Estrategias de Programación y Estructuras de Datos

Grado en Ingeniería Informática

Grado en Tecnologías de la Información

Práctica curso 2018-2019

Enunciado

Índice

1. Presentación del problema

Al utilizar números enteros estamos limitados por el tamaño de los mismos. En Java podemos utilizar cuatro tipos de enteros:

- byte: de -128 a +127 (8 bits)
- short: de -32768 a +32767 (16 bits)
- int: de -2147483648 a +2147483647 (32 bits)
- long: de -9223372036854775808 a +9223372036854775807 (64 bits)

Esto significa que si queremos operar con números del orden de los diez trillones o más, al no poderse almacenar en 63 bits (dejando un bit para el signo), no podemos utilizar los enteros que nos ofrece Java.

El objetivo de esta práctica es crear una máquina pila capaz de evaluar expresiones formadas por sumas, restas y multiplicaciones de enteros de tamaño ilimitado. Dicha máquina recibirá el árbol sintáctico de la expresión y deberá devolver un valor entero con el resultado. Por ejemplo, para la expresión:

```
4 + (5 * (2 - (-1)))
```

el resultado es 19.

El proceso de construcción del árbol sintáctico queda fuera de esta asignatura, por lo que el Equipo Docente proporcionará todas las clases necesarias para realizar este trabajo.

Dicha máquina trabajará con operandos enteros, de los cuales utilizaremos dos implementaciones diferentes:

1. Una implementación que empleará el tipo `int` de Java.
2. Una implementación que empleará una secuencia de dígitos para almacenar enteros de precisión ilimitada (salvo por la propia memoria del ordenador).

Así pues, la tarea de cada estudiante será la siguiente:

1. Programar una clase con la máquina pila (cuyo funcionamiento se detalla más adelante en este enunciado), la cual podrá probarse inicialmente con la implementación de enteros que usa el tipo `int`.
2. Programar una clase que implemente las operaciones de enteros sobre secuencias de dígitos, lo cual permitirá que la misma máquina pila opere con enteros de precisión ilimitada.

La única restricción que se plantea es que ***no se permite el uso de iteradores***, por lo que las secuencias de dígitos que representan un entero deberán ser recorridas preservando su estructura de forma adecuada.

2. Primera parte: máquina pila.

En este apartado vamos a ver el diseño de clases de la práctica necesario para implementar la máquina pila. Explicaremos el funcionamiento de las clases ya programadas por el Equipo Docente e indicaremos cuáles habrán de programar los estudiantes.

2.1 Árbol Sintáctico: clase SynTree (proporcionada por el ED).

De la creación del árbol sintáctico de las expresiones se encarga la clase **`SynTree`**, cuyo constructor recibe una cadena de caracteres conteniendo la expresión en notación prefija[1], es decir, con los operadores delante de los operandos, por ejemplo:

Notación infija: `4 + (5 * (2 - (-1)))`
Notación prefija: `+ 4 * 5 - 2 -1`

La ventaja de utilizar esta notación es que no se necesitan paréntesis, ya que no cabe ambigüedad posible sobre la precedencia de los operadores como en la notación infija. Recordemos que la conversión de notación infija a prefija queda fuera del ámbito de la asignatura.

El constructor delega la creación del árbol sintáctico al método recursivo **`createSynTree()`**. Este método va leyendo los diferente operadores y operandos de la expresión y crea los nodos del árbol de forma que represente de manera adecuada la expresión leída. En nuestro caso, el árbol construido sería el siguiente:

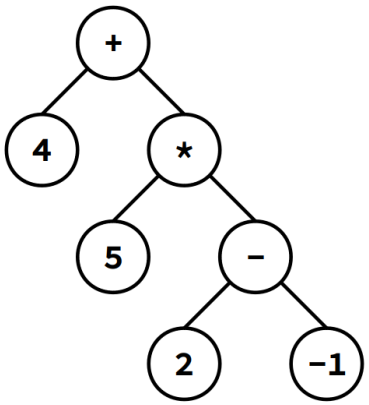

*Figura 1: Árbol sintáctico de la expresión 4 + (5 * (2 - (-1)))*

Dado que todos los operadores que vamos a considerar (suma, resta y producto) son binarios, el árbol sintáctico se representa mediante una estructura de árbol binario en el que los nodos intermedios van a contener operadores y las hojas operandos (como se puede ver en la Figura 1).

Nótese que un recorrido en preorden de este árbol binario nos daría la notación prefija de la expresión, mientras que un recorrido en inorden nos daría la notación infija de la expresión.

Para poder tener nodos de diferente tipo, necesitamos crear una clase abstracta **`Node`** que nos permita agrupar los dos tipos de nodos bajo una misma clase, de manera que el árbol binario que utilizamos como estructura sea un árbol binario de objetos de la clase **`Node`**.

2.1.1 Nodos: clase abstracta Node (proporcionada por el ED).

La clase **`Node`** es la encargada de agrupar los dos tipos de nodos que vamos a considerar en el árbol binario donde representaremos nuestro árbol sintáctico.

1 Para escribir una expresión, podemos utilizar varias notaciones. Normalmente utilizamos la infija, en la que los operadores binarios (suma, resta…) están entre sus dos operandos. La notación prefija establece que los operadores deben aparecer delante de sus dos operandos y la notación infija dicta que los operadores deben aparecer detrás de sus dos operandos.

Esta clase es abstracta, lo que nos impide crear un objeto de la clase **`Node`** directamente. Pero nos define un tipo enumerado indicando qué tipos de nodos consideramos (**`OPERATOR`** y **`OPERAND`**) y describe un getter **`getNodeType()`** que devuelva el tipo de nodo para todas las clases que hereden de ella.

2.1.2 Operadores: clase Operator (proporcionada por el ED).

Los operadores están definidos en la clase **`Operator`**, que extiende a (es decir, hereda de) **`Node`**. Define un tipo enumerado con los tres diferentes operadores que vamos a considerar (**`ADD`**, **`SUB`** y **`MULT`**) y un constructor que recibe como parámetro el tipo de operador que representa.

El getter **`getNodeType()`** descrito por la clase padre devuelve directamente que se trata de un nodo **`OPERATOR`** y además añade un nuevo getter **`getOperatorType()`** que devuelve el tipo de operador que representa.

2.1.3 Operandos: clase Operand (proporcionada por el ED).

En nuestro árbol sintáctico el otro tipo de nodos son los operandos, los cuales se implementan en la clase **`Operand`**. Esta clase extiende (al igual que la anterior) la clase **`Node`**, por lo que implementa el getter **`getNodeType()`**, el cuál devuelve que se trata de un nodo **`OPERAND`**.

Definiremos un operando como un valor entero (que se almacenará mediante un objeto de la clase **`Value`**, la cual veremos más adelante) y un signo (almacenado mediante un **`byte`**, que puede ser -1 para enteros negativos, +1 para enteros positivos y 0 para el 0). Ambos atributos tienen su getter correspondiente: **`getVal()`** y **`getSig()`** respectivamente.

El constructor de esta clase recibe una cadena de caracteres con el texto del operando y se encarga de detectar el signo del operando y su valor, delegando esta última operación en un método de la clase **`Value`**. También implementa un método que transforma el operando en una cadena de caracteres para su correcta visualización: **`toString()`**.

Por último, esta clase implementa tres operaciones modificadoras, que se corresponden con los tres tipos de operadores que estamos considerando:

1. **`add(Operand n)`**: este método modifica el operando llamante sumándole el operando parámetro.

   En primer lugar comprueba si los signos de ambos operandos coinciden, en cuyo caso el signo del resultado es el mismo y el valor es la suma de ambos valores. Por ejemplo:

   - 3 + 5 = 8, con ambos operandos positivos.
   - (-3) + (-5) = (-8), con ambos operandos negativos.

   En caso contrario, comprueba cuál de los dos valores (ya sin signo) es el mayor (estrictamente) y le resta el menor. En cuanto al signo:

   - Si el nuevo valor es 0, será 0.
   - En cualquier otro caso, será el signo del mayor de ambos valores.

2. **`sub(Operand n)`**: este método modifica el operando llamante restándole el operando parámetro.

En primer lugar comprueba si los signos de ambos operandos son contrarios (es decir, uno positivo y otro negativo), en cuyo caso el signo del resultado es el mismo que el del operando llamante y el valor será la suma de ambos valores. Por ejemplo:

- 3 - (-5) = 8, positivo menos negativo.
- (-3) - 5 = -8, negativo menos positivo.

En caso contrario, comprueba cuál de los dos valores (ya sin signo) es el mayor (estrictamente) y le resta el menor. En cuanto al signo:

- Si el nuevo valor es 0, será 0.
- Si el valor del operando llamante es el mayor, el signo del resultado será el signo del operando llamante: (-4) - (-3) = -1

- Si el valor del operando parámetro es el mayor, el signo del resultado será el contrario al signo del operando parámetro: (-3) - (-4) = 1

3. **`mult(Operand n)`**: este método modifica el operando llamante multiplicándolo por el operando parámetro.

   En primer lugar calculamos el signo del resultado multiplicando los signos de ambos operandos. Si resultase ser 0 significa que alguno de los operandos es 0, por lo que sabemos directamente que el valor del resultado será 0, por lo que no realizamos ningún tipo de operación.

   En caso contrario, el valor del resultado será el producto de ambos valores.

Nótese que estos métodos trabajan sobre operandos y delegan las operaciones sobre los **valores** en métodos de la clase **`Value`**, por lo que son válidos para cualquier representación de valores que utilicemos. En esta práctica utilizaremos dos representaciones: el tipo **`int`** de Java y secuencias de dígitos; pero se podría utilizar cualquier otra.

2.2 Máquina Pila: clase StackMachine (trabajo del estudiante).

Una máquina pila es una máquina virtual que permite evaluar expresiones formadas por operandos y operadores mediante el uso de una pila de operandos. Para ello, debe recibir la expresión a ser evaluada en notación postfija, es decir, con los operandos delante de los operadores. Por ejemplo, para la expresión de la Figura 1:

Notación infija: `4 + (5 * (2 - (-1)))`
Notación prefija: `+ 4 * 5 - 2 -1`
Notación postfija: `4 5 2 -1 - * +`

Al igual que la notación prefija, la notación postfija no requiere el uso de paréntesis para desambiguar la precedencia de los operadores. Se obtiene mediante un recorrido en postorden del árbol sintáctico.

Dado que la clase **`SynTree`** nos transforma una cadena conteniendo la expresión a evaluar (en notación prefija) en un árbol sintáctico, será este árbol el que pasaremos al constructor de la máquina pila, que deberá recorrerlo **en postorden** (**nota: recuérdese que no pueden utilizarse iteradores**) y realizar las operaciones necesarias para evaluar la expresión.

2.2.1 Funcionamiento de la máquina

Una máquina pila evalúa una expresión mediante una pila de operandos que inicialmente está vacía y se va modificando según se avanza en el recorrido de la notación postfija de la expresión. Así, nuestra máquina va a recibir un árbol sintáctico que deberá ser recorrido en postorden y cada nodo será procesado de la siguiente manera:

- Si se trata de un nodo operando, dicho operando se apila en la pila de operandos.
- Si se trata de un nodo operador (y ya que todos los operadores que consideramos son binarios), la máquina extrae dos operandos de la pila de operandos, calcula el resultado del operador sobre ambos operandos (***atención al orden***) y apila dicho resultado en la pila. Cuidado: para que el método funcione sobre cualquier representación de enteros, es necesario que las operaciones se realicen mediante los métodos **`add`**, **`sub`** y **`mult`** de la clase **`Operand`** explicados anteriormente.
- Al terminar de evaluar la expresión, en la cima de la pila queda el resultado de la misma.

Vamos a ver cómo la máquina evaluaría la expresión 4 + (5 * (2 - (-1))), cuya notación postfija es:

```
4 5 2 (-1) - * +
```

1. **Nodo operando 4**: se apila el 4 en la pila de operandos:

   pila → 4

2. **Nodo Operando 5**: se apila el 5 en la pila de operandos:

   pila → 5 , 4

3. **Nodo Operando 2**: se apila el 2 en la pila de operandos:

   pila → 2 , 5 , 4

4. **Nodo Operando (-1)**: se apila el (-1) en la pila de operandos:

   pila → (-1) , 2 , 5 , 4

5. **Nodo Operador – (resta)**: desapilamos dos operandos de la pila (-1) y 2. Dado que el 2 se apiló antes, este será el primer operando y el (-1) el segundo. Ahora calculamos 2 - (-1) = 3 y apilamos el 3 en la pila de operandos.

   pila → 3 , 5 , 4

6. **Nodo Operador * (multiplicación)**: desapilamos dos operandos de la pila 3 y 5. Dado que el 5 se apiló antes, éste será el primer operando y el 3 el segundo. Ahora calculamos 5 * 3 = 15 y apilamos 15 en la pila de operandos.

   pila → 15 , 4

7. **Nodo Operador + (suma)**: desapilamos dos operandos de la pila 15 y 4. Dado que el 4 se apiló antes, éste será el primer operando y el 15 el segundo. Ahora calculamos 4 + 15 = 19 y apilamos 19 en la pila de operandos.

pila → 19

8. Hemos terminado de evaluar la expresión en notación postfija. En la cima de la pila tenemos el resultado de dicha evaluación: 19.

2.2.2 Descripción de los métodos que se han de implementar

La clase **`StackMachine`** deberá ser implementada por los estudiantes, aunque el Equipo Docente proporciona una plantilla que deberá ser modificada para que los métodos funcionen adecuadamente. Deberá incluir los siguientes métodos públicos (pudiendo incluir cuantos métodos privados fuesen necesarios):

- Constructor: el constructor de la clase no recibirá ningún parámetro, pero se encargará de inicializar adecuadamente los atributos que se necesiten:

```
public StackMachine() {…}
```

- Método **`execute`**: este método recibe un árbol sintáctico, lo recorre en postorden, evalúa la expresión siguiendo el método explicado en el punto anterior y devuelve el resultado de dicha evaluación:

```
public Operand execute(SynTree syn) {…}
```

Una vez implementada esta clase, antes de pasar a la siguiente se puede comprobar su funcionamiento con operandos que utilicen valores representados mediante el tipo **`int`** de Java, ya que se proporciona una implementación completa de estos valores.

Para ello se deberá ejecutar la práctica indicándole mediante el primer parámetro que utilice el tipo **`int`**:

```
java –jar eped2019.jar INT <expresiones>
```

y proporcionarle un fichero de expresiones cuyos enteros sean representables mediante dicho tipo.

3. Segunda parte: Secuencias de Dígitos.

En la segunda parte de la práctica vamos a realizar una implementación de números enteros mediante secuencias de dígitos. Dado que estas secuencias no estarán limitadas, podremos operar con valores arbitrariamente grandes, sobrepasando sin problemas los valores máximos de las representaciones de enteros proporcionadas por Java.

3.1 Clase abstracta Value (proporcionada por el ED).

Los objetos de la clase **`Value`** representan valores enteros sin signo. Como vamos a utilizar dos representaciones diferentes (mediante el tipo **`int`** y mediante secuencias de dígitos), esta clase va a ser abstracta, por lo que no podremos crear ningún objeto de ella.

Esta clase define un tipo enumerado **`ValueClass`** que indica qué clase de valores vamos a utilizar: **`INT`** para valores representados mediante el tipo **`int`** y **`SEQ`** para valores representados mediante secuencias de dígitos.

Además, esta clase tiene un atributo y un método estáticos (lo que significa que son accesibles sin necesidad de que exista un objeto de la clase) cuyo funcionamiento detallamos a continuación:

- Atributo **`valueClass`**: este atributo de tipo **`ValueClass`** va a hacer el papel de una variable global que se inicializa en el programa principal. Servirá para almacenar la clase de valores que vamos a tener que utilizar.
- Método **`construct`**: este método delegará en el constructor de la clase adecuada la construcción del valor[2]. Así, si **`valueClass == ValueClass.INT`** significará que estamos utilizando valores representados mediante el tipo **`int`** de Java, por lo que se llamará al constructor de la clase **`ValueInt`**. En caso contrario se llamaría al constructor de la clase **`ValueSeq`**. Este método es llamado desde el constructor de la clase **`Operand`** para construir el valor del nuevo operando (y también desde el método **`mult`** de la misma clase).

De esta forma, utilizando esta clase encapsulamos la representación interna de los valores, de manera que sólo el programa principal (encargado de inicializar el valor del atributo **`valueClass`**) y esta clase saben qué tipo de representación estamos utilizando. El resto del programa es agnóstico con respecto al tipo de valores que consideremos.

Por último, esta clase describe una serie de métodos abstractos que deberán ser implementados por toda clase (no abstracta) que la extienda, las cuales veremos a continuación: **`ValueInt`** y **`ValueSeq`**. Hablaremos de ellos más adelante.

3.2 Clase ValueInt (proporcionada por el ED).

Los objetos de la clase **`ValueInt`** son valores enteros (sin signo) representados mediante el tipo **`int`** de Java. Esta clase extiende la clase **`Value`**, por lo que todo objeto **`ValueInt`** es, a su vez, un objeto **`Value`**.

El constructor de esta clase recibe una cadena de caracteres con un número sin signo, es decir, una cadena de caracteres en la que cada carácter es un dígito. El cometido del constructor es analizar la cadena y convertirla en un valor adecuado del tipo **`int`**, el cual será almacenado en el atributo entero **`value`**. El proceso inverso lo realiza el método **`toString()`**, que devuelve una cadena de caracteres que representa el valor numérico almacenado en el objeto.

Por último, se implementan todos los métodos necesarios para realizar los cálculos entre valores, los cuales son llamados desde los métodos de la clase **`Operand`**. Aquí hay que hacer notar que en aquellos métodos que reciben un parámetro de tipo **`Value`**, a dicho parámetro se le aplica un casting explícito. Vamos a ver un ejemplo concreto para entender por qué:

```
public void addValue(Value n) {
     this.value += ((ValueInt) n).value;
}
```

El cometido del método **`addValue`** es modificar el valor del objeto llamante sumándole el valor del objeto parámetro. Este método está implementado dentro de **`ValueInt`**, por lo que los objetos parámetro además de ser **`Value`** van a ser **`ValueInt`**. En esta situación es necesario hacer el casting:

```
((ValueInt) n)
```

para decirle a Java que considere que **`n`** es un objeto **`ValueInt`**, que es más específico que **`Value`**. De esta forma podemos acceder al atributo **`value`** que es el que contiene el valor del número. Como puede verse en la implementación, este casting se debe hacer siempre que intentemos acceder

2 Al tratarse de una clase abstracta no puede existir un constructor. Por eso usamos un método para llamar al constructor de la clase adecuada.

al atributo **value** de un objeto recibido como parámetro.

3.3 Clase ValueSeq (trabajo del estudiante).

Los objetos de la clase **ValueSeq** serán valores enteros (sin signo) representados mediante secuencias de dígitos. Al igual que la anterior, esta clase extiende la clase **Value**, por lo que todo objeto **ValueSeq** será, a su vez, un objeto **Value**.

Esta clase deberá ser implementada por los estudiantes, aunque el Equipo Docente proporciona una plantilla que deberá ser modificada de acuerdo con las instrucciones del siguiente apartado.

3.3.1 Descripción de los métodos que se han de implementar en ValueSeq

En este apartado vamos a describir el funcionamiento de los métodos que se han de implementar en la clase **ValueSeq**. Para ello será útil comparar su funcionamiento con el de los métodos equivalentes ya implementados en la clase **ValueInt**.

- Constructor: recibe una cadena de caracteres que representa el valor y debe crear la secuencia de dígitos que lo represente, la cual se almacenará en un atributo de esta clase.
- Método **String toString()**: debe devolver una cadena de caracteres que represente el valor almacenado en la secuencia de dígitos.
- Método **void addValue(Value n)**: recibe un valor **n** (al que se le deberá aplicar el casting a **ValueSeq**, al igual que en la clase **ValueInt**) y modifica el valor del objeto llamante sumándole el valor del objeto **n**.
- Método **void subValue(Value n)**: recibe un valor **n** y modifica el valor del objeto llamante restándole el valor del objeto **n**.
- Método **void subFromValue(Value n)**: recibe un valor **n** y modifica el valor del objeto llamante restando su valor al valor del objeto **n**. Parece igual al anterior pero no lo es, ya que la resta no posee la propiedad conmutativa.
- Método **void multValue(Value n)**: recibe un valor **n** y modifica el valor del objeto llamante multiplicándolo por el valor del objeto **n**.
- Método **boolean greater(Value n)**: recibe un valor **n** y devuelve un valor booleano que indica si el valor del objeto llamante es estrictamente mayor que el valor del objeto **n**.
- Método **boolean isZero()**: devuelve un valor booleano que indica si el valor del objeto llamante representa al número 0.

Para la implementación de los métodos de suma, resta y multiplicación, será necesario implementar los algoritmos adecuados que sumen, resten y multipliquen respectivamente dos secuencias de dígitos.

Una vez estén implementados todos los métodos, la máquina pila podrá operar con enteros representados mediante secuencias de dígitos, salvando las limitaciones de los tipos predefinidos en Java.

4. Cuestiones teóricas (trabajo del estudiante).

A continuación se enuncian una serie de preguntas sobre la práctica, cuya respuesta implica la aplicación de los conocimientos teóricos de la asignatura:

1. Explique y justifique la elección de la estructura de datos que ha utilizado para operar con las secuencias de dígitos de la clase **ValueSeq**. ¿Qué otras alternativas hay y por qué las ha descartado?

2. Calcule el coste asintótico temporal en el caso peor de la implementación que ha realizado de los métodos **`addValue`** y **`subValue`** en la clase **`ValueSeq`**.
3. Calcule el coste asintótico temporal en el caso peor de la implementación que ha realizado del método **`multValue`** en la clase **`ValueSeq`**.
4. Supongamos que la implementación de **`multValue`** en la clase **`ValueSeq`** se realiza mediante sumas sucesivas. Esto es, para multiplicar, por ejemplo, 10 por 283, se sumaría 10 + 10 + … + 10 (un total de 283 veces). Calcule el coste asintótico temporal en el caso peor de esta variante de implementación de **`multValue`** en la clase **`ValueSeq`**.

5. Implementación.

Se deberá realizar un programa en Java llamado **`eped2019.jar`** que contenga las siguientes clases programadas por el estudiante:

- La clase **`StackMachine`**, que implemente la máquina pila que evalúa los árboles sintácticos con operaciones sobre enteros.
- La clase **`ValueSeq`**, que implementa los enteros representados mediante una secuencia de dígitos y las operaciones que se pueden realizar sobre ellos.

Todas ellas se implementarán en un único paquete llamado:

`es.uned.lsi.eped.pract2018_2019`

Para la implementación de las estructuras de datos de soporte **se deberá utilizar las interfaces y las implementaciones proporcionadas por el Equipo Docente** de la asignatura. Recuérdese que **no se permite el uso de iteradores, por lo que si se detecta su uso la práctica estaría automáticamente suspensa**.

El programa deberá contener, además, todas las clases necesarias para:

- Identificar los *parámetros de entrada* pasados al programa.
- Leer un *fichero de expresiones* que contendrá una lista de las expresiones a evaluar.

El Equipo Docente proporcionará a través del Curso Virtual las clases que realizan todas estas operaciones, por lo que los estudiantes sólo deberán programar las dos clases anteriormente mencionadas.

5.1 Parámetros de entrada.

El programa recibirá dos parámetros de entrada que determinarán su comportamiento. El orden y significado de los parámetros será el siguiente:

1. Selección de la estructura de datos que se desea utilizar para la representación de los valores, que tendrá únicamente dos valores válidos (ya sean en mayúsculas o en minúsculas):
    - INT para utilizar la implementación que emplea el tipo int de Java.
    - SEQ para utilizar la implementación que emplea secuencias de dígitos.
2. Fichero de expresiones, que contendrá el nombre del fichero de expresiones que se desean evaluar.

5.2 Formato del fichero de expresiones.

El fichero de expresiones contendrá las expresiones sobre enteros que queremos evaluar. Su estructura es la siguiente:

- Las líneas en blanco serán ignoradas.
- Si la línea comienza por el carácter “#” se considerará un comentario y se copiará directamente en la salida.
- En caso contrario, se asumirá que la línea contiene una expresión en notación prefija (es decir, los operadores delante de los operandos) y que ésta es, además, correcta. El programa asume que todas las expresiones son correctas y no realiza ningún tipo de comprobación sobre la corrección de las mismas. Por eso, si se introdujese una expresión mal formada, podrían aparecer errores ajenos al programa. El formato de la expresión será el siguiente:
  - Los operadores estarán representados por los caracteres ‘**+**’ (suma), ‘**-**’ (resta) y ‘*****’ (producto).
  - Los operandos deberán estar formados exclusivamente por dígitos del **0** al **9**, sin ceros a la izquierda y, opcionalmente, un carácter ‘**-**’ al comienzo para indicar que el entero es negativo.
  - Los diferentes operadores y operandos que forman la expresión estarán separados por espacios en blanco.

Un ejemplo de fichero de expresiones puede ser el siguiente:

```
# Expresión 4 + (5 * (2 - (-1)))
+ 4 * 5 - 2 -1
# Expresión 134 - 85 * (4 + 3 * (3 + 9))
- 134 * 85 + 4 * 3 + 3 9
```

5.3 Salida del programa.

La salida del programa se realizará por la salida estándar de Java y consistirá en una línea por cada línea del fichero de expresiones utilizado:

- Si la línea era un comentario, se copiará literalmente a la salida.
- Si la línea era una expresión, se imprimirá en la salida el resultado de su evaluación con la estructura de datos para los valores que se hubiera seleccionado.

En nuestro ejemplo la salida esperada sería la siguiente:

```
# Expresión 4 + (5 * (2 - (-1)))
19
# Expresión 134 - 85 * (4 + 3 * (3 + 9))
-3266
```

6. Ejecución y juegos de prueba.

El Equipo Docente proporcionará, a través del curso virtual, unos juegos de prueba para que los estudiantes puedan comprobar el correcto funcionamiento del programa. Para la ejecución del programa se deberá abrir una consola y ejecutar:

```
java –jar eped2019.jar <estructura> <expresiones>
```

siendo:

- **<estructura>** parámetro que selecciona la estructura de datos a utilizar (INT ó SEQ).
- **<expresiones>** nombre del fichero con las expresiones a evaluar.

7. Documentación y plazos de entrega.

La práctica supone un 20% de la calificación de la asignatura, y es necesario aprobarla para superar la asignatura. Además será necesario obtener, al menos, un 4 sobre 10 en el examen presencial para que la calificación de la práctica sea tenida en cuenta de cara a la calificación final de la asignatura.

Los estudiantes deberán asistir a una sesión obligatoria de prácticas con su tutor en el Centro Asociado. Estas sesiones son organizadas por los Centros Asociados teniendo en cuenta sus recursos y el número de estudiantes matriculados, por lo que en cada Centro las fechas serán diferentes. Los estudiantes deberán, por tanto, dirigirse a su tutor para conocer las fechas de celebración de estas sesiones.

De igual modo, el plazo y forma de entrega son establecidos por los tutores de forma independiente en cada Centro Asociado, por lo que deberán ser consultados también con ellos.

La documentación que debe entregar cada estudiante consiste en:

- Memoria de práctica, en la que se deberán responder a las preguntas teóricas.
- Implementación en Java de la práctica, de la cual se deberá aportar tanto el código fuente como el programa compilado.