
Cuadernillo de prácticas Tecnología de la Programación



Curso: 2016/2017

Departamento de Ingeniería del Software e Inteligencia Artificial
Departamento de Sistemas Informáticos y Computación
Facultad de Informática
UNIVERSIDAD COMPLUTENSE DE MADRID

Práctica 3: Compilador para la

TPMV

Fecha de entrega: 16 de Enero de 2017, 9:00

Objetivo: Herencia, polimorfismo, vinculación dinámica y clases abstractas, tratamiento de excepciones e interfaces.

1. Descripción de la práctica

En las prácticas anteriores hemos utilizado programas bytecode, directamente escritos por el usuario, y ejecutados a través de una máquina virtual. En esta práctica vamos a permitir que el usuario escriba programas en un lenguaje imperativo simple, de forma que estos programas sean traducidos a programas bytecode que más tarde ejecutará la TPMV. Los programas imperativos (a partir de ahora *programas fuente*) se cargarán de ficheros de texto y serán secuencias de instrucciones. Es decir, un programa fuente responde a la siguiente gramática:

SourceProgram	::=	Instruction ₁ ... Instruction _n
Instruction	::=	SimpleAssignment CompoundAssignment While If Write Return
SimpleAssignment	::=	Variable = Term
CompoundAssignment	::=	Variable = Term ArithmeticOper Term
While	::=	while BooleanCond SourceProgram endwhile
If	::=	if BooleanCond SourceProgram endif
Write	::=	write Variable
Term	::=	Variable Number
ArithmeticOper	::=	+ - / *
Variable	::=	a ... z
Number	::=	any integer number
BooleanCond	::=	Term BooleanOper Term
BooleanOper	::=	= != <= <

Los programas fuente escritos en un fichero deben cumplir además los siguientes requisitos sintácticos:

- Cada asignación o instrucción `write` debe aparecer en una línea;
- No puede haber líneas en blanco en ningún sitio del fichero, ni tampoco al principio ni al final del mismo.
- Una instrucción `while` debe aparecer como:

```
while BooleanCond
    instruccion_1
    instruccion_2
    ...
endwhile
```

- Una instrucción `if` debe aparecer como:

```
if BooleanCond
    instruccion_1
    instruccion_2
    ...
endif
```

A continuación mostramos un programa correcto (izquierda) que calcula en la variable `r` el factorial de 5 y lo muestra por pantalla. El mismo programa de la derecha es sintácticamente incorrecto debido a que aparece en la segunda instrucción `r= 1`, es decir no hay blanco entre la `r` y el `=`. Lo mismo ocurre en la condición del `while` `0 <x`.

<pre>x = 5 r = 1 while 0 < x r = r * x x = x - 1 endwhile write r end</pre>	<pre>x = 5 r= 1 while 0 <x r = r * x x = x - 1 endwhile write r end</pre>
--	--

En el siguiente ejemplo, el programa de la izquierda es incorrecto porque la asignación está en la misma línea que la condición del `if`. El mismo programa a la derecha sí es correcto.

<pre>x = -5 if x < 0 x = 0 - x endif write x end</pre>	<pre>x = -5 if x < 0 x = 0 - x endif write x end</pre>
--	---

2. Conjunto de comandos de la práctica

En esta nueva práctica desaparecen varios comandos usados en las prácticas anteriores, y aparecen otros nuevos. Los únicos comandos que se admiten son los siguientes.

- **HELP**: muestra la ayuda correspondiente a los comandos que describiremos a continuación.
- **QUIT**: cierra la aplicación.
- **LOAD FICH**: carga el fichero de nombre **FICH** como programa fuente. No realiza ningún tipo de comprobación sintáctica.
- **REPLACEBC N**: solicita al usuario una nueva instrucción bytecode y reemplaza la línea **N** del programa bytecode por la nueva instrucción introducida por el usuario.
- **COMPILE**: realiza el análisis léxico del programa fuente, generando un nuevo programa parseado y, posteriormente a partir del programa parseado genera un programa bytecode.
- **RUN**: ejecuta el programa bytecode igual que en la Práctica 2.

Salvo por el tratamiento de excepciones, en la Práctica 3 tendremos una interfaz **Command**, igual que la clase abstracta correspondiente en la Práctica 2, y por cada uno de los comandos descritos anteriormente habrá una clase (no abstracta) implementando la interfaz **Command**. Así pues, existirá una clase **Compile** que tendrá la siguiente implementación del método **execute**:

```
@Override
public void execute(Engine engine)
    throws LexicalAnalysisException, ArrayException {
    engine.compile();
}
```

3. Traducción de un programa fuente a un programa bytecode

El proceso de compilación de un programa fuente a su correspondiente programa bytecode se lleva a cabo a través del comando **COMPILE** y su método **execute(Engine engine)**, tal y como se acaba de decir en la sección anterior. La clase **Engine** tendrá al menos los siguientes atributos:

- **private SourceProgram sProgram**: donde se almacena el programa fuente que se carga del fichero de texto.
- **private ParsedProgram parsedProgram**: donde se almacena el programa parseado, una vez realizado el análisis léxico sobre el programa fuente.
- **private ByteCodeProgram bytecodeProgram**: donde se almacena el programa bytecode, una vez realizada la compilación.

junto con un método público **compile**, que consta de dos fases:

- (1) Primero el programa fuente se transforma en un *programa parseado*.
- (2) Si la fase anterior ha tenido éxito, es decir, no se ha producido ninguna excepción, entonces el programa parseado se compila para generar el *programa bytecode*.

Así pues, el método `compile` básicamente tendrá esta implementación:

```
public void compile() throws ... {
    try {
        this.lexicalAnalysis();
        this.generateByteCode();
    }
    catch ...
}
```

donde `lexicalAnalysis()` genera el programa parseado a partir del programa fuente, y `generateByteCode()` genera el programa bytecode a partir del programa parseado. Ambos métodos son privados dentro de la clase `Engine`. El método privado `lexicalAnalysis()` (respectivamente `generateByteCode()`) utiliza un objeto `LexicalParser lexicalParser` (respectivamente `Compiler compiler`) para realizar el análisis léxico (respectivamente, generar el bytecode). A continuación pasamos a describir ambas clases.

3.1. Análisis léxico

Para implementar la práctica, necesitamos una nueva clase `SourceProgram` para almacenar un programa fuente. La clase `SourceProgram` se implementa de forma similar a la clase `ByteCodeProgram` de la Práctica 2, excepto que contiene un array de elementos de tipo `String` (`private String[] sProgram`) en lugar de un array de elementos de tipo `ByteCode`. En esta clase se almacenará, línea a línea, el fichero de texto que se cargue. Observa que las componentes de `sProgram` puede que no contengan una instrucción en cada componente. Esto ocurre en presencia de `while` e `if`, que se almacenan en varias componentes de `sProgram`. Por ejemplo, si el fichero está compuesto por las líneas:

```
r = 0
x = 3
while 0 < x
    r = r + x
    x = x - 1
endwhile
end
```

entonces `sProgram[0]="r = 0"`, `sProgram[1]="x = 3"`, `sProgram[2]="while 0 < x"`, `sProgram[3]="r = r + x"`, `sProgram[4]="x = x - 1"`, `sProgram[5]="endwhile"` y `sProgram[6]="end"`.

Además de los programas fuente, tenemos los programas parseados. Por lo tanto necesitamos también una clase `ParsedProgram` para representarlos. La única diferencia entre esta clase y la anterior es que esta última almacena elementos del tipo `Instruction`, que son las componentes resultantes de llevar a cabo el análisis léxico. `Instruction` se implementará como una interfaz

```
public interface Instruction {
    Instruction lexParse(String[] words, LexicalParser lexParser);
    void compile(Compiler compiler) throws ...;
}
```

que será implementada por las clases `SimpleAssignment`, `CompoundAssignment`, `While`, `IfThen`, `Return` y `Write`. Dado que estamos en la fase de análisis léxico vamos a centrarnos en cómo se implementa el método `Instruction lexParse(String[] words, LexicalParser lexparser)` en algunas de las clases asociadas a las instrucciones.

- Clase `SimpleAssignment`. Esta clase almacena asignaciones simples de la forma `Variable = Term`. Por lo tanto tendrá dos atributos privados `String varName` y `Term rhs`, donde `varName` es el nombre de la variable del lado izquierdo de la asignación, y `rhs` es el término que aparece en el lado derecho. `Term` es una interfaz definida como:

```
public interface Term {
    Term parse(String term);
    ByteCode compile(Compiler compiler);
}
```

que es implementada por las clases `Variable` y `Number`. La clase `Variable` implementa el método `parse` como:

```
@Override
public Term parse(String term) {
    if (term.length() != 1) return null;
    else{
        char name = term.charAt(0);
        if ('a' <= name && name <= 'z') return new Variable(term);
        else return null;
    }
}
```

es decir, una variable ha de ser una letra. La clase `Number` hace lo propio y parsea un entero. En este caso debes utilizar tratamiento de excepciones. Con estas dos clases, y volviendo a `SimpleAssignment`, su método `lexParse(String[] words, LexicalParser lexparser)` haría lo siguiente: (1) si la longitud de `words` es distinta de 3 o `words[0]` no es un carácter entre `a ... z` o `words[1]` es distinto de `"="`, devuelve `null`. En otro caso realiza la llamada `TermParser.parse(words[2])` para devolver el parseo del término del lado derecho de la asignación. La clase `TermParser` se implementa igual que cualquier parser de la Práctica 2.

- Clase `CompoundAssignment`. Representa asignaciones de la forma `Variable = Term ArithmeticOper Term`. Por lo tanto tiene cuatro atributos privados `String varName`, `String operator`, `Term term1` y `Term term2`. Su método `lexParse` se implementa de forma similar al implementado en `SimpleAssignment`, utilizando `TermParser`.
- Clase `While`. Esta clase representa un bucle, y por lo tanto necesita almacenar la condición del mismo y su cuerpo. Así pues contiene los atributos privados `Condition condition` y `ParsedProgram body`. Para implementar su método `lexParse(String[]`

`words`, `LexicalParser lexical`) es necesario parsear la condición y el cuerpo del bucle. Para el caso de la condición necesitamos crear una nueva clase `Condition`. Esta clase representa condiciones booleanas de la forma `Term1 BooleanOper Term2`. Por lo tanto tiene atributos `Term term1` y `Term term2`, junto con los métodos `public Condition parse(String t1, String op, String t2, LexicalParser parser)` y `public void compile(Compiler compiler)`. Para la compilación usará, además, un atributo `ConditionalJump condition`. De esta clase heredan las clases `Less`, `LessEq`, `Equal` y `NotEqual` que representan los cuatro operadores booleanos. Con estas clases, el método `lexParse(String[] words, LexicalParser lexical)` de la clase `While` debe parsear la condición con la llamada `ConditionParser.parse(words[1], words[2], words[3], lexical)`, donde `ConditionParser` es una clase similar al resto de parsers implementados, y posteriormente debe parsear el cuerpo utilizando:

```
ParsedProgram whileBody = new ParsedProgram();
lexical.lexicalParser(wbody, "ENDWHILE");
lexical.increaseProgramCounter();
```

Sobre la definición de estos dos métodos de `LexicalParser`, léase más abajo.

- Clases `IfThen`, `Return` y `Write`. La primera de ellas tiene una implementación similar (más simple) que la clase `While`. La segunda y la tercera tienen una implementación trivial.

Para finalizar con el análisis léxico, nos queda la implementación de la clase `LexicalParser`, encargada de realizar el análisis léxico. La clase contiene atributos privados `SourceProgram sProgram` y `private int programCounter`. Este último controla desde qué componente de `sProgram` se va a hacer el análisis léxico. Esto es necesario ya que, como se ha explicado previamente, las componentes de `sProgram` no tienen por qué contener una instrucción completa. El método más importante de la clase `LexicalParser` es:

```
public void lexicalParser(ParsedProgram pProgram, String stopKey) throws ...
```

donde `stopKey` indica dónde debe finalizar el proceso de análisis. Para el caso de una instrucción `While`, finaliza en `"ENDWHILE"`. Si es una instrucción `IfThen`, finaliza en `"ENDIF"`. Si es un programa completo, finaliza en `"END"`. Este método recorre las instrucciones almacenadas en `sProgram`. Parsea cada instrucción a través de la llamada `ParserInstruction.parse(line, this)` y, en caso de que el parseo haya sido correcto, almacena la instrucción en `pProgram`. De nuevo aparece una clase que parsea `InstructionParser` que se implementa de forma similar al resto de parsers.

3.2. Generación del programa bytecode

Una vez realizado el análisis léxico, y almacenado en el atributo `parsedProgram` de la clase `Engine`, es necesario realizar la generación del programa bytecode, que lleva a cabo el método `generateByteCode()` de la clase `Engine`. Para ello necesitamos utilizar una nueva clase `Compiler`. Esta clase, encargada de generar el programa bytecode, tiene como atributos privados `ByteCodeProgram bytecode`, `String[] varTable` y `int numVars`. El primer atributo almacenará el programa generado, mientras que los dos restantes son para controlar las posiciones en memoria de las variables. El método más importante de esta clase es:


```
public void compile(ParsedProgram pProgram) throws ... {
    int i = 0;
    try {
        while (i < pProgram.getNumeroInstrucciones()) {
            Instruction instr = pProgram.getInstruction(i);
            instr.compile(this);
            i++;
        }
    }
    catch ...
}
```

Pasamos entonces a describir cómo implementar el método `compile(Compiler compiler)` de las instrucciones `SimpleAssignment`, `CompoundAssignment`, `Write`, `While`, `IfThen`, `Return` y `Condition`. Este método genera las instrucciones bytecode asociadas a la instrucción correspondiente, y las añade al programa bytecode a través de `compiler`. Antes de pasar a detallar las clases, vamos a explicar brevemente en qué consiste el proceso de generación de bytecodes a partir de una instrucción.

- **Variable(varName):** Una variable genera el bytecode `LOAD i`, donde `i` es el índice que ocupa la variable `varName` en la tabla de variables de la clase `Compiler`.
- **Number(n):** Un número se traduce a `PUSH n`.
- **SimpleAssignment(varName,term):** Representa la instrucción `varName = term` y se traduce primero generando el bytecode asociado a `term` tal y como se especifica en los dos items previos, seguido de `STORE i`, donde `i` es el índice que ocupa `varName` en la tabla de variables de la clase `Compiler`.
- **CompoundAssignment(varName, term1, ArithmeticOper, term2):** Representa la asignación `varName = term1 ArithmeticOper term2`. Los bytecodes asociados a esta instrucción son los siguientes:
 - Se generan los bytecodes asociados a `term1`;
 - Se generan los bytecodes asociados a `term2`;
 - Si `ArithmeticOper` es `“+”`, generamos `ADD`. Si es `“-”`, generamos `SUB`. Para `“/”` generamos `DIV`. Finalmente para `“*”` generamos `MUL`.
 - Se termina añadiendo `STORE i`, donde `i` es el índice que ocupa `varName` en la tabla de variables de la clase `Compiler`.
- **Write(varName):** Genera `LOAD i, OUT`, donde `i` es el índice que ocupa `var_name` en la tabla de variables de la clase `Compiler`.
- **return:** Genera `HALT`.
- **Condition(term1,BooleanOper,term2):** Se generan los bytecodes asociados a `term1` y `term2`. Posteriormente, dependiendo del operador booleano, generamos el siguiente bytecode:
 - si `BooleanOper` es `“=”`, generamos `IfEq`
 - si `BooleanOper` es `“<”`, generamos `IfLe`

- si `BooleanOper` es “<=”, generamos `IfLeq`
 - si `BooleanOper` es “!=”, generamos `IfNeq`
que da valor al atributo `ConditionalJump condition` que se mencionó en el análisis léxico del `while`.
- `IfThen(condition,body)`: Generamos los bytecodes asociados a la compilación de `condition` tal y como se ha explicado en el item previo. En este punto todavía no sabemos a qué instrucción hay que saltar en caso de que no se cumpla la condición. Compilamos `body`. Finalmente modificamos el valor del salto condicional de `condition` al tamaño del programa tras las dos compilaciones que acabamos de hacer.
 - `While(condition,body)`: El proceso de traducción es similar al `IfThen`, pero tras la traducción de `body` hay que añadir el bytecode `GOTO pc1`, donde `pc1` es el tamaño del programa bytecode de `Compiler` antes de empezar la compilación del `while`.

Por ejemplo, consideremos el programa fuente:

```
x = 5
r = 1
while 0 < x
    r = r * x
    x = x - 1
endwhile
write r
return
end
```

La fase del análisis léxico genera un programa parseado compuesto por las instrucciones de la izquierda mientras que la fase de compilación genera los bytecodes de la derecha:

<code>SimpleAssignment(x,5)</code>	(0) PUSH 5, (1) STORE 0
<code>SimpleAssignment(r,1)</code>	(2) PUSH 1, (3) STORE 1
<code>While(Less(0,x),</code>	(4) PUSH 0, (5) LOAD 0, (6) IFLE 16
<code>CompoundAssignment(r,r,*,x)</code>	(7) LOAD 1, (8) LOAD 0, (9) MUL, (10) STORE 1
<code>CompoundAssignment(x,x,-,1))</code>	(11) LOAD 0, (12) PUSH 1, (13) SUB, (14) STORE 0, (15) GOTO 4
<code>Write(r)</code>	(16) LOAD 1, (17) OUT
<code>Return</code>	(18) HALT

A modo de ejemplo, presentamos cómo sería la implementación del método `compile` para la clase `IfThen`:

```
public void compile(Compiler compiler) throws ... {
    condition.compile(compiler);
    compiler.compile(this.body);
    ConditionalJump cj=condition.cj;
    int n=compiler.getSizeBcProgram();
    cj.setN(n);
}
```

4. Implementación

Enumeramos a continuación las clases necesarias para implementar la práctica. No incluimos detalles puesto que ya han sido explicados anteriormente:

- Interfaz `ByteCode` y el resto de clases que implementan a dicha interfaz, que coinciden con las de la Práctica 2.
- Clase `ByteCodeParser`: Igual que la Práctica 2.
- Interfaz `Command` y una clase por cada uno de los comandos que aparecen en la Sección 2, que implementan la interfaz.
- Clase `CommandParser` similar a la de la Práctica 2.
- Clases `Compiler`, `LexicalParser`, `ParsedProgram` y `SourceProgram` implementadas como se ha descrito en la Sección 3.
- Interfaz `Instruction` y las clases `Return`, `Write`, `SingleAssignment`, `CompoundAssignment`, `While` e `IfThen` que implementan la interfaz.
- Clase `Condition` y las clases `Less`, `LessEq`, `Equal` y `NotEqual` que heredan de ella.
- Clase `ConditionParser`.
- Clase `InstructionParser`.
- Interfaz `Term` y las clases `Variable` y `Number` que la implementan.
- Clase `TermParser`.
- Clases `Engine`, `BytecodeProgram` y `Main` similares a las de la Práctica 2, salvo por pequeñas modificaciones.
- Clases `CPU`, `Memory`, `OperandStack` similares a las de la Práctica 2.
- Clases para implementar excepciones. Al menos debe haber las siguientes clases:
 - `ArrayException`: Para contemplar errores de acceso a posiciones incorrectas de un array.
 - `BadFormatByteCode`: Se utiliza para detectar que un posible bytecode no se ajusta a la sintaxis especificada.
 - `DivisionByZero`: Detecta divisiones por cero.
 - `LexicalAnalysisException`: Detecta errores que se pueden producir en el análisis léxico, por incorrecciones en el programa fuente.
 - `ExecutionError`: Se usa para errores en ejecución.
 - `StackException`: Detecta errores que se producen en la pila de operandos al ejecutar un bytecode.

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 incluir el método `toString()` en (casi) todas las clases.

5. Ejemplos de ejecución

Los siguientes ejemplos de ejecución muestran el uso de las instrucciones de la práctica 3.

```
> load simplewhile.txt
Comienza la ejecución de LOAD simplewhile.txt
Programa fuente almacenado:
0: x = 5
1: r = 1
2: while 0 < x
3:   r = r * x
4:   x = x - 1
5: endwhile
6: write r
7: return
8: end
```

```
> compile
Comienza la ejecución de COMPILE
Programa fuente almacenado:
0: x = 5
1: r = 1
2: while 0 < x
3:   r = r * x
4:   x = x - 1
5: endwhile
6: write r
7: return
8: end
```

```
Programa bytecode almacenado:
0: PUSH 5
1: STORE 0
2: PUSH 1
3: STORE 1
4: PUSH 0
5: LOAD 0
6: IFLE 16
7: LOAD 1
8: LOAD 0
9: MUL
10: STORE 1
11: LOAD 0
12: PUSH 1
13: SUB
14: STORE 0
15: GOTO 4
16: LOAD 1
17: OUT
```

18: HALT

> run

Comienza la ejecución de RUN

consola: 120

El estado de la maquina tras ejecutar programa es:

Estado de la CPU:

Memoria: [0]:0 [1]:120

Pila: <vacía>

Programa fuente almacenado:

0: x = 5

1: r = 1

2: while 0 < x

3: r = r * x

4: x = x - 1

5: endwhile

6: write r

7: return

8: end

Programa bytecode almacenado:

0: PUSH 5

1: STORE 0

2: PUSH 1

3: STORE 1

4: PUSH 0

5: LOAD 0

6: IFLE 16

7: LOAD 1

8: LOAD 0

9: MUL

10: STORE 1

11: LOAD 0

12: PUSH 1

13: SUB

14: STORE 0

15: GOTO 4

16: LOAD 1

17: OUT

18: HALT

> replacebc 0

Comienza la ejecución de REPLACEBC 0

Nuevo bytecode: add

Programa fuente almacenado:

0: x = 5

1: r = 1

```
2: while 0 < x
3:   r = r * x
4:   x = x - 1
5: endwhile
6: write r
7: return
8: end
```

Programa bytecode almacenado:

```
0: ADD
1: STORE 0
2: PUSH 1
3: STORE 1
4: PUSH 0
5: LOAD 0
6: IFLE 16
7: LOAD 1
8: LOAD 0
9: MUL
10: STORE 1
11: LOAD 0
12: PUSH 1
13: SUB
14: STORE 0
15: GOTO 4
16: LOAD 1
17: OUT
18: HALT
```

> run

Comienza la ejecución de RUN

Excepcion en la ejecucion del bytecode 0

Excepcion-bytecode ADD: Tamaño de pila insuficiente

> load nestedwhile.txt

Comienza la ejecución de LOAD nestedwhile.txt

Programa fuente almacenado:

```
0: x = 5
1: r = 0
2: while 0 < x
3:   y = x
4:   s = 1
5:   while 0 < y
6:     s = s * y
7:     y = y - 1
8:   endwhile
9:   write s
10:  r = r + s
11:  x = x - 1
12: endwhile
```

```
13: return
14: end
```

```
> compile
```

Comienza la ejecución de COMPILE

Programa fuente almacenado:

```
0: x = 5
1: r = 0
2: while 0 < x
3:   y = x
4:   s = 1
5:   while 0 < y
6:     s = s * y
7:     y = y - 1
8:   endwhile
9:   write s
10:  r = r + s
11:  x = x - 1
12: endwhile
13: return
14: end
```

Programa bytecode almacenado:

```
0: PUSH 5
1: STORE 0
2: PUSH 0
3: STORE 1
4: PUSH 0
5: LOAD 0
6: IFLE 34
7: LOAD 0
8: STORE 2
9: PUSH 1
10: STORE 3
11: PUSH 0
12: LOAD 2
13: IFLE 23
14: LOAD 3
15: LOAD 2
16: MUL
17: STORE 3
18: LOAD 2
19: PUSH 1
20: SUB
21: STORE 2
22: GOTO 11
23: LOAD 3
24: OUT
25: LOAD 1
```

```
26: LOAD 3
27: ADD
28: STORE 1
29: LOAD 0
30: PUSH 1
31: SUB
32: STORE 0
33: GOTO 4
34: HALT
```

```
> run
```

Comienza la ejecución de RUN

consola: 120

consola: 24

consola: 6

consola: 2

consola: 1

El estado de la maquina tras ejecutar programa es:

Estado de la CPU:

Memoria: [0]:0 [1]:153 [2]:0 [3]:1

Pila: <vacía>

Programa fuente almacenado:

```
0: x = 5
1: r = 0
2: while 0 < x
3:   y = x
4:   s = 1
5:   while 0 < y
6:     s = s * y
7:     y = y - 1
8:   endwhile
9:   write s
10:  r = r + s
11:  x = x - 1
12: endwhile
13: return
14: end
```

Programa bytecode almacenado:

```
0: PUSH 5
1: STORE 0
2: PUSH 0
3: STORE 1
4: PUSH 0
5: LOAD 0
6: IFLE 34
7: LOAD 0
```



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

```
> load noexiste.txt
```

Comienza la ejecución de LOAD noexiste.txt

Excepcion: Fichero no Encontrado...

```
> load twoif.txt
```

Comienza la ejecución de LOAD twoif.txt

Programa fuente almacenado:

```
0: x = 10
1: if x = 10
2:   y = 7
3: endif
4: x = 3
5: if x != 10
6:   y = 5
7: endif
8: end
```

```
> compile
```

Comienza la ejecución de COMPILE

Programa fuente almacenado:

```
0: x = 10
1: if x = 10
```

```
2:   y = 7
3: endif
4: x = 3
5: if x != 10
6:   y = 5
7: endif
8: end
```

Programa bytecode almacenado:

```
0: PUSH 10
1: STORE 0
2: LOAD 0
3: PUSH 10
4: IFEQ 7
5: PUSH 7
6: STORE 1
7: PUSH 3
8: STORE 0
9: LOAD 0
10: PUSH 10
11: IFNEQ 14
12: PUSH 5
13: STORE 1
```

> run

Comienza la ejecución de RUN

El estado de la maquina tras ejecutar programa es:

Estado de la CPU:

Memoria: [0]:3 [1]:5

Pila: <vacía>

Programa fuente almacenado:

```
0: x = 10
1: if x = 10
2:   y = 7
3: endif
4: x = 3
5: if x != 10
6:   y = 5
7: endif
8: end
```

Programa bytecode almacenado:

```
0: PUSH 10
1: STORE 0
2: LOAD 0
3: PUSH 10
4: IFEQ 7
```

```
5: PUSH 7
6: STORE 1
7: PUSH 3
8: STORE 0
9: LOAD 0
10: PUSH 10
11: IFNEQ 14
12: PUSH 5
13: STORE 1
```

>

6. 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 contenido¹:

- 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.

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