Práctica 1: Máquina Virtual

Fecha de entrega: 14 de Noviembre de 2016, 9:00

Objetivo: Iniciación a la orientación a objetos y a Java; uso de arrays y enumerados; manipulación de cadenas con la clase `String`; entrada y salida por consola.

Introducción

Una máquina virtual es un software que simula a un ordenador y que puede ejecutar programas como si fuese un ordenador real. Puedes encontrar más información en `http://es.wikipedia.org/wiki/Maquina_virtual`. El ordenador simulado puede ejecutar programas como si se tratase de un ordenador real. Este tipo de aplicaciones tiene diferentes usos aunque el más extendido es la "prueba" de sistemas operativos sin tener que cambiar el que utilizan habitualmente. La máquina virtual que pretendemos implementar a lo largo del curso es una máquina de pila, cuyos principales componentes son una *pila de operandos* y una *memoria*. La pila de operandos almacena los datos que se generan durante la ejecución mientras que en la memoria podemos almacenar algunos de esos datos. Es decir, la memoria sería similar a una tabla de variables de un programa.

A lo largo del primer cuatrimestre realizaremos distintas prácticas encaminadas a finalizar con una aplicación completa de máquina virtual (a la que llamaremos TPMV), capaz de ejecutar programas creados para esta máquina, y cuya sintaxis iremos definiendo a lo largo del primer cuatrimestre. La aplicación final permitirá escribir un programa en un lenguaje imperativo simple, compilar dicho programa al lenguaje admitido por la TPMVy mostrar el *estado* de la máquina después de la ejecución. El objetivo de esta primera práctica es construir los primeros bloques lógicos necesarios.

Descripción de la práctica

En esta primera práctica vamos a crear una versión inicial muy sencilla de nuestra máquina virtual. En concreto la aplicación será capaz de ejecutar una serie de comandos introducidos por el usuario para crear, modificar y ejecutar programas, donde un programa es una lista de instancias de las instrucciones que se presentan en la siguiente sección.

Al arrancar la aplicación, se iniciarán todas las estructuras de la TPMV y se presentará un *prompt* en donde el usuario irá tecleando los distintos comandos que se quieren ejecutar. El conjunto de comandos disponibles serán:

- HELP: Que muestra información sobre los distintos comandos disponibles.
- QUIT: Cierra la aplicación.
- NEWINST BC: Introduce la instrucción bytecode BC al programa actual. Si BC no está correctamente escrito, entonces manda un mensaje de error y no lleva a cabo la insercción.
- RUN: Ejecuta el programa actual. En caso de que se produzca un error de ejecución, avisa al usuario mediante un mensaje.
- RESET: Inicializa el programa actual eliminando todas las instrucciones almacenadas.
- REPLACE N: Solicita al usuario una nueva instrucción BC, que en caso de ser correcta reemplazará a la instrucción bytecode número N del programa.

Tras la ejecución de cada comando, la aplicación mostrará el programa actual almacenado. Para el caso del comando RUN, después de la ejecución de cada una de las instrucciones del programa se mostrará además los estados en los que ha quedado la máquina (el formato utilizado para mostrar esta información aparece en los ejemplos de ejecución de la sección 4).

Es posible que la ejecución de algún comando falle. En ese caso la aplicación simplemente mostrará un mensaje indicando que no ha sido posible la ejecución y pedirá un nuevo comando. La aplicación terminará cuando se ejecute el comando QUIT.

Descripción de TPMV en la práctica 1

TPMV está compuesta de dos partes muy simples:

- Una *memoria* capaz de almacenar datos. La unidad mínima de memoria es el entero, es decir en cada celda almacena un entero completo (y no un *byte* como suele ocurrir en las máquinas reales). La capacidad de la memoria es *ilimitada*, es decir se podrá escribir en cualquier dirección (≥ 0), hasta que la memoria de la máquina física subyacente "aguante".
- Una *pila de operandos* en la que se realizan las operaciones. Gran parte de las distintas instrucciones bytecode de la máquina virtual trabajan sobre la pila de operandos, cogiendo de ella valores y/o dejando en ella resultados.

En esta primera práctica la TPMV tiene un conjunto reducido de instrucciones que no la permitirán ejecutar programas sofisticados. La mayoría de ellas *no* tienen parámetros (pues trabajan directamente con la pila de operandos). Sólo unas pocas de ellas tienen un parámetro de tipo entero. El conjunto de instrucciones bytecode admitidas es:

- PUSH n: apila en la pila de operandos el entero `n`.
- LOAD pos: lee de la memoria el valor almacenado en `pos` y lo apila en la pila de operandos.

- STORE pos: escribe en la posición `pos` de la memoria el contenido de la cima de la pila de operandos, y lo elimina de ella.
- ADD, SUB, MUL, DIV: operaciones aritméticas de suma, resta, multiplicación y división. Todas ellas utilizan como operandos la subcima y la cima de la pila. Tanto la cima como la subcima son sustituidas por el resultado de la operación. El primer operando es *la subcima* (esto es importante para las operaciones no conmutativas).
- OUT: escribe el entero almacenado en la cima de la pila.
- HALT: para la máquina.

Ejemplo de ejecución

A continuación mostramos un ejemplo de ejecución de nuestra simulación. Observa que los comandos e instrucciones bytecode *no* son sensibles a mayúsculas ni minúsculas (puedes escribirlas indistintamente). Eso sí, cuando la aplicación tiene que mostrar una instrucción o un bytecode, lo hará utilizando mayúsculas.

En el momento de mostrar el estado de la máquina, la aplicación muestra la pila y la memoria tras la ejecución de cada bytecode. Si no tienen elementos, se indica `<vacia>`. En el caso de la memoria eso viene a significar que alguna celda de memoria ha sido escrita en algún momento. Sólo se mostrará el contenido de las posiciones de memoria que están ocupadas.

A continuación aparece un ejemplo de ejecución. El texto en negrita representa lo que el usuario de la aplicación introduce por teclado.

```
> help
Comienza la ejecución de HELP

  HELP: Muestra esta ayuda
  QUIT: Cierra la aplicacion
  RUN: Ejecuta el programa
  NEWINST BYTECODE: Introduce una nueva instrucción al programa
  RESET: Vacia el programa actual
  REPLACE N: Reemplaza la instruccion N por la solicitada al usuario


> newinst push 2
Comienza la ejecución de NEWINST PUSH 2
Programa almacenado:
0: PUSH 2

> newinst push 3
Comienza la ejecución de NEWINST PUSH 3
Programa almacenado:
0: PUSH 2
1: PUSH 3

> run
Comienza la ejecución de RUN
El estado de la maquina tras ejecutar el bytecode PUSH 2 es:

Estado de la CPU:
  Memoria: <vacia>
```

```
  Pila: 2

El estado de la maquina tras ejecutar el bytecode PUSH 3 es:

Estado de la CPU:
  Memoria: <vacia>
  Pila: 2  3

Programa almacenado:
0: PUSH 2
1: PUSH 3

> newinst add
Comienza la ejecución de NEWINST ADD
Programa almacenado:
0: PUSH 2
1: PUSH 3
2: ADD

> newinst store 4
Comienza la ejecución de NEWINST STORE 4
Programa almacenado:
0: PUSH 2
1: PUSH 3
2: ADD
3: STORE 4

> run
Comienza la ejecución de RUN
El estado de la maquina tras ejecutar el bytecode PUSH 2 es:

Estado de la CPU:
  Memoria: <vacia>
  Pila: 2

El estado de la maquina tras ejecutar el bytecode PUSH 3 es:

Estado de la CPU:
  Memoria: <vacia>
  Pila: 2  3

El estado de la maquina tras ejecutar el bytecode ADD es:

Estado de la CPU:
  Memoria: <vacia>
  Pila: 5

El estado de la maquina tras ejecutar el bytecode STORE 4 es:

Estado de la CPU:
  Memoria:  [4]:5
  Pila: <vacia>

Programa almacenado:
0: PUSH 2
1: PUSH 3
```

```
2: ADD
3: STORE 4

> replace 2
Comienza la ejecución de REPLACE
Nueva instrucción: sub
Programa almacenado:
0: PUSH 2
1: PUSH 3
2: SUB
3: STORE 4

> run
Comienza la ejecución de RUN
El estado de la maquina tras ejecutar el bytecode PUSH 2 es:

Estado de la CPU:
  Memoria: <vacia>
  Pila: 2

El estado de la maquina tras ejecutar el bytecode PUSH 3 es:

Estado de la CPU:
  Memoria: <vacia>
  Pila: 2  3

El estado de la maquina tras ejecutar el bytecode SUB es:

Estado de la CPU:
  Memoria: <vacia>
  Pila: -1

El estado de la maquina tras ejecutar el bytecode STORE 4 es:

Estado de la CPU:
  Memoria:  [4]:-1
  Pila: <vacia>

Programa almacenado:
0: PUSH 2
1: PUSH 3
2: SUB
3: STORE 4

> reSet
Comienza la ejecución de RESET

> newinst push 30
Comienza la ejecución de NEWINST PUSH 30
Programa almacenado:
0: PUSH 30

> newinst push 22
Comienza la ejecución de NEWINST PUSH 22
Programa almacenado:
0: PUSH 30
```

```
1: PUSH 22

> quit
Comienza la ejecución de QUIT
Programa almacenado:
0: PUSH 30
1: PUSH 22

Fin de la ejecucion....
```

A continuación aparece un pequeño ejemplo de ejecución donde alguno de los comandos o bytecodes fallan. Como se ve, el mensaje de error es único.

```
> newinst push
Comienza la ejecución de NEWINST
Error: Ejecucion incorrecta del comando

> newinst push 2
Comienza la ejecución de NEWINST PUSH 2
Programa almacenado:
0: PUSH 2

> newinst add
Comienza la ejecución de NEWINST ADD
Programa almacenado:
0: PUSH 2
1: ADD

> run
Comienza la ejecución de RUN
El estado de la maquina tras ejecutar el bytecode PUSH 2 es:

Estado de la CPU:
  Memoria: <vacia>
  Pila: 2

Error: Ejecucion incorrecta del comando
Programa almacenado:
0: PUSH 2
1: ADD

> replace 4
Comienza la ejecución de REPLACE
Error: Ejecucion incorrecta del comando
Programa almacenado:
0: PUSH 2
1: ADD

> replace 1
Comienza la ejecución de REPLACE
Nueva instrucción: push 0
Programa almacenado:
0: PUSH 2
1: PUSH 0

> newinstr div
```

```
Error: Comando incorrecto
Programa almacenado:
0: PUSH 2
1: PUSH 0

> newinst div
Comienza la ejecución de NEWINST DIV
Programa almacenado:
0: PUSH 2
1: PUSH 0
2: DIV

> run
Comienza la ejecución de RUN
El estado de la maquina tras ejecutar el bytecode PUSH 2 es:

Estado de la CPU:
  Memoria: <vacia>
  Pila: 2

El estado de la maquina tras ejecutar el bytecode PUSH 0 es:

Estado de la CPU:
  Memoria: <vacia>
  Pila: 2  0

Error: Ejecucion incorrecta del comando
Programa almacenado:
0: PUSH 2
1: PUSH 0
2: DIV

>
```

A continuación presentamos un último ejemplo de ejecución donde se muestra el uso de la memoria de la máquina:

```
> newinst push 22
Comienza la ejecución de NEWINST PUSH 22
Programa almacenado:
0: PUSH 22

> newinst push 33
Comienza la ejecución de NEWINST PUSH 33
Programa almacenado:
0: PUSH 22
1: PUSH 33

> newinst store 5
Comienza la ejecución de NEWINST STORE 5
Programa almacenado:
0: PUSH 22
1: PUSH 33
2: STORE 5

> run
```

```
Comienza la ejecución de RUN
El estado de la maquina tras ejecutar el bytecode PUSH 22 es:

Estado de la CPU:
  Memoria: <vacia>
  Pila: 22

El estado de la maquina tras ejecutar el bytecode PUSH 33 es:

Estado de la CPU:
  Memoria: <vacia>
  Pila: 22  33

El estado de la maquina tras ejecutar el bytecode STORE 5 es:

Estado de la CPU:
  Memoria:  [5]:33
  Pila: 22

Programa almacenado:
0: PUSH 22
1: PUSH 33
2: STORE 5

> newinst load 5
Comienza la ejecución de NEWINST LOAD 5
Programa almacenado:
0: PUSH 22
1: PUSH 33
2: STORE 5
3: LOAD 5

> newinst store 10
Comienza la ejecución de NEWINST STORE 10
Programa almacenado:
0: PUSH 22
1: PUSH 33
2: STORE 5
3: LOAD 5
4: STORE 10

> run
Comienza la ejecución de RUN
El estado de la maquina tras ejecutar el bytecode PUSH 22 es:

Estado de la CPU:
  Memoria: <vacia>
  Pila: 22

El estado de la maquina tras ejecutar el bytecode PUSH 33 es:

Estado de la CPU:
  Memoria: <vacia>
  Pila: 22  33

El estado de la maquina tras ejecutar el bytecode STORE 5 es:
```

```
Estado de la CPU:
  Memoria:  [5]:33
  Pila: 22

El estado de la maquina tras ejecutar el bytecode LOAD 5 es:

Estado de la CPU:
  Memoria:  [5]:33
  Pila: 22  33

El estado de la maquina tras ejecutar el bytecode STORE 10 es:

Estado de la CPU:
  Memoria:  [5]:33 [10]:33
  Pila: 22

Programa almacenado:
0: PUSH 22
1: PUSH 33
2: STORE 5
3: LOAD 5
4: STORE 10

>
```

Implementación

Para implementar la práctica necesitarás, al menos, las siguientes clases:

- Engine: Representa el bucle de control de la aplicación. Contiene un método public void start() que se encarga de leer sucesivamente los comandos introducidos por el usuario, hasta recibir el comando QUIT. Cada comando introducido se parsea convenientemente, utilizando el método public static Command parse(String line) de la clase CommandParser. Después se lleva a cabo la ejecución del comando invocando al método public boolean execute(Engine engine) de la clase Command. Si se produce un error en la ejecución del comando debe mostrarse un mensaje tal y como aparece en la Sección 4. Al final de la ejecución de cualquier comando se debe mostrar el programa actual. Los atributos de esta clase son al menos private ByteCodeProgram program y private boolean end que representan, respectivamente, el programa actual y la terminación de la aplicación. Además esta clase contendrá todos los métodos públicos necesarios para que puedan ejecutarse los distintos comandos.
- Command: Representa los distintos comandos que puede utilizar un usuario. Para representar un comando necesitaremos los atributos private ENUM_COMANDO command, private ByteCode instruction y private int replace. Lógicamente el atributo instruction toma valor cuando se hace referencia al comando NEWINST, mientras que el atributo replace hace referencia al comando REPLACE. El tipo enumerado ENUM_COMMAND contiene una constante de enumeración por cada uno de los comandos. Además esta clase contiene el método public boolean execute(Engine engine), que es el encargado de ejecutar el comando, dando la orden correspodiente a engine. En caso de que la ejecución del comando sea incorrecta, el método devuelve false. En otro caso devuelve true.

- CommandParser: Contiene como único método public static Command parse(String line), que se encarga de analizar el parámetro line y generar a partir de él el correspondiente comando. Si line no se corresponde con la sintáxis de ningún comando entonces el método devuelve null.

- ByteCodeProgram: Representa el programa actual. Al menos contiene el atributo privado private ByteCode[] program, así como métodos públicos para añadir un bytecode al programa, inicializar el programa, colocar una instrucción bytecode en una posición dada del programa o devolver la instrucción i-ésima del programa, etc..

- Memory: Representa la memoria de la maquina. Recuerda que la memoria de la máquina es ilimitada, y se puede escribir en cualquier dirección mayor o igual que 0. Esto significa que cuando se almacena o se lee en cualquier dirección no se generará ningún error de ejecución a menos que sea provocado por la memoria de la máquina física externa, en cuyo caso no haremos nada. Para poder visualizar la memoria por pantalla necesitamos saber qué posiciones de memoria han sido utilizadas. Se considera que una posición de memoria ha sido utilizada si se ha escrito o leído en ella, es decir se ha ejecutado alguna instrucción bytecode STORE o LOAD sobre esa posición. Al menos esta clase necesitará un atributo private Integer[] memory, donde almacenar valores. Un elemento con valor null indicará que la posición no ha sido utilizada. Como métodos públicos bastarán public boolean write(int pos, int value) y public int read(int pos), para escribir y leer de la memoria.

- OperandStack: Implementa la pila de operandos donde se van apilando elementos de tipo entero. Las instrucciones bytecode que modifican la pila de operandos son PUSH n, LOAD n y las operaciones aritméticas. Esta clase contendrá al menos un atributo privado private int[] stack donde almacenar los operandos, así como métodos públicos para meter y sacar un elemento de la pila.

- ByteCode: Implementa las distintas instrucciones bytecode que puede manejar nuestra máquina virtual. Para representar las distintas instrucciones utiliza un tipo enumerado de nombre ENUM_BYTECODE. Los atributos privados de la clase son private ENUM_BYTECODE name y private int param, donde param es necesario para las instrucciones PUSH, STORE y LOAD.

- CPU: Es la unidad de procesamiento de nuestra máquina virtual. Contiene una memoria, una pila de operandos y una varible booleana para determinar si la ejecución ha terminado, es decir, si se ha ejecutado la instrucción `HALT`. En esta clase se encuentra el método público public boolean execute(ByteCode instr), que es el encargado de ejecutar la instrucción que le llega como parámetro modificando convenientemente la memoria y/o la pila de operandos. Si la ejecución genera un error el método devuelve false.

- ByteCodeParser: Es una clase similar a la clase CommandParser. En este caso es la clase encargada de parsear un string que contiene un posible bytecode. Concretamente dispone de un método public static ByteCode parse(String s) que devuelve el bytecode almacenado en s o bien null si s no representa ningún comando.

- Main: Es la clase que contiene el método main de la aplicación, que en este caso será de la forma:

```
public static void main(String args[]) {
```

```
        Engine engine = new Engine();
        engine.start();
    }
```

El resto de información concreta para implementar la práctica será explicada por el profesor durante las distintas clases de teoría y laboratorio. En esas clases se indicará qué aspectos de la implementación se consideran obligatorios para poder aceptar la práctica como correcta y qué aspectos se dejan a la voluntad de los alumnos. Recuerda además de incluir el método toString en todas las clases.

Entrega de la práctica

La práctica debe entregarse utilizando el mecanismo de entregas del campus virtual, no más tarde de la fecha y hora indicada en la cabecera de la práctica. Debes subir un fichero comprimido (.zip) que contenga al menos el siguiente contenido[1]:

- Directorio `src` con el código de todas las clases de la práctica.
- Directorio `doc` con la documentación de la práctica generada con javadoc.
- Fichero `alumnos.txt` donde se indicará el nombre de los componentes del grupo.

[1] Puedes incluir también opcionalmente los ficheros de información del proyecto de Eclipse