Códigos fuentes en formato de texto plano (.c y .h), *debidamente documentados*.

NOTA: Se debe generar y subir un único archivo (comprimido) con todos los elementos de la entrega digital. **NO usar RAR.** La compresión RAR no es un formato libre, en tanto sí se puede utilizar *ZIP*, *GUNZIP*, u otros (soportados, por ejemplo, por la aplicación de archivo *TAR*).

Si no se presenta cada uno de estos ítems, será rechazado el TP.

9. Bibliografía

Debe incluirse la referencia a toda bibliografía consultada para la realización del presente TP: libros, artículos, URLs, etc., citando:

- Denominación completa del material (Título, Autores, Edición, Volumen, etc.).

- Código ISBN del libro (opcional: código interbibliotecario).
- URL del sitio consultado. No poner Wikipedia.org o stackexchange.com, sino que debe incluírse un enlace al artículo, hilo, etc. consultado.

Utilizando LaTeX, la inclusión de citas/referencias es trivial. Los editores de texto gráficos de las suites de ofimática, como LibreOffice Write o MS Word, admiten plugins que facilitan la inclusión.

Ejemplo de referencias

[1] P. Deitel y H. Deitel. *C How to Program*. 7.ª ed. Pearson Education, 2012. ISBN: 9780133061567.

[2] B.W. Kernighan y D.M. Ritchie. *The C Programming Language*. 2.ª ed. Prentice-Hall software series. Prentice Hall, 1988. ISBN: 9780131103627.

[3] ISO/IEC. *Programming Languages – C*. ISO/IEC 9899:1999(E). ANSI, dic. de 1999, págs. 270-271.

[4] ISO/IEC. *Programming Languages – C*. INCITS/ISO/IEC 9899:2011. INCITS/ISO/IEC, 2012, pág. 305.