Estudio de caso: Evaluador de expresiones

Evaluador de expresiones – Primera parte

En esta sección, exploraremos el uso de la recursividad para evaluar expresiones numéricas complejas. Primero, examinaremos las diferentes convenciones para especificar expresiones numéricas y luego veremos cómo la recursión hace que sea relativamente fácil implementar la evaluación de expresiones numéricas en una de las convenciones estándar.

Notación Infija, prefija y postfija

Cuando escribimos expresiones numéricas en un programa Java, generalmente colocamos operadores numéricos como + y * entre los dos operandos, como se muestra a continuación:

```
3.5 + 8.2

9.1 * 12.7

7.8 * (2.3 + 2.5)
```

Poner el operador entre los operandos es una convención conocida como notación infija. Una segunda convención es poner el operador delante de los dos operandos, como en los siguientes ejemplos:

```
+ 3.5 8.2

* 9.1 12.7

* 7.8 + 2.3 2.5
```

Poner el operador delante de los operandos es una convención conocida como notación prefija. La notación prefija parece extraña para símbolos como + y *, pero se parece a la notación de función matemática, en la que el nombre de la función va primero. Por ejemplo, si estuviéramos llamando a métodos en lugar de usar operadores, escribiríamos:

```
sum(3.5, 8.2)

mult(9.1, 12.7)

mult(7.8, sum(2.3, 2.5))
```

También existe una tercera convención, en la que el operador aparece después de los dos operandos, como en los siguientes ejemplos:

```
3.5 8.2 +

9.1 12.7 *

7.8 2.3 2.5 + *
```

Esta convención se conoce como notación postfija. A veces también se le conoce como notación polaca inversa o RPN. Durante muchos años, Hewlett-Packard ha vendido calculadoras científicas que usan RPN en lugar de la notación infija normal.

Estamos tan acostumbrados a la notación infija que toma un tiempo acostumbrarse a las otras dos convenciones. Uno de los hechos interesantes que descubrirá si se toma el tiempo de aprender las convenciones prefija y posfija es que la notación infija es la única notación que requiere paréntesis. Las otras dos notaciones son inequívocas. La tabla 1 resume las tres notaciones.

Tabla 1 Notaciones aritméticas

Notación	Descripción	Ejemplos
Infija	Operador entre operandos	2.3 + 4.7 2.6 * 3.7 (3.4 + 7.9) * 18.6 + 2.3 / 4.7
Prefija	Operador antes de operandos (notación funcional)	+ 2.3 4.7 * 2.6 3.7 + * + 3.4 7.9 18.6 / 2.3 4.7
Postfija	Operador después de operandos (notación polaca inversa)	2.3 4.7 + 2.6 3.7 * 3.4 7.9 + 18.6 * 2.3 4.7 / +

Evaluación de expresiones prefijas

De las tres notaciones estándar, la notación prefija se implementa más fácilmente con recursividad. En esta sección, escribiremos un método que lea una expresión en notación prefija desde un Scanner y calcule su valor. Nuestro método se verá así:

```
// pre: la entrada contiene una expresión prefija legal
// post: se consume la expresión y se devuelve el resultado
public static double evaluate(Scanner input) {
 ...
}
```

Antes de que podamos comenzar a escribir el método, debemos considerar el tipo de entrada que vamos a obtener. Como indica la condición previa, supondremos que el Scanner contiene una expresión correcta en notación prefija. La expresión más simple posible sería un número como

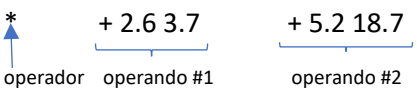
38.9

No hay mucho que evaluar en este caso, simplemente podemos leer y devolver el número. Las expresiones en notación prefija más complejas involucrarán uno o más operadores. Recuerde que el operador va delante de los operandos en una expresión en notación prefija. Una expresión un poco más compleja sería dos números como operandos con un operador delante:

+ 2.6 3.7

Esta expresión podría ser en sí misma un operando en una expresión más grande. Por ejemplo, podríamos pedir * + 2.6 3.7 + 5.2 18.7

En el nivel más externo, tenemos un operador de multiplicación con dos operandos:



En otras palabras, esta expresión está calculando el producto de dos sumas. Aquí está la misma expresión escrita en la notación infija más familiar:

(2.6 + 3.7) * (5.2 + 18.7)

Estas expresiones pueden volverse arbitrariamente complejas. La observación clave que se debe hacer sobre ellos es que todos comienzan con un operador. En otras palabras, cada expresión en notación prefija tiene una de dos formas:

• Un número simple

• Un operador seguido de dos operandos

Esta observación se convertirá en una hoja de ruta para nuestra solución recursiva. La expresión en notación prefija más simple será un número, y podemos distinguirla del otro caso porque cualquier otra expresión comenzará con un operador. Entonces podemos comenzar nuestra solución recursiva verificando si el siguiente token en el Scanner es un número. Si es así, tenemos un caso simple y simplemente podemos leer y devolver el número:

```
public static double evaluate(Scanner input) {
 if (input.hasNextDouble()) {
 // caso base con un número simple
 return input.nextDouble();
 } else {
 // caso recursivo con un operador y dos operandos
 ...
 }
}
```

Volviendo nuestra atención al caso recursivo, sabemos que la entrada debe ser un operador seguido de dos operandos. Podemos comenzar leyendo el operador:

```
public static double evaluate(Scanner input) {
 if (input.hasNextDouble()) {
 // caso base con un número simple
 return input.nextDouble();
 } else {
 // caso recursivo con un operador y dos operandos
 String operator = input.next();
 ...
 }
}
```

En este punto, llegamos a una decisión crítica. Hemos leído el operador, y ahora necesitamos leer el primer operando y luego el segundo operando. Si supiéramos que los operandos son números simples, podríamos escribir un código como el siguiente:

```
// no es el enfoque correcto
```

```
public static double evaluate(Scanner input) {
 if (input.hasNextDouble()) {
 // caso base con un número simple
 return input.nextDouble();
 } else {
 // caso recursivo con un operador y dos operandos
 String operator = input.next();
 double operand1 = input.nextDouble();
 double operand2 = input.nextDouble();
 ...
 }
}
```

Pero no tenemos garantía de que los operandos sean números simples. Pueden ser expresiones complejas que comienzan con operadores. Su instinto podría ser, probar si el operador original es seguido o no por otro operador (en otras palabras, si el primer operando comienza con un operador), pero ese razonamiento no lo llevará a un resultado satisfactorio. Recuerde que las expresiones pueden ser arbitrariamente complejas, por lo que cualquiera de los operandos puede contener docenas de operadores para procesar.

La solución a este rompecabezas implica la recursividad. Necesitamos leer dos operandos del Scanner, y pueden ser muy complejos. Pero sabemos que están en forma de prefijo y sabemos que no son tan complejos como la expresión original que nos pidieron evaluar. La clave es evaluar recursivamente cada uno de los dos operandos:

```
public static double evaluate(Scanner input) {
 if (input.hasNextDouble()) {
 // caso base con un número simple
 return input.nextDouble();
 } else {
 // caso recursivo con un operador y dos operandos
 String operator = input.next();
 double operand1 = evaluate(input);
 double operand2 = evaluate(input);
 ...
 }
}
```

Esta sencilla solución funciona. Por supuesto, todavía tenemos la tarea de evaluar al operador. Después de que se hayan ejecutado las dos llamadas recursivas, tendremos un operador y dos

números (digamos, +, 3.4 y 2.6). Sería bueno si pudiéramos escribir una declaración como la siguiente:

return operand1 operator operand2; // no funciona

Desafortunadamente, Java no funciona de esa manera. Tenemos que usar una declaración if/else anidada para probar qué tipo de operador tenemos y devolver un valor apropiado:

```
if (operator.equals("+")) {
 return operand1 + operand2;
} else if (operator.equals("–")) {
 return operand1 – operand2;
} else if (operator.equals("*")) {
 ...
```

Podemos incluir este código en su propio método para que nuestro método recursivo sea bastante corto:

```
public static double evaluate(Scanner input) {
 if (input.hasNextDouble()) {
 // caso base con un número simple
 return input.nextDouble();
 } else {
 // caso recursivo con un operador y dos operandos
 String operator = input.next();
 double operand1 = evaluate(input);
 double operand2 = evaluate(input);
 return apply(operator, operand1, operand2);
 }
}
```

Programa completo

Cuando programe con recursividad, notará dos cosas. Primero, el código recursivo que escriba tenderá a ser bastante corto, aunque podría estar resolviendo una tarea muy compleja. En segundo lugar, la mayor parte de su programa generalmente terminará siendo código de soporte para la recursividad que realiza tareas de bajo nivel. Para nuestra tarea actual de evaluar una expresión en notación prefija, tenemos un evaluador breve y potente, pero necesitamos incluir algún código de apoyo que explique el programa al usuario, solicite una expresión en notación prefija e informe el resultado. También descubrimos que necesitábamos un método que aplicara un operador a dos operandos. Las partes no recursivas del programa son bastante sencillas, por lo que se incluyen en el siguiente código sin una discusión detallada:

```
// Este programa solicita y evalúa una expresión prefija
import java.util.*;
public class PrefixEvaluator {
public static void main(String[] args) {
Scanner console = new Scanner(System.in);
System.out.println("Este programa evalúa");
System.out.println("expresiones de prefijo que incluyen los ");
System.out.println("operadores +, –, *, / y %");
System.out.print("expresión? ");
double value = evaluate(console);
System.out.println("valor = " + value);
}
// pre: la entrada contiene una expresión de prefijo legal
// post: se consume la expresión y se devuelve el resultado
public static double evaluate(Scanner input) {
if (input.hasNextDouble()) {
return input.nextDouble();
} else {
String operator = input.next();
double operand1 = evaluate(input);
double operand2 = evaluate(input);
return apply(operator, operand1, operand2);
}
}
// pre : el operador es uno de +, -, *, / o %
// post: devuelve el resultado de aplicar el operador dado
// a los operandos dados
public static double apply(String operator, double operand1, double operand2)
{
if (operator.equals("+")) {
return operand1 + operand2;
} else if (operator.equals("-")) {
return operand1 - operand2;
} else if (operator.equals("*")) {
```

```
return operand1 * operand2;
} else if (operator.equals("/")) {
return operand1 / operand2;
} else if (operator.equals("%")) {
return operand1 % operand2; } else {
 throw new IllegalArgumentException("Operador incorrecto: "
 + operator);
}
 }
 }
```

El programa puede manejar números simples, como en la siguiente ejecución de ejemplo:

```
Este programa evalúa
expresiones de prefijo que incluyen los
operadores +, -, *, / y %
expresión? 38.9
valor = 38.9
>
```

También puede manejar expresiones con un solo operador, como en la siguiente ejecución:

```
Este programa evalúa
expresiones de prefijo que incluyen los
operadores +, -, *, / y %
expresión? + 2.6 3.7
valor = 6.300000000000001
```

Y maneja el caso que consideramos que involucraba un producto de dos sumas:

```
Este programa evalúa
expresiones de prefijo que incluyen los
operadores +, -, *, / y %
expresión? * + 2.6 3.7 + 5.2 18.7
valor = 150.57000000000002
```

De hecho, puede manejar expresiones arbitrariamente complejas, como en la siguiente ejecución de ejemplo:

```
Este programa evalúa
expresiones de prefijo que incluyen los
operadores +, -, *, / y %
expresión? / + * - 17.4 8.9 - 3.9 4.7 18.4 - 3.8 * 7.9 2.3
valor = -0.8072372999304106
```

La expresión que se calcula en el ejemplo anterior es el prefijo equivalente a la siguiente expresión infija: ((17.4 – 8.9) * (3.9 – 4.7) + 18.4) / (3.8 – 7.9 * 2.3)

Evaluador de expresiones – Segunda parte

En el bloque anterior, vimos cómo escribir un programa que usa recursividad para evaluar expresiones en notación prefija. Sabemos que las soluciones recursivas aprovechan la pila de llamadas a métodos. Pero, ¿y si no tuviera recursividad disponible para usted? Resulta que muchos problemas que se resuelven fácilmente con la recursividad también se resuelven fácilmente con una pila.

Vamos a escribir un programa que evalúe expresiones aritméticas entre paréntesis usando los operadores estándar para suma, resta, multiplicación y división, y un operador especial para exponenciación ("^"). Por ejemplo, si quisiéramos evaluar esta expresión:

$$18.4 - \frac{2.3 \times 8.5}{19.5 + 2.7^{4.9}}$$

Lo escribiríamos como una expresión entre paréntesis como:

(18.4 - ((2.3*8.5) / (19.5+(2.7^4.9))))

Tenemos convenciones que nos permiten omitir muchos de los paréntesis en esta expresión, pero es muy difícil procesar tales expresiones porque luego tenemos que manejar los problemas de precedencia. Lo mantendremos simple requiriendo siempre paréntesis para cada operador.

En el bloque anterior donde resolvimos el caso con recursividad, asumimos que se utilizarían espacios para separar todos los tokens individuales. Eso nos permitió usar un Scanner para leer los tokens, pero puede ser muy molesto tener que incluir espacios para cada elemento individual. Por ejemplo, la expresión anterior tendría que ser reescrita como:

(18.4 - ((2.3 * 8.5) / (19.5 + (2.7 ^ 4.9))))

Resulta que separar una cadena como esta en tokens conduce a una aplicación interesante para una cola. Así que esta vez escribiremos una versión más sofisticada que le permita tener tanto o tan poco espacio como desee.

También debemos tener cuidado de informar la mayoría de los errores si el usuario deja los paréntesis o no los hace coincidir correctamente. Esta verificación de errores es un buen complemento para las otras operaciones que realizaremos para evaluar estas expresiones.

Pero evitaremos un tema espinoso. No permitiremos que los números tengan un signo menos delante de ellos. Entonces, aunque tiene sentido formar una expresión como esta:

(-2.5/4.5)

No permitiremos ese signo menos delante del 2.5. En su lugar, tendría que formar una expresión como:

((0-2.5)/4.5)

División en tokens

Hemos visto que la clase Scanner se puede usar para “tokenizar” una cadena usando espacios en blanco, pero queremos permitir que un usuario omita el espacio en blanco y aún tenga su

entrada “tokenizada” correctamente. Vamos a construir una clase de soporte llamada StringSplitter que resolverá esta tarea bastante especializada. Se comportará como un Scanner, pero no requerirá espacios en blanco para dividir la cadena.

Podemos usar los nombres de los métodos estándar de la clase Scanner para obtener el siguiente token y preguntar si hay otro token. También deberíamos introducir un método “peek” que permita al cliente preguntar sobre el siguiente token sin leerlo realmente. Y necesitaremos un constructor que tome la cadena a dividir como parámetro. Entonces los métodos públicos serán:

```
public StringSplitter(String line)

public boolean hasNext()

public String next()

public String peek()
```

Esta tarea implicará escanear las letras de una cadena de principio a fin y devolver al cliente de la clase los tokens individuales que encontremos. Nuestra clase tiene que hacer un seguimiento de lo que ya se ha leído y lo que queda por leer. También requerirá mucho mirar hacia adelante para descubrir cómo separar los caracteres individuales en tokens. Por ejemplo, si la cadena es "(2.3*4.8)", queremos producir esta secuencia de tokens:

"(", "2.3", "*", "4.8", ")"

Como estamos leyendo un número como 2.3, tendremos que mirar hacia adelante para ver qué viene a continuación para que podamos reconocer el final del número.

Resulta que una cola es una gran estructura para resolver este problema. Nos permite examinar todos los caracteres de principio a fin y hace un seguimiento de lo que ya hemos mirado y lo que nos queda por mirar. También nos permite mirar hacia adelante.

La gran diferencia entre nuestro divisor y un Scanner simple es que necesitamos reconocer ciertos caracteres especiales como tokens. Si encontramos un paréntesis o uno de los operadores aritméticos, tenemos que convertirlo en un token, esté o no rodeado de espacios en blanco. Es mejor incluir estos caracteres especiales en una constante de cadena para la clase.

En cuanto a los campos o atributos de la clase, dado que estamos leyendo un carácter a la vez, queremos que Queue<Character> almacene los caracteres individuales de la cadena y un campo o atributo para almacenar el siguiente token. Debido a que queremos permitir que el cliente mire hacia adelante, debemos tener el campo o atributo para que el token almacene siempre el próximo token que se procesará (de esa manera podemos permitir que el cliente lo mire tanto como quiera sin realmente leer nada).

Tenemos que proporcionar al cliente una forma de preguntar si hay un próximo token para procesar. Podríamos usar un campo o atributo adicional para esto, pero también podríamos establecer el campo de token en null cuando no queden tokens para procesar.

Aquí hay un esquema básico de nuestra clase que incluye nuestros dos campos o atributos, nuestra constante especial y un constructor que agrega las letras de la cadena a nuestra cola de caracteres:

```
public class StringSplitter {
```

```
  private Queue<Character> characters;
  private String token;
  public static final String SPECIAL_CHARACTERS = "()+-*/^";
  public StringSplitter(String line) {
    characters = new LinkedList<>();
    for (int i = 0; i < line.length(); i++) {
      characters.add(line.charAt(i));
      }
    findNextToken();
  }
  ...
}
```

Observe que el constructor termina con una llamada a un método llamado findNextToken. La idea es que queremos un método privado que procese la cola y le dé un valor apropiado al campo o atributo llamado token. La mayor parte del trabajo de escribir esta clase es escribir ese método.

Para escribir el método findNextToken, tenemos que considerar todos los casos posibles y asegurarnos de manejar cada uno. Queremos omitir los espacios en blanco tal como lo hace el Scanner, por lo que tendremos que incluir código para hacerlo. Después de omitir cualquier espacio en blanco inicial, normalmente estaríamos listos para construir el siguiente token. Pero tenemos que considerar el caso en el que nos quedamos sin caracteres, en cuyo caso no hay más tokens para producir. Si tenemos un token para producir, tendremos que eliminar el contenido de la cola de uno en uno y agregarlos al token que estamos construyendo. El enfoque básico se puede describir con el siguiente pseudocódigo:

```
omitir espacios en blanco.
if (no queda nada) {
token = null;
} else {
inicializa el token al siguiente carácter de la cola.
while (el siguiente carácter de cola es parte de este token) {
token = token + (siguiente carácter de la cola).
}
}
```

Para omitir el espacio en blanco inicial, podemos mirar hacia adelante en la cola para ver si el siguiente carácter es un carácter de espacio en blanco. Utilizaremos un método estático útil de la clase contenedora Character llamada isWhitespace que devuelve true si un carácter determinado es un espacio en blanco, como un espacio, una tabulación o un salto de línea.

Pero debemos tener en cuenta que no podemos mirar hacia adelante cuando la cola está vacía, por lo que también debemos incluir una prueba especial para eso:

```
while (!characters.isEmpty() && Character.isWhitespace(characters.peek())) {
 characters.remove();
}
if (characters.isEmpty()) {
 token = null;
} else {
 ...
}
```

Ahora necesitamos escribir el código para construir un token carácter por carácter. Podemos inicializar el token al siguiente carácter en la cola diciendo:

token = "" + characters.remove ();

Entonces nos encontramos con el problema de saber cuántos caracteres más incluir en este token. Tenemos un conjunto de caracteres especiales que se supone que son tokens de un carácter, por lo que si el token es alguno de esos, debemos dejar de agregar caracteres al token. Podemos usar nuestra constante de cadena y una llamada a contains para manejar ese caso especial:

```
if (!SPECIAL_CHARACTERS.contains(token)) {
 …
}
```

Ahora necesitamos un ciclo que agregará caracteres al token actual hasta que encuentre algo que no sea parte del token. Queríamos detenernos si nos encontramos con un carácter de espacio en blanco. Pero también nos gustaría detenernos si nos encontramos con alguno de los caracteres especiales. Y tenemos un problema adicional en el sentido de que podríamos quedarnos completamente sin caracteres si este es el último token que se procesará.

La lógica se vuelve bastante complicada en este caso, por lo que es útil introducir una variable booleana que realice un seguimiento de si hemos terminado o no:

```
boolean done = false;
while (!characters.isEmpty() && !done) {
 char ch = characters.peek();
 if (Character.isWhitespace(ch) ||
 SPECIAL_CHARACTERS.indexOf(ch) >= 0) {
 done = true;
 } else {
 token = token + characters.remove();
 }
```

```
}
```

Esto completa el método privado para encontrar el siguiente token. Como se mencionó anteriormente, las otras partes de la clase son bastante sencillas. La siguiente es la definición completa de la clase. Tenga en cuenta que la clase tiene un método privado adicional llamado checkToken que lanza una excepción NoSuchElementException si el cliente llama a los métodos peek o next cuando no quedan más tokens.

```
// Esta clase divide una cadena en una secuencia de tokens utilizando
// espacios en blanco y una lista de caracteres especiales que se
// consideran tokens. Los caracteres especiales en este caso
// se utilizan para tokenizar una expresión aritmética. Por ejemplo
// la expresion:
// 2*3.8/(4.95–7.8)
// se tokenizaría como 2 * 3.8 / ( 4.95 – 7.8 ) a pesar de que no tiene
// espacios en blanco para separar estos tokens
import java.util.*;
public class StringSplitter {
private Queue<Character> characters;
private String token;
public static final String SPECIAL_CHARACTERS = "()+-*/^";
public StringSplitter(String line) {
characters = new LinkedList<>();
for (int i = 0; i < line.length(); i++) {
characters.add(line.charAt(i));
}
findNextToken();
}
// post: Devuelve true si hay otro token
public boolean hasNext() {
return token != null;
}
// pre: hay otro token para devolver (lanza
// NoSuchElementException si no es así)
// post: devuelve y consume el siguiente token
public String next() {
checkToken();
String result = token;
findNextToken();
```

```
return result;
}
// pre: hay otro token para devolver (lanza
// NoSuchElementException si no es así)
// post: devuelve el siguiente token sin consumirlo
public String peek() {
checkToken();
return token;
 }
// post: encuentra el siguiente token, si lo hay
private void findNextToken() {
while (!characters.isEmpty()&& Character.isWhitespace(characters.peek())){
characters.remove();
}
if (characters.isEmpty()) {
token = null;
 } else {
token = "" + characters.remove();
if (!SPECIAL_CHARACTERS.contains(token)) {
boolean done = false;
while (!characters.isEmpty() && !done) {
char ch = characters.peek();
if (Character.isWhitespace(ch) ||
SPECIAL_CHARACTERS.indexOf(ch) >= 0) {
done = true;
 } else {
token = token + characters.remove();
 }
 }
}
 }
 }
// post: lanza una excepción si no queda ningún token
private void checkToken() {
if (!hasNext()) {
throw new NoSuchElementException();
```

```
}
}
}
```

El evaluador

Ahora que tenemos una clase de soporte que nos permitirá leer los tokens de una cadena, podemos trabajar en el código que evaluará los tokens que encontremos en una expresión completamente entre paréntesis. Vamos a implementar una variación de un famoso algoritmo conocido como el algoritmo del patio de maniobras (shunting-yard) que fue inventado por Edsger Dijkstra. Utiliza dos pilas para guardar resultados intermedios. Una pila almacena números y la otra pila almacena símbolos.

La idea básica es que almacenamos valores en las dos pilas hasta que estemos listos para procesarlos. Cuando vemos paréntesis izquierdos y operadores, los colocamos en la pila de símbolos. A medida que vemos los números, los colocamos en la pila de números. Y cuando vemos un paréntesis derecho, sabemos que tenemos toda la información para una subexpresión determinada, seguimos adelante y la evaluamos. Luego empujamos el resultado de vuelta a la pila de números. Considere un caso simple de evaluar "(2+3)". La tabla 2 muestra cómo a las pilas inicialmente vacías se les agregan elementos hasta que encontramos el paréntesis correcto, momento en el que evaluamos la suma y empujamos el resultado a la pila de números.

Tabla 2 Evaluación de "(2+3)"

Token	Acción	Pila de símbolos	Pila de números
		[]	[]
(	Push hacia la pila de símbolos	[(]	[]
2	Push hacia la pila de números	[(]	[2.0]
+	Push hacia la pila de símbolos	[(, +]	[2.0]
3	Push hacia la pila de números	[(, +]	[2.0, 3.0]
)	Evaluar la expresión y enviar el resultado (push) a la pila de números	[]	[5.0]

Tenga en cuenta que el resultado general es el único valor almacenado en la pila de números cuando terminamos y la pila de símbolos está vacía cuando terminamos. Si quedan otros valores en cualquiera de las pilas, entonces sabemos que teníamos una expresión ilegal.

Esto parece mucho trabajo para un cálculo bastante simple. Pero recuerde que podemos formar subexpresiones complejas que también deben evaluarse. Por ejemplo, ¿qué pasaría si la expresión a evaluar hubiera sido "((4/2)+(7-4))"? Esta expresión tendrá el mismo valor porque (4/2) se evalúa como 2 y (7−4) se evalúa como 3. Con el enfoque de dos pilas, podemos realizar un seguimiento de cada parte de esta expresión hasta que estemos listos para procesarla. La tabla 3 muestra la evaluación de la expresión más compleja.

Tabla 3 Evaluación de "((4/2)+(7-4))"

Token	Acción	Pila de símbolos	Pila de números
		[]	[]
(	Push hacia la pila de símbolos	[(]	[]
(	Push hacia la pila de símbolos	[(,(]	[]
4	Push hacia la pila de números	[(, (]	[4.0]
/	Push hacia la pila de símbolos	[(, (, /]	[4.0]
2	Push hacia la pila de números	[(, (, /]	[4.0, 2.0]
)	Evaluar la expresión y enviar el resultado (push) a la pila de números	[(]	[2.0]
+	Push hacia la pila de símbolos	[(, +]	[2.0]
(	Push hacia la pila de símbolos	[(, +, (]	[2.0]
7	Push hacia la pila de números	[(, +, (]	[2.0, 7.0]
-	Push hacia la pila de símbolos	[(, +, (, -]	[2.0, 7.0]
4	Push hacia la pila de números	[(, +, (, -]	[2.0, 7.0, 4.0]
)	Evaluar la expresión y enviar el resultado (push) a la pila de números	[(, +]	[2.0, 3.0]
)	Evaluar la expresión y enviar el resultado (push) a la pila de números	[]	[5.0]

Para escribir el código, solo tenemos que implementar el algoritmo. Es más fácil si asumimos que la entrada no tiene errores, pero es mejor reconocer los errores cuando podamos. Realmente no podemos recuperarnos de un error y puede ser complejo informar la naturaleza de cada error. Entonces, busquemos un término medio para reconocer tantos errores como podamos, pero dando solo un mensaje de error simple si encontramos un error.

Sabemos que queremos procesar tokens hasta que encontremos un error o nos quedemos sin tokens. Es útil introducir un indicador booleano que realice un seguimiento de si se ha detectado un error. Esperamos llegar a un punto en el que la pila de símbolos esté vacía y la pila de números tenga exactamente un valor. En ese caso, podríamos reportar ese número como el resultado general. La estructura básica de nuestra solución es:

```
StringSplitter data = new StringSplitter(line);
Stack<String> symbols = new Stack<>();
Stack<Double> values = new Stack<>();
boolean error = false;
while (!error && data.hasNext()) {
 ...
}
if (error || values.size() != 1 || !symbols.isEmpty()) {
 System.out.println("expresión ilegal");
} else {
 System.out.println(values.pop());
}
```

Nuestra tarea restante es completar el cuerpo del ciclo while para procesar tokens. Dos de los casos son relativamente simples. Cuando vemos un paréntesis izquierdo u operador, lo empujamos en la pila de símbolos. Cuando vemos un número, lo empujamos en la pila de números. La parte difícil es cuando vemos un paréntesis derecho. El cuerpo de nuestro ciclo while se verá así:

```
String next = data.next();
if (next.equals(")")) {
 // proceso )
} else if ("(+–*/^".contains(next)) {
 symbols.push(next);
  } else { // debería ser un número
 values.push(Double.parseDouble(next));
}
```

Observe que en el caso final llamamos al método Double.parseDouble para convertir el token de una cadena en un double. Como se señaló anteriormente, dos de cada tres de estos casos son simples. El tercer caso es el difícil. ¿Cómo procesamos un paréntesis derecho?

Si una expresión es legal, entonces deberíamos haber encontrado un paréntesis izquierdo al principio y dos números con los que trabajar y un operador para evaluar. Así que esperamos que la pila de símbolos tenga un operador en la parte superior de la pila y un paréntesis izquierdo justo debajo. Los dos valores deben estar en la pila de números. En general, eliminaremos el operador y el paréntesis izquierdo, eliminaremos los dos números y luego aplicaremos el operador. Lo que complica esto es que puede haber todo tipo de errores. Tenemos que tener cuidado de comprobar en cada paso que tenemos lo que se supone que tenemos.

Una cosa que sabemos es que los únicos valores que se insertan en la pila de símbolos son los operadores legales y los paréntesis izquierdos. Podemos usar los tamaños de pila para varias

de nuestras pruebas, pero tenemos que asegurarnos de que tenemos un operador y no un paréntesis izquierdo y que debajo de él en la pila hay un paréntesis izquierdo. El siguiente código verifica estos errores y evalúa un operador, devolviendo el resultado a la pila de números:

```
if (symbols.size() < 2 || symbols.peek().equals("(")) {
 error = true;
} else {
 String operator = symbols.pop();
 if (!symbols.peek().equals("(")) {
 error = true;
 } else {
 symbols.pop(); // para eliminar el "("
 double op2 = values.pop();
 double op1 = values.pop();
 double value = evaluate(operator, op1, op2);
 values.push(value);
 }
}
```

El código anterior implica una llamada a un método llamado evaluate. Escribimos este método en la solución recursiva de la evaluación de la expresión prefija. Toma un operador y dos operandos y devuelve el resultado de aplicar ese operador a los dos operandos. Poniendo todo esto junto, terminamos con el siguiente programa completo:

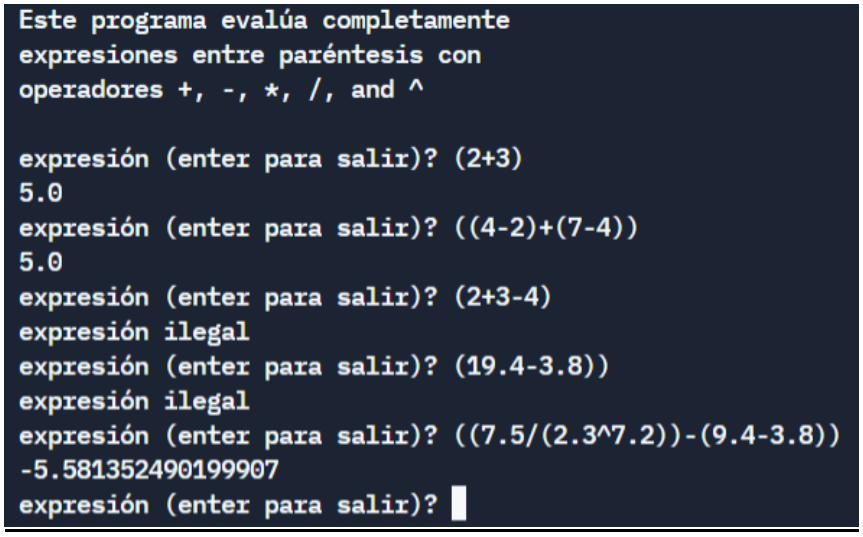
```
// Este programa solicita expresiones aritméticas
// completamente entre paréntesis y evalúa cada expresión. Utiliza dos
// pilas para evaluar las expresiones.

import java.util.*;
public class Evaluator {
public static void main(String[] args) {
System.out.println("Este programa evalúa completamente");
System.out.println("expresiones entre paréntesis con");
System.out.println("operadores +, -, *, /, and ^");
System.out.println();
Scanner console = new Scanner(System.in);
System.out.print("expresión (enter para salir)? ");
String line = console.nextLine().trim();
```

```
while (line.length() > 0) {
evaluate(line);
System.out.print("expresión (enter para salir)? ");
line = console.nextLine().trim();
}
}
// pre: la línea contiene una expresión completamente entre paréntesis
// post: imprime el valor de la expresión o un error
// mensaje si la expresión no es legal
public static void evaluate(String line) {
StringSplitter data = new StringSplitter(line);
Stack<String> symbols = new Stack<String>();
Stack<Double> values = new Stack<Double>();
boolean error = false;
while (!error && data.hasNext()) {
String next = data.next();
if (next.equals(")")) {
if (symbols.size() < 2 ||
symbols.peek().equals("(")) {
error = true;
 } else {
 String operator = symbols.pop();
if (!symbols.peek().equals("(")) {
error = true;
 } else {
symbols.pop(); // para eliminar el "("
double op2 = values.pop();
double op1 = values.pop();
 double value = evaluate(operator, op1, op2);
 values.push(value);
 }
 }
 } else if ("(+-*/^".contains(next)) {
symbols.push(next);
 } else { // debería ser un número
values.push(Double.parseDouble(next));
```

```
 }
}
 if (error || values.size() != 1 || !symbols.isEmpty()) {
System.out.println("expresión ilegal");
 } else {
 System.out.println(values.pop());
 }
 }
// pre: el operador es uno de +, -, *, / o ^
// post: devuelve el resultado de aplicar el operador dado a
// los operandos dados
public static double evaluate(String operator, double operand1,
double operand2) {
if (operator.equals("+")) {
return operand1 + operand2;
 } else if (operator.equals("-")) {
return operand1 - operand2;
} else if (operator.equals("*")) {
return operand1 * operand2;
 } else if (operator.equals("/")) {
 return operand1 / operand2;
 } else if (operator.equals("^")) {
return Math.pow(operand1, operand2);
 } else {
throw new RuntimeException(
 "operador ilegal " + operator);
 }
 }
 }
```

A continuación, se muestra un ejemplo de registro de ejecución:



Este programa es bastante robusto. El único error que no verifica es si se ingresa un token ilegal. Se asumirá que cualquier token ilegal es un número. El ciclo principal de procesamiento de tokens incluye una llamada a Double.parseDouble que generará una NumberFormatException si encuentra dicho token. Esto podría solucionarse agregando un bloque try/catch que establece el indicador de error en verdadero si se lanza NumberFormatException .

Problema propuesto

Modifique el programa del estudio de caso discutido para crear un nuevo programa que acepte como entrada una expresión infija con paréntesis y devuelva una cadena que represente una expresión postfija equivalente. Las expresiones Posfija son aquellas en las que cada operador sigue sus dos operandos, como 1 2 + en lugar de 1 + 2. Las expresiones Postfija son elegantes porque no necesitan paréntesis. Por ejemplo, la expresión infija dada:

d)

es equivalente a la siguiente expresión postfija:

9 8 7 * 6 5 4 ^ / 3 * - 2 * +

Su algoritmo debe leer la expresión token por token, utilizando una pila para almacenar operadores. (Ya no necesita la pila de valores del estudio de caso original). En su lugar, cada vez que encuentre un número, agréguelo a una cadena que esté creando. Cada vez que encuentre un paréntesis derecho, abra la pila para obtener un operador y agréguelo a la cadena que está creando. Deje el resto del código en su lugar para preservar la verificación de errores. Devuelva la cadena "expresión ilegal" si se encuentra un error.