Práctica 2: Máquina Virtual con Instrucciones de Salto

Fecha de entrega: 12 de Diciembre de 2016, 9:00

Objetivo: Herencia, polimorfismo, vinculación dinámica y clases abstractas.

1. Nueva funcionalidad

En esta nueva práctica desaparece el comando NEWINST BC, y aparece el nuevo comando BYTECODE que permite al usuario introducir en orden las distintas instrucciones que componen un programa bytecode. Cuando el usuario haya terminado de escribir todas las instrucciones, tecleará "END" (que no es una instrucción bytecode).

Además, y siguiendo con la evolución de nuestra TPMV, añadiremos soporte para nuevas instrucciones. En concreto, instrucciones de comparación e instrucciones de salto para permitir la ejecución de bucles. La introducción de estas instrucciones de salto obliga a que la CPU necesite incorporar un *contador de programa* que marque cuál es la siguiente instrucción a ejecutar.

Nada más arrancar la aplicación, la TPMV estará vacía (es decir, la cpu estará sin contenido en la pila de operandos, memoria y programa). El nuevo comando BYTECODE solicitará entonces al usuario que introduzca, una a una, todas las instrucciones del programa bytecode a ejecutar. Debido a la existencia de instrucciones de salto, la última instrucción del mismo *no* tiene por qué ser `HALT` por lo que el usuario utilizará la palabra `END` para indicar que ya no quedan más instrucciones que añadir. Tras la ejecución de cada comando, la TPMV mostrará el programa bytecode almacenado.

2. Nuevas instrucciones

Las nuevas instrucciones bytecode que se incorporar son la siguientes:

- *Instrucciones de salto condicional.* Concretamente habrá cuatro instrucciones de este tipo, que son IFEQ N, IFLE N, IFLEQ N y IFNEQ N. Estas instrucciones cogen la

subcima sc y la cima c de la pila (borrándolas) y *comparan* sus valores enteros. Si la condición no se cumple entonces provocan un salto a la instrucción N del programa. Si la condición se cumple entonces se ejecuta la siguiente instrucción. Al contrario de los operadores aritméticas, no escriben ningún resultado en la pila. El proceso de comparación es el siguiente: IFEQ comprueba que sc es igual a c; La instrucción IFNEQ chequea que sc es distinto de c; De forma similar, IFLE (respectivamente IFLEQ) comprueba que sc es menor (respectivamente menor o igual) que c.

- *Instrucción de salto incondicional.* Esta instrucción, cuya sintaxis es GOTO N, provoca un cambio en el contador de programa de la cpu, que pasa a ser N. Es decir el efecto de esta instrucción es pasar a ejecutar la instrucción N.

Por ejemplo, un programa bytecode que calcula el factorial de 5 tendría la siguiente forma (mostramos como se introduce el programa y el resultado de la ejecución):

```
> bytecode
Comienza la ejecución de BYTECODE
Introduce el bytecode. Una instruccion por línea:

push 5
store 0
push 1
store 1
push 0
load 0
ifle 16
load 0
load 1
mul
store 1
load 0
push 1
sub
store 0
goto 4
halt
end

Programa almacenado:

0: PUSH 5
1: STORE 0
2: PUSH 1
3: STORE 1
4: PUSH 0
5: LOAD 0
6: IFLEQ 16
7: LOAD 0
8: LOAD 1
9: MUL
```

```
10: STORE 1
11: LOAD 0
12: PUSH 1
13: SUB
14: STORE 0
15: GOTO 4
16: HALT

> run
Comienza la ejecución de RUN
El estado de la maquina tras ejecutar programa es:

Estado de la CPU:
  Memoria:  [0]:0 [1]:120
  Pila: <vacía>

Programa almacenado:
0: PUSH 5
1: STORE 0
2: PUSH 1
3: STORE 1
4: PUSH 0
5: LOAD 0
6: IFLEQ 16
7: LOAD 0
8: LOAD 1
9: MUL
10: STORE 1
11: LOAD 0
12: PUSH 1
13: SUB
14: STORE 0
15: GOTO 4
16: HALT

>
```

3. Implementación

La implementación de la Práctica 2 preserva las clases utilizadas en la Práctica 1, modificadas convenientemente para incluir herencia y polimorfismo. De hecho, gracias a la herencia y polimorfismo, podemos eliminar los tipos enumerados que utilizamos en la Práctica 1. Para implementar la práctica necesitarás las siguientes clases:

- Engine: Esta clase es similar a la de la Práctica 1. Como ahora ha desaparecido el comando NEWINST, debes de eliminar de esta clase su método de ejecución asociado. Por otro lado, la presencia del comando BYTECODE obliga a implementar un método público en esta clase: public boolean readByteCodeProgram(), que lleva a cabo el proceso de lectura de las instrucciones y carga el atributo ByteCodeProgram program

de esta clase.

- Command: Ahora esta clase pasa a ser abstracta, y contiene tres métodos abstractos:

  abstract public boolean execute(Engine engine): Ejecuta el comando dando la orden pertinente a engine en caso de que sea necesario. Si la ejecución es errónea, devuelve false. En otro devuelve true.

  abstract public Command parse(String[ ] s): Recibe un array de String, lo parsea, y en caso de que se ajuste a la sintaxis del comando en cuestión, entonces lo devuelve. En otro caso devuelve null.

  abstract public String textHelp(): Devuelve en forma de String la información de ayuda correspondiente al comando.

  De esta clase heredan seis clases no abstractas, una por cada uno de los comandos de la aplicación. Por ejemplo, el comando *reset* tendrá asociado la clase Reset que hereda de Command e implementa los tres métodos abstractos. En este caso la implementación de los tres métodos sería:

```
public boolean execute(Engine engine) {
  return engine.resetProgram();
}
public Command parse(String[] s) {
  if (s.length==1 && s[0].equalsIgnoreCase("RESET"))
       return new Reset();
  else return null;
}
public String textHelp() {
  return "  RESET: Vacia el programa actual " +
            System.getProperty("line.separator");
}
```

  El resto de clases AddByteCodeProgram, Help, Quit, Replace y Run se implementarían de forma similar. Además todas estas clases implementan el método toString().

- CommandParser: Contiene dos métodos: public static Command parse(String line), que ya existía en la Práctica 1, y el nuevo método public static void showHelp() que muestra por pantalla la ayuda asociada al conjunto de comandos. Como ahora los comandos están organizados en una jerarquía de clases, el método parse simplemente invoca al método parse de cada uno de los comandos hasta que alguno de ellos devuelve un valor distinto de null. Para ello esta clase necesita el siguiente atributo:

```
private final static Command[] commands = {
      new Help(), new Quit(), new Reset(),
      new Replace(), new Run(), new AddByteCodeProgram() }
```

- ByteCodeProgram, Memory, OperandStack, Main: Igual en en la Práctica 1.

- ByteCode: Esta clase pasa a ser una clase abstracta de la que heredan las distintas instrucciones bytecode de las que puede constar un programa. Contiene como métodos:

abstract public boolean execute(CPU cpu): ejecuta el correspondiente bytecode en la cpu.

abstract public ByteCode parse(String[ ] s): parsea s y en caso de que corresponda a la sintaxis del bytecode correspondiente, entonces lo devuelve. En otro caso devuelve null.

De esta clase heredan todas las instrucciones bytecode de la práctica. Puedes organizar a su vez estas clases formando nuevas jerarquías. Por ejemplo, se puede crear una clase abstracta Arithmetics, que herede de ByteCode, y de la cual hereden a su vez las clases Add, Sub, Div y Mul, que corresponden a los bytecodes aritméticos. Estas operaciones tienen parte de su ejecución y parseo común, que debe implementarse en la clase Arithmetics. Lo mismo ocurre con las instrucciones de salto condicional. Puedes crear una clase abstracta ConditionalJumps que herede de ByteCode, y de la cual herenden las clases IfEq, IfNeq, IfLe e IfLeq.

- ByteCodeParser: Es una clase similar a la clase CommandParser.
- CPU: Además de los atributos de la Práctica 1, ahora la CPU contiene dos nuevos atributos: private int programCounter y private ByteCodeProgram bcProgram. El primero indica la siguiente instrucción a ejecutar, y el segundo contiene el programa bytecode a ejecutar. Esta clase es ahora la encargada de ejecutar el programa bytecode completo, y por lo tanto necesita un método public boolean run() que se encarga de ello, devolviendo false si se ha producido algún error de ejecución. El método selecciona el bytecode indicado por programCounter, y le manda ejecutarse. El proceso continua hasta que se ejecute la instrucción HALT o se produzca un error de ejecución. Dado que los distintos bytecodes deben interaccionar con la CPU, esta clase tendrá además métodos públicos que permitan dicha interacción, como por ejemplo public boolean push(int n), que introduce el elemento n en la pila, o public boolean read(int n), que devuelve el contenido existente en la posición n de la memoria.

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.

4. Ejemplos de ejecución

Los siguientes ejemplos de ejecución muestran el uso de las instrucciones de salto. El primer programa calcula $1! + \ldots + 5!$. El segundo programa es equivalente al primero pero implementado utilizando una condición distinta en el primer while. Finalmente el tercer programa se corresponde con:

```
x=10;
if (x=10) y = 7;
else y = 5;
```

Recuerda que en la Sección 2 tienes además la ejecución de un programa que calcula factorial de 5.

```
> bytecode
Comienza la ejecución de BYTECODE
Introduce el bytecode. Una instruccion por línea:

push 5
store 0
push 0
store 1
push 0
load 0
ifle 32
load 0
store 2
push 1
store 3
push 0
load 2
ifle 23
load 2
load 3
mul
store 3
load 2
push 1
sub
store 2
goto 11
load 1
load 3
add
store 1
load 0
push 1
sub
store 0
goto 4
halt
end

Programa almacenado:

0: PUSH 5
1: STORE 0
2: PUSH 0
3: STORE 1
4: PUSH 0
5: LOAD 0
6: IFLE 32
7: LOAD 0
```

```
8: STORE 2
9: PUSH 1
10: STORE 3
11: PUSH 0
12: LOAD 2
13: IFLE 23
14: LOAD 2
15: LOAD 3
16: MUL
17: STORE 3
18: LOAD 2
19: PUSH 1
20: SUB
21: STORE 2
22: GOTO 11
23: LOAD 1
24: LOAD 3
25: ADD
26: STORE 1
27: LOAD 0
28: PUSH 1
29: SUB
30: STORE 0
31: GOTO 4
32: HALT

> run
Comienza la ejecución de RUN
El estado de la maquina tras ejecutar programa es:

Estado de la CPU:
  Memoria:  [0]:0 [1]:153 [2]:0 [3]:1
  Pila: <vacia>

Programa almacenado:

0: PUSH 5
1: STORE 0
2: PUSH 0
3: STORE 1
4: PUSH 0
5: LOAD 0
6: IFLE 32
7: LOAD 0
8: STORE 2
9: PUSH 1
10: STORE 3
11: PUSH 0
12: LOAD 2
```

```
13: IFLE 23
14: LOAD 2
15: LOAD 3
16: MUL
17: STORE 3
18: LOAD 2
19: PUSH 1
20: SUB
21: STORE 2
22: GOTO 11
23: LOAD 1
24: LOAD 3
25: ADD
26: STORE 1
27: LOAD 0
28: PUSH 1
29: SUB
30: STORE 0
31: GOTO 4
32: HALT

> replace 4
Comienza la ejecución de REPLACE 4
Nueva instrucción: push 1

Programa almacenado:

0: PUSH 5
1: STORE 0
2: PUSH 0
3: STORE 1
4: PUSH 1
5: LOAD 0
6: IFLE 32
7: LOAD 0
8: STORE 2
9: PUSH 1
10: STORE 3
11: PUSH 0
12: LOAD 2
13: IFLE 23
14: LOAD 2
15: LOAD 3
16: MUL
17: STORE 3
18: LOAD 2
19: PUSH 1
20: SUB
21: STORE 2
```

```
22: GOTO 11
23: LOAD 1
24: LOAD 3
25: ADD
26: STORE 1
27: LOAD 0
28: PUSH 1
29: SUB
30: STORE 0
31: GOTO 4
32: HALT

> replace 6
Comienza la ejecución de REPLACE 6
Nueva instrucción: ifleq 32

Programa almacenado:

0: PUSH 5
1: STORE 0
2: PUSH 0
3: STORE 1
4: PUSH 1
5: LOAD 0
6: IFLEQ 32
7: LOAD 0
8: STORE 2
9: PUSH 1
10: STORE 3
11: PUSH 0
12: LOAD 2
13: IFLE 23
14: LOAD 2
15: LOAD 3
16: MUL
17: STORE 3
18: LOAD 2
19: PUSH 1
20: SUB
21: STORE 2
22: GOTO 11
23: LOAD 1
24: LOAD 3
25: ADD
26: STORE 1
27: LOAD 0
28: PUSH 1
29: SUB
30: STORE 0
```

```
31: GOTO 4
32: HALT

> run
Comienza la ejecución de RUN
El estado de la maquina tras ejecutar programa es:

Estado de la CPU:
  Memoria:  [0]:0 [1]:153 [2]:0 [3]:1
  Pila: <vacia>

Programa almacenado:

0: PUSH 5
1: STORE 0
2: PUSH 0
3: STORE 1
4: PUSH 1
5: LOAD 0
6: IFLEQ 32
7: LOAD 0
8: STORE 2
9: PUSH 1
10: STORE 3
11: PUSH 0
12: LOAD 2
13: IFLE 23
14: LOAD 2
15: LOAD 3
16: MUL
17: STORE 3
18: LOAD 2
19: PUSH 1
20: SUB
21: STORE 2
22: GOTO 11
23: LOAD 1
24: LOAD 3
25: ADD
26: STORE 1
27: LOAD 0
28: PUSH 1
29: SUB
30: STORE 0
31: GOTO 4
32: HALT

> bytecode
Comienza la ejecución de BYTECODE
```

```
Introduce el bytecode. Una instruccion por línea:

push 10
store 0
load 0
push 10
ifEq 8
push 7
store 1
goto 10
push 5
store 1
halt
end

Programa almacenado:

0: PUSH 10
1: STORE 0
2: LOAD 0
3: PUSH 10
4: IFEQ 8
5: PUSH 7
6: STORE 1
7: GOTO 10
8: PUSH 5
9: STORE 1
10: HALT

> run
Comienza la ejecución de RUN
El estado de la maquina tras ejecutar programa es:

Estado de la CPU:
  Memoria:  [0]:10 [1]:7
  Pila: <vacia>

Programa almacenado:

0: PUSH 10
1: STORE 0
2: LOAD 0
3: PUSH 10
4: IFEQ 8
5: PUSH 7
6: STORE 1
7: GOTO 10
8: PUSH 5
9: STORE 1
```

```
10: HALT

>
```

5. 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