Práctica 2

Máquina Virtual
con Instrucciones
de Salto
(curso 2016-2017)

Puri Arenas
Facultad de Informática
UCM

Descripción de la Práctica 2

- Añadir a la TPMV instrucciones bytecode de salto.
- Utilizar herencia y polimorfismo en la implementación.
- Desaparece el comando **NEWINST** y aparece el comando **BYTECODE**. Este comando permite al usuario introducir un programa completo que debe finalizar con la palabra **END**.

Las nuevas instrucciones bytecode de salto son:

- Saltos condicionales: IFEQ N, IFLE N, IFLEQ N, IFNEQ N.
- Salto incondicional: GOTO N

Las nuevas instrucciones representan **bucles** e **if-then-else**.

Correspondencia con un programa imperativo

```
                          tv[0] = x, tv[1]=r
while (0 < x)             (0) push 0, (1) load 0, (2) ifle 12
   r = r * x              (3) load 1, (4) load 0, (5) mul, (6) store 1
   x = x - 1              (7) load 0, (8) push 1, (9) sub (10) store 0
                         (11) goto 0 (12) halt

                          tv[0] = x, tv [1] = y, tv[2] = z
if (x <= y) z = x         (0) load 0, (1) load 1, (2) ifleq 6
else z = y                (3) load 0, (4) store 2, (5) goto 8 (THEN)
                         (6) load 1, (7) store 2, (8) halt    (ELSE)
```


Clase Command

- Es una clase abstracta de la que heredan los distintos comandos de la práctica (HELP, RESET, REPLACE N, RUN, ADDBYTECODEPROGRAM).

```
abstract public class Command {

  abstract public boolean execute(Engine engine);
  abstract public Command parse(String[] s);
  abstract public String textHelp();
}
```

Clase AddByteCodeProgram

```
public class AddByteCodeProgram extends Command {

  @Override
  public boolean execute(Engine engine) {
     return engine.readByteCodeProgram();
  }
  @Override
  public Command parse(String[] s) {
    if (s.length!=1 || !s[0].equalsIgnoreCase("BYTECODE")) return null;
    else return new AddByteCodeProgram();
  }
  @Override
  public String textHelp() {
     return "  BYTECODE: Permite introducir un programa " +
             System.getProperty("line.separator");
  }
  public String toString(){
     return "BYTECODE";
  }
}
```


Clase CommandParser

```
public class CommandParser {

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

  public static Command parse(String line) {
    // eliminar blancos innecesarios igual que en la Práctica 1line = line.trim();
    // descomponer line en  palabras
    boolean found = false;
    int i=0;
    Command c = null;
    while (i < commands.length && !found){
     c = commands[i].parse(words);
     if (c!=null) found=true;
     else i++;
    }
    return c;
  }
  public static void showHelp() {
    for (int i=0; i < CommandParser.commands.length; i++)
        System.out.println(CommandParser.commands[i].textHelp());
  }
}
```


Clase abstracta ByteCode

```
abstract public class ByteCode {
   abstract public boolean execute(CPU cpu);
   abstract public ByteCode parse(String[] words);
}
```

(1) Al igual que para los comandos, hace falta la clase ByteCodeParser, que se implementa de forma similar.

(2) De la clase ByteCode heredan todas las instrucciones bytecode, que a su vez se pueden ordenar jerárquicamente.

Clase abstracta ByteCodeOneParameter

```
public abstract class ByteCodesOneParameter extends ByteCode {

   protected int param;

   public ByteCodesOneParameter(){};
   public ByteCodesOneParameter(int p){ this.param = p; }

   @Override
   public ByteCode parse(String[] words) {
      if (words.length!=2) return null;
      else return this.parseAux(words[0],words[1]);
   }
   abstract protected ByteCode parseAux(String string1, String string2);

   public String toString(){
       return this.toStringAux() + " " + this.param;
   }
   abstract protected String toStringAux();
}
```

(*) De esta clase heredan las instrucciones bytecode con un parámetro, incluidas por tanto las instrucciones condicionales.

Clase abstracta ConditionalJumps

```
abstract public class ConditionalJumps extends ByteCodesOneParameter {

     public ConditionalJumps(){}
     public ConditionalJumps(int j){ super(j); }

    @Override
    public boolean execute(CPU cpu) {
       if (cpu.getSizeStack()>=2){
          ...
          if (compare(n2,n1)) cpu.setProgramCounter(this.param);
          else cpu.increaseProgramCounter();
          return true;
       }
       else return false;
    }
    abstract protected boolean compare(int n1, int n2);
}
```

(*) De esta clase heredan finalmente las clases IfEq, IfLe, IfLeq, IfNeq y ninguna De ellas es abstracta

Resto de Bytecodes

(*) Los comandos arithméticos se pueden implementar de forma similar, creado una clase abstracta, por ejemplo de nombre `Arithmetics`, que extienda a `ByteCode`. De esta nueva clase heredarán las clases `Add`, `Sub`, `Div` y `Mul`

(*) El resto de instrucciones bytecode pueden heredar directamente de la clase `ByteCode`.

```
public class Halt extends ByteCode {
  public boolean execute(CPU cpu) {
    cpu.halt();
    return true;
  }
  public ByteCode parse(String[] words) {...}
  public String toString(){ return "HALT"; }
}
```


Clase CPU

```
public class CPU {
  // memory, stack y halt como en la práctica 1
  private int programCounter = 0;
  private ByteCodeProgram bcProgram = new ByteCodeProgram();

  public CPU(ByteCodeProgram program){ this.bcProgram = program;}
  public void halt() { this.halt = true; }
  public String toString(){ … }
  public int getSizeStack() {...}
  public int pop() { return this.stack.pop(); }
  public boolean push(int i) {...}
  public boolean read(int param) {...}
  public void write(int param, int value) {...}
  public void setProgramCounter(int jump) {...}
  public void increaseProgramCounter() {...}

  public boolean run() {
    this.programCounter=0;
    boolean error = false;
    while (this.programCounter < bcProgram.getNumeroInstrucciones() && ...) {
        ByteCode bc = bcProgram.getByteCode(this.programCounter);
        if (!bc.execute(this)) // salir del bucle
    }
    return // ejecución correcta?
  }
```


Resto de clases

* **La clase Engine es igual que en la Práctica 1, pero incorpora el nuevo metodo “`readByteCodeProgram()`”.**

* **Las clases `Memory`, `OperandStack` y `ByteCodeProgram` iguales a la Práctica 1.**

* **Los tipos enumerados desaparecen gracias a la herencia.**