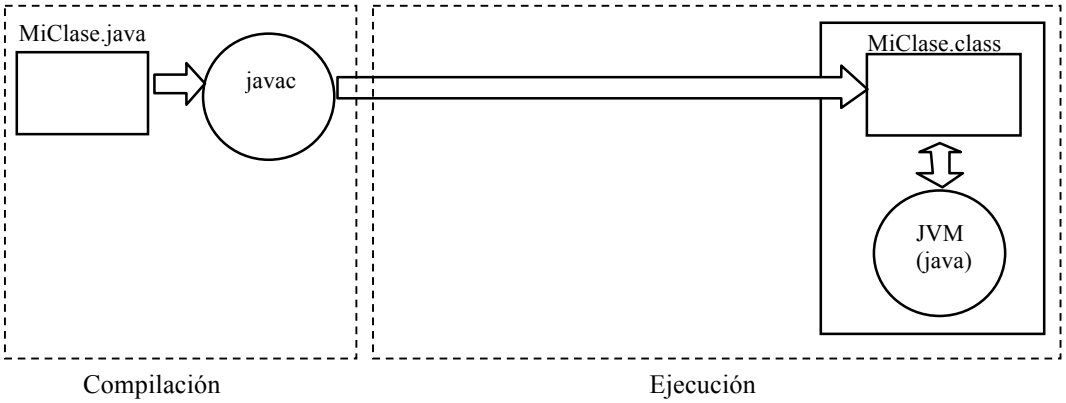
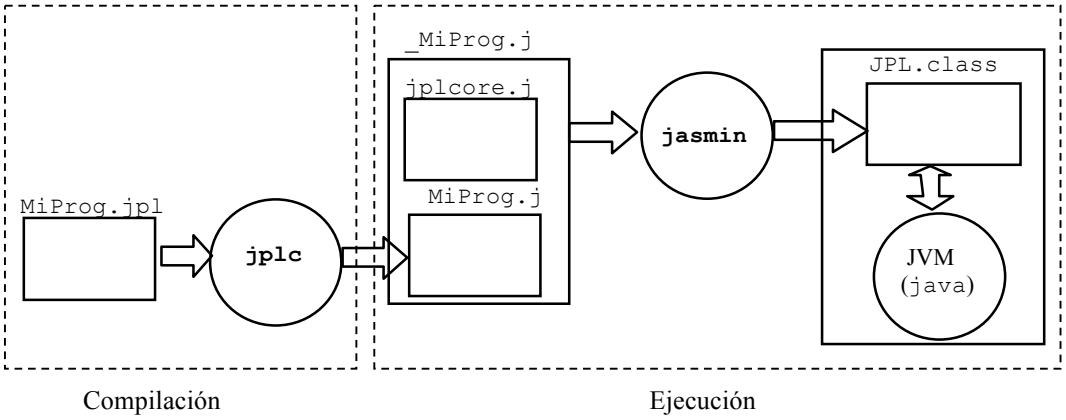
INTRODUCCION

La máquina virtual de Java (JVM) es una máquina abstracta (implementada por el programa `java`) que interpreta el lenguaje intermedio generado por el compilador de Java (`javac`). El código de dicho lenguaje se encuentra en los ficheros `.class`, El código de la JVM es una secuencia de números que resulta difícil de leer, por lo que se ha definido un lenguaje ensamblador que hace más simple tanto la lectura como la escritura de programas en el lenguaje intermedio de la JVM.

El esquema normal de compilación de Java es el siguiente:



Se desea definir un nuevo lenguaje (al que llamaremos JavaPL), mucho más simple, pero que sea compatible con Java, que se va a compilar utilizando algunas herramientas auxiliares, siguiendo el esquema:



Es decir, el código fuente de `MiProg.jpl` se traduce a código ensamblador de JAVA generando el fichero `MiProg.j` al que se añade un cierto código fijo (que está en el fichero `jplcore.j`). La unión de ambos trozos de código ensamblador se guardan en el fichero temporal `_MiProg.j`. Utilizando el programa `jasmin` se convierte este fichero en un fichero `JPL.class`, que puede ser directamente ejecutado por la máquina virtual de Java.

SE PIDE: implementar con Java, JFlex y Cup el compilador del lenguaje fuente JavaPL al código ensamblador. Para ello, será necesario (al menos) implementar los ficheros `JPLC.java`, `JPLC.flex` y `JPLC.cup`, que una vez compilados darán lugar al programa JPLC.class. Es decir, construir el compilador que antes hemos denominado `jplc`, (equivalente a "`java JPLC`"), según las instrucciones indicadas a continuación.

En páginas siguientes se describen los lenguajes fuente (JavaPL) y objeto (ensamblador de Java reducido).

El lenguaje JavaPL (código fuente)

El lenguaje fuente tiene una sintaxis similar a C y consiste en un conjunto de funciones de un solo argumento, de tipo entero, que devuelven siempre un entero. El lenguaje no admite variables globales, sólo variables locales de tipo entero. El lenguaje puede contener expresiones aritméticas simples (suma, resta, multiplicación y división); expresiones de asignación; llamadas a otras funciones y expresiones relacionales (menor, mayor, igual, etc.); así como sentencias de control ***if-else*** y ***while***; y la sentencia ***return***. Todo programa en JavaPL tiene una función principal denominada `main`, que también tiene como entrada un entero y devuelve otro entero, y que es siempre la primera en ejecutarse. Por ejemplo, el siguiente es un programa válido en JavaPL:

MiProg.jpl

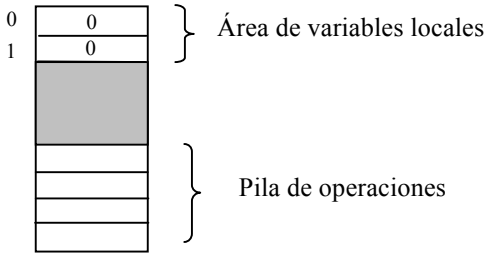
```
int g(int x) {
    int i = 1 ;
    int factorial = 1;
    while (i<x+1) {
         factorial = factorial * i ;
         i = i+1;
    }
    return factorial;
}

int f(int n) {
    if (n<1) {
          return 1;
    } else {
          return n*f(n-1);
    }
}

int main(int x) {
    int y,z ;
    y = f(x) ;
    z = g(x) ;
    return (z+y)/2 ;
}
```

EL CÓDIGO OBJETO (Código ensamblador de Java):

El código objeto es un subconjunto del código ensamblador de la máquina virtual de Java. La JVM crea un registro de activación para cada función, Para los fines de este ejercicio, los registros de activación (*F*) constan únicamente de dos secciones: "El área de variables locales" (*LV*) y "La pila de operaciones", (*SO*). El área de variables locales da soporte al almacenamiento en memoria tanto de los parámetros de la función, como de las variables locales. Se compone de n_L posiciones, cada una de las cuales es capaz de almacenar un entero, y que se numeran empezando por 0. Los parámetros de la función se ubican en las primeras posiciones (comenzando por 0). La pila de operaciones es un área de memoria de tamaño n_S posiciones cada una de las cuales puede contener un número entero, inicialmente vacía, que sirve para realizar las instrucciones de código objeto en tiempo de ejecución. El siguiente esquema representa un registro de activación para un método que tiene un parámetro formal y una variable local $n_L = 2$; y usa una pila de operaciones de tamaño 4 $n_S = 4$

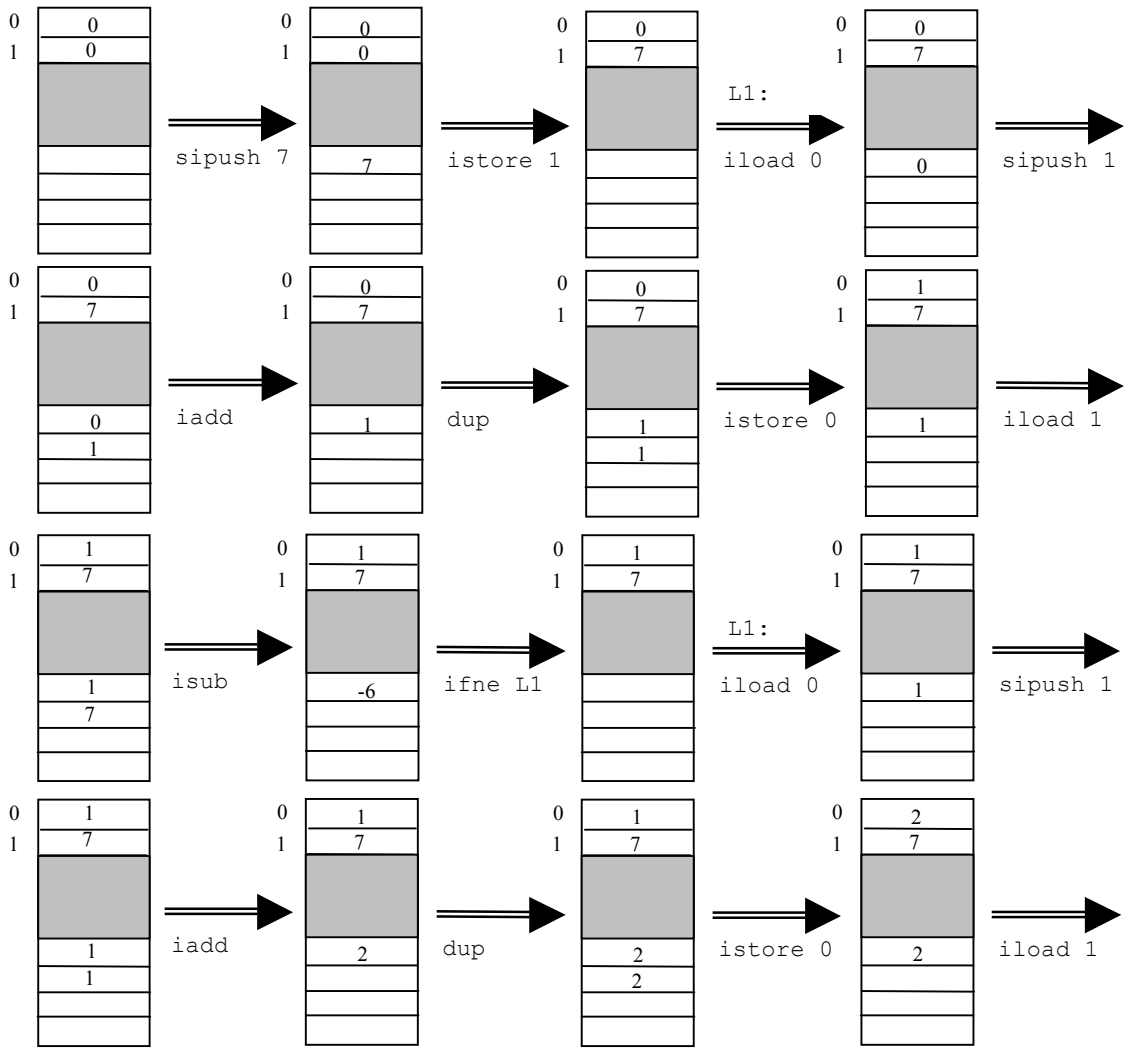


El conjunto de instrucciones del código ensamblador, y su semántica son las siguientes:

Instrucción	Acción
`.method public static <f> (I)I`	Declara el comienzo de la definición del método <*f*>
`sipush <n>`	Mete una constante entera <*n*> en la pila.
`iload <n>`	Carga la variable situada en la posición <*n*> del área de variables locales, en la cima de la pila.
`istore <n>`	Almacena en la posición <*n*> del área de variables locales, el valor situado en la cima de la pila.
`iadd`	Suma los dos valores situados en la cima de la pila, sustituyendo ambos valores por el resultado.
`isub`	Resta el valor situado en la cima de la pila del valor situado justo antes, reemplazando ambos por el resultado.
`imul`	Suma los dos valores situados en la cima de la pila, sustituyendo ambos valores por el resultado.
`idiv`	Divide el valor situado debajo de la cima de la pila por el valor de la cima, reemplazando ambos por el resultado.
`pop`	Saca un elemento de la pila.
`dup`	Copia el elemento de la cima de la pila, y lo coloca encima, de manera que los dos elementos superiores de la pila son iguales
`nop`	No hace nada
<*L*>`:`	Sirve para marcar un punto de destino de una etiqueta de salto. <*L*> es una cadena de texto
`goto <L>`	Salta a la siguiente instrucción tras la etiqueta <*L*>:
`ifeq <L>`	Lee y saca de la pila el número situado en la cima, y si es igual a cero salta a la instrucción <*L*>:
`ifne <L>`	Lee y saca de la pila el número situado en la cima, y si es distinto de cero salta a la instrucción <*L*>:
`ifge <L>`	Lee y saca de la pila el número situado en la cima y si es igual o mayor que cero, salta a la instrucción <*L*>:
`ifle <L>`	Lee y saca de la pila el número situado en la cima, y si es igual o menor que cero salta a la instrucción <*L*>:
`invokestatic JPL/`<*f*>`(I)I`	Llama a la función <*f*>, pasándole como parámetro el valor situado en la cima de la pila, el cual saca.
`ireturn`	Termina la ejecución de la función y sitúa en la cima de la pila de la función que realizó la llamada, el valor devuelto.
`.limit stack <` n_S `>`	Define el tamaño máximo de la pila de operaciones
`.limit locals <` n_L `>`	Define el tamaño máximo del área de variables locales.
`.end method`	Declara el final de la definición del método <*f*>

EJEMPLO: Al ejecutar el siguiente trozo de código JVM se realiza la secuencia de instrucciones que se muestra en los esquemas que hay a continuación

```
.method public static main(I)I
   sipush 7
   istore 1
L1:
   iload 0
   sipush 1
   iadd
   dup
   istore 0
   iload 1
   isub
   ifne L1
   sipush 1
    …
   .limit stack 3
   .limit locals 2
.end method
```



IMPLEMENTACIÓN DE LA PRÁCTICA:

Para realizar la implementación de este ejercicio se puede utilizar cualquier tipo de apuntes o código anteriormente elaborado por el alumno durante las prácticas.

Se proporciona una solución compilada del ejercicio (versiones para Linux, Windows y Mac). Esto puede servir de ayuda para comprobar los casos de prueba y las instrucciones del código JVM,. No es necesario que el código generado sea idéntico al que se propone como ejemplo (que de hecho no es óptimo), basta con que sea equivalente, es decir que dé los mismos resultados al ejecutar los casos de prueba. Para compilar y ejecutar un programa en JavaPL, deben utilizarse las instrucciones

	Linux
Compilación	`./jplc MiProg.jpl MiProg.j`
Ejecución	`./jpl MiProg 12`

En donde `MiProg.jpl` contiene el código fuente en JavaPL, `MiProg.j` contiene el código ensamblador de la JVM, y `12` (un número entero cualquiera) es el argumento que se pasará inicialmente a la función `main` de `MiProg`. El programa `jplc` es un *script* del *shell* del sistema operativo que llama a (`java JPLC`), que es el programa que se pide construir en este ejercicio. El programa `jpl` es otro *script* del *shell* que se encarga de realizar las llamadas al ensamblador (`jasmin`) y realizar la ejecución del programa ensamblado (`java JPL`).

EVALUACIÓN DE ESTE EJERCICIO:

Para la corrección de este ejercicio, se tendrán en cuenta resultados parciales según el número de casos de prueba que supere el compilador, de acuerdo a los siguientes criterios:

a) Funciones que contengan solamente expresiones con números enteros. Se deben implementar los operadores aritméticos +,-,*,/, operador menos unario, y paréntesis. (6 puntos)
b) Funciones que contengan solamente expresiones y asignaciones con números enteros y/o variables locales. Debe comprobarse que las variables han sido declaradas antes de usarlas, generando un mensaje de error al compilar. (4 puntos)
c) Declaración de variables con inicialización. Si una variable no se inicializa, se considera que su valor es cero. (4 puntos)
d) Llamada a otras funciones. (4 puntos)
e) Lenguaje con estructura de bloques (Sentencias compuestas) (2 puntos)
f) Sentencia `if-else`. La parte "*else*" es opcional, al igual que en Java. Se deben implementar los seis operadores relacionales <=, <, ==, !=, >, >= (4 puntos)
g) Sentencia `while` (4 puntos)
h) Cálculo exacto del límite máximo de la pila de operaciones y del número de variables locales. (2 puntos)

Se proporcionan dos casos de prueba para cada apartado (excepto el último).

NOTAS IMPORTANTES:

1. Toda práctica debe contener al menos tres ficheros denominados "`JPLC.java`", "`JPLC.flex`" y "`JPLC.cup`", correspondientes respectivamente al programa principal y a las especificaciones en Jflex y Cup.. Para realizar la compilación se utilizarán las siguientes instrucciones:

```
cup JPLC.cup
jflex JPLC.flex
javac *.java
```

2. Para que la compilación de las cinco primeras fases iniciales no provoque problemas de ejecución por los tamaños del área de variables locales o la pila de operaciones se recomienda considerar fijos estos tamaños, es decir, incluir simplemente las instrucciones

```
.limit stack 10
.limit locals 10
```