CAPÍTULO

11 Las pilas y los compiladores

Las pilas se utilizan muy a menudo en los compiladores. Este capítulo presenta dos componentes simples de un compilador: un analizador sintáctico de símbolos equilibrados y una calculadora sencilla. El objetivo consiste en presentar algoritmos simples que utilicen pilas para, a través de ellos, ver cómo se utilizan las estructuras de datos descritas en el Capítulo 6.

En este capítulo veremos:

- Cómo utilizar una pila para comprobar si los símbolos de una expresión están equilibrados.
- Cómo utilizar una máquina de estados para hacer el análisis sintáctico de símbolos equilibrados.
- Cómo usar un análisis sintáctico con precedencia de operadores para evaluar expresiones infijas en un programa que implemente una calculadora sencilla.

11.1 Analizador de símbolos equilibrados

Como se discutió en la Sección 6.2, los compiladores comprueban los programas buscando errores sintácticos. Sin embargo, en ocasiones la falta de un símbolo (como la falta de un finalizador de comentario `*/` o `}`) puede causar que el compilador desparrame cientos de líneas de errores sin llegar a identificar el error real. Una herramienta útil para ayudar a la depuración de mensajes de error del compilador es un programa que compruebe si los símbolos están equilibrados. En otras palabras, toda `{` debe tener asociada una `}`, todo `[` un `]`, y así sucesivamente. Sin embargo, contar simplemente el número de apariciones de cada símbolo no es suficiente. Por ejemplo, la secuencia `[()]` es correcta, pero `[(])` no lo es.

11.1.1 El algoritmo básico

Una pila puede utilizarse para detectar símbolos mal emparejados.

En esta situación es útil una pila porque sabemos que cuando encontramos un símbolo de terminación como `)`, debe emparejarse con el `(` anterior más reciente aún no emparejado. Por tanto, colocando los símbolos de apertura en una pila, pode-

mos fácilmente comprobar si la aparición de un símbolo de terminación es o no oportuna. En concreto, tenemos el siguiente algoritmo:

1. Inicializar una pila vacía.
2. Leer símbolos hasta el final del fichero.
   a) Si el símbolo es de apertura, apilarlo en la pila.
   b) Si el símbolo es de terminación y la pila está vacía, producir un error.
   c) En otro caso, desapilar la cima de la pila. Si el símbolo desapilado no es el correspondiente símbolo de apertura, producir un error.
3. Al finalizar el fichero, si la pila no está vacía, producir un error.

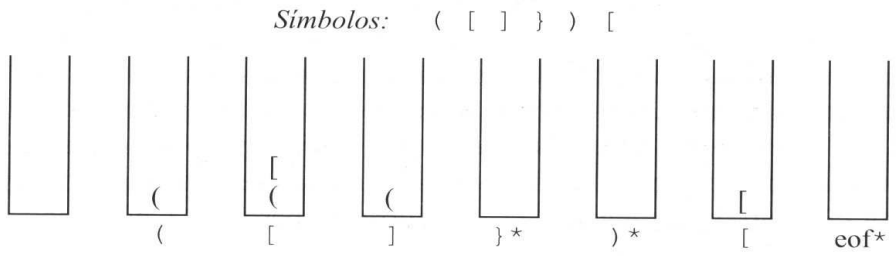
En este algoritmo, ilustrado en la Figura 11.1, los símbolos cuarto, quinto y sexto generan errores. La `}` es un error porque el símbolo desapilado es `(`, por lo que se detecta un mal emparejamiento. El `)` provoca un error porque la pila está vacía, de modo que no existe el correspondiente `(`. Finalmente, el `[` provoca un error, que se detecta cuando finaliza el fichero y la pila no está vacía.

Los símbolos en comentarios, constantes de tipo cadena y constantes de tipo carácter no tiene por qué estar equilibrados.

Para que esto funcione con programas en Java, necesitamos considerar todos los contextos en los cuales los paréntesis, las llaves y los corchetes no tienen por qué estar emparejados. Por ejemplo, no deberíamos considerar un paréntesis como un símbolo cuando aparece dentro de un comentario, de una cadena de caracteres constante o es una constante carácter. Necesitamos, por tanto, rutinas que se salten los comentarios, las cadenas constantes y los caracteres constantes. En Java es bastante complejo detectar una constante carácter debido a las secuencias de escape, por lo que simplificaremos las hipótesis de trabajo. El objetivo será diseñar un programa que funcione para la mayor parte de entradas comunes.

Mostrar los números de línea donde se detectan los errores es fundamental de cara a localizar dichos errores.

Para que el programa sea útil, no sólo debe mostrar los fallos de emparejamiento, sino también identificar la línea donde aparecen. En consecuencia, llevaremos cuenta del número de línea donde se encuentra cada símbolo leído. Cuando se detecta un error, componer el mensaje apropiado siempre resulta complicado. Si hay una `}` de más, ¿significa que sobra la `}`, o que falta una `{`? Mantendremos el tratamiento de errores tan simple como sea posible. Irremediablemente es posible que una vez que se muestra un error el programa se desoriente y empiece a mostrar muchos errores. Por tanto, sólo debemos, en principio, buscar la causa del primer error detectado en cada zona del texto examinado. A pesar de todas estas limitaciones, el programa que desarrollamos resulta bastante útil.




```
Errores (indicados con *):
    } cuando se esperaba un )
    ) con ningún símbolo de apertura con el que emparejarlo
    [ sin pareja al terminar la entrada
```

Figura 11.1 Operaciones sobre la pila en el algoritmo de comprobación de símbolos equilibrados.

11.1.2 Implementación

La Figura 11.2 muestra la clase `AnalizadorJava`, que realiza todo el trabajo. Además del constructor, la única otra rutina visible públicamente es `comprobarEquilibrados`, que aparece en la línea 26. El resto son rutinas auxiliares o atributos de la clase. Empezamos describiendo los atributos de la clase.

```
1  // Clase AnalizadorJava: comprobación de símbolos equilibrados
2  //
3  // CONSTRUCCIÓN: con un objeto PushbackReader
4  // ********************OPERACIONES PÚBLICAS*********************
5  // int comprobarEquilibrados( ) --> Imprime los errores y
6  //                                  devuelve el número de errores
7  // ***************************ERRORES****************************
8  // Se producen errores al comprobar comentarios o cadenas
9
10 import java.io.*;
11 import EstructurasDatos.*;
12 import Excepciones.*;
13 import Soporte.*;
14
15 class AnalizadorJava
16 {
17     public AnalizadorJava( PushbackReader cadenaEntrada )
18     {
19        errores = 0;
20        car = '\0';
21        lineaActual = 1;
22        entrada = cadenaEntrada;
23        tokensPendientes = new PilaVec( );
24     }
25
26     public int comprobarEquilibrados( )
27       { /* Figura 11.8 */ }
28
29     private PushbackReader entrada; // La cadena de entrada
30     private char car;               // Carácter actual
31     private int lineaActual;        // Línea actual
32     private Pila tokensPendientes;  // Símbolos de apertura
                                          pendientes
33     private int errores;            // Número de errores encontrados
34
35     private boolean siguienteCar( )
36       { /* Figura 11.4 */ }
37     private void devolverCar( )
38       { /* Figura 11.4 */ }
39     private void saltarComentario( int comienzo )
40       { /* Figura 11.5 */ }
41     private void saltarCadena( char tipoCadena )
42       { /* Figura 11.6 */ }
43     private boolean siguienteSimbolo( )
44       { /* Figura 11.7 */ }
45     private void procesarBarra( )
46       { /* Figura 11.7 */ }
47     private void comprobarPareja( Simbolo simAper, Simbolo simFin )
48       { /* Figura 11.9 */ }
49 }
```

Figura 11.2 Esqueleto de la clase para el programa de comprobación de símbolos equilibrados.

`entrada` es un objeto de la clase `PushbackReader` y se inicializa en el constructor. Un objeto de la clase `PushbackReader` es como un `BufferedReader`, excepto por el hecho de que también proporciona un método `unread`. El carácter actual que está siendo explorado se almacena en `car`, y el número de línea actual se almacena en `lineaActual`. El algoritmo para comprobar símbolos equilibrados necesita que coloquemos los símbolos de apertura en una pila. Para poder imprimir diagnósticos, almacenamos un número de línea con cada símbolo, como se muestra en la clase `Simbolo` de la Figura 11.3. La pila se llama `tokensPendientes`, y se declara en la línea 32. Finalmente, en la línea 33 se declara un entero para contar el número de errores encontrados.

El constructor, mostrado en las líneas de la 17 a la 24, inicializa el contador de errores a 0 y el número de línea actual a 1, y da valor a la referencia `PushbackReader`. El resto de atributos, a saber, `car` y `tokensPendientes`, se inizializan adecuadamente. En el caso de `tokensPendientes`, se utiliza un constructor sin parámetros para crear una pila vacía.

El *análisis léxico* se utiliza para ignorar comentarios y reconocer símbolos.

Podemos pasar a examinar algunas de las rutinas auxiliares. Muchas de ellas tienen que ver con el tratamiento de la línea actual, y con la diferenciación de símbolos que representan tokens de apertura o cierre de aquellos que están dentro de comentarios, cadenas o constantes de caracteres. Todo esto se engloba dentro del denominado *análisis léxico*. La Figura 11.4 muestra un par de rutinas, `siguienteCar` y `devolverCar`. `siguienteCar` lee el siguiente carácter, lo asigna a `car`, y actualiza `lineaActual` cuando se trate de un salto de línea. Finalmente, devuelve `false` cuando se alcanza el final de fichero. El procedimiento complementario, `devolverCar`, coloca de nuevo el carácter (`car`) en la cadena de entrada, y decrementa `lineaActual` cuando ello es oportuno. Es claro que `devolverCar` debería llamarse sólo una vez entre dos llamadas a `siguienteCar`. Ya que es una rutina privada, no nos preocupamos de abusos por parte de la clase usuaria.

Devolver caracteres a la línea de entrada es una técnica común en el análisis sintáctico. En muchos casos, se han leído demasiados caracteres y es preciso deshacer el proceso de lectura. En nuestro caso, esto ocurre cuando procesamos una `/`. Debemos ver si el siguiente carácter corresponde al token de comienzo de comentario. Sin embargo, cuando ello no es así, no podemos ignorar este último carácter leído, pues podría ser un símbolo de apertura o cierre, o unas comillas. Por tanto, debemos actuar como si todavía no hubiera sido leído.

```
1  /**
2   * Símbolo que representa el objeto que se apilará.
3   */
4  class Simbolo
5  {
6      char token;
7      int  laLinea;
8
9      Simbolo( char tok, int linea )
10     {
11         token = tok;
12         laLinea = linea;
13     }
14 }
```

Figura 11.3 Clase de objetos que se insertan en la pila.

```
1  /**
2   * siguienteCar da valor a car basándose en el siguiente carácter
3   * en la línea de entrada. devolverCar devuelve el carácter a la
4   * cadena. Sólo debe utilizarse una vez después de siguienteCar.
5   * Ambas rutinas actualizan lineaActual cuando ello es necesario.
6   */
7  private boolean siguienteCar( )
8  {
9      try
10     {
11         int valorLeido = entrada.read( );
12         if( valorLeido == -1 )
13             return false;
14         car = (char) valorLeido;
15         if( car == '\n' )
16             lineaActual++;
17         return true;
18     }
19     catch( IOException e )
20       { return false; }
21 }
22
23 private void devolverCar( )
24 {
25     if( car == '\n' )
26         lineaActual--;
27     try
28       { entrada.unread( (int) car ); }
29     catch( IOException e ) { }
30 }
```

Figura 11.4 Rutina `siguienteCar` para leer el siguiente carácter, actualizar `lineaActual` cuando ello es necesario y devolver `true` cuando no estamos al final del fichero. Rutina `devolverCar` para devolver `car` a la entrada y actualizar `lineaActual` cuando ello es necesario.

La siguiente rutina a comentar es `saltarComentario`, mostrada en la Figura 11.5. El propósito de esta rutina es saltarse los caracteres de un comentario, para que después se siga leyendo la entrada a partir del siguiente carácter tras la finalización del comentario. Las cosas se complican por el hecho de que los comentarios pueden empezar bien por `//`, en cuyo caso el comentario termina con la línea, o con `/*`, en cuyo caso `*/` termina el comentario. Cuando estamos en el primer caso, leemos repetidamente el siguiente carácter, hasta que se alcanza el final del fichero (en cuyo caso, la primera parte del operador `&&` falla) o el final de línea. Tras hacerlo acabamos. Observe que el número de línea es actualizado automáticamente por `siguienteCar`. El caso `/*` se procesa a partir de la línea 17. La posible aparición de `/**` se trata antes de llamar a esta rutina (*véase* la Figura 11.7, líneas 17 y 18).

La rutina `saltarComentario` utiliza una *máquina de estados* simplificada. La máquina de estados es una técnica común en el análisis de símbolos. En cada momento la máquina se encuentra en un determinado estado; cada nuevo carácter de entrada, nos lleva a un nuevo estado. Si todo va bien acabamos por alcanzar un estado en el cual el símbolo ha sido reconocido.

La *máquina de estados* es una técnica común utilizada en el análisis de símbolos.

```
1  /**
2   * Precondición: Estamos a punto de procesar un comentario; hemos
3   *               encontrado ya el token de comienzo de comentario.
4   * Postcondición: La entrada seguirá leyendo justo a partir
5   *                del token de finalización de comentario
6   */
7  private void saltarComentario( int comienzo )
8  {
9      if( comienzo == SLASH_SLASH )
10     {
11         while( siguienteCar( ) && ( car != '\n' ) )
12             ;
13         return;
14     }
15
16         // Buscar la secuencia */
17     boolean estado = false; // true si hemos encontrado *
18
19     while( siguienteCar( ) )
20     {
21         if( estado && car == '/' )
22             return;
23         estado = ( car == '*' );
24     }
25     errores++;
26     System.out.println( "¡Comentario sin terminar!" );
27 }
```

Figura 11.5 Rutina `saltarComentario`, para colocarse después de un comentario ya empezado.

En cada momento, la máquina se encuentra en un determinado estado. Cada nuevo carácter de entrada, nos lleva a un nuevo estado. Si todo va bien, acabamos por alcanzar un estado en el cual el símbolo ha sido reconocido.

En la rutina `saltarComentario`, en cada momento se habrán tratado 0, 1 o 2 caracteres del finalizador `*/`, correspondientes a los estados 0, 1 y 2. Si ya se han leído los dos caracteres, podemos terminar. Por tanto, dentro del bucle sólo podemos estar en los estados 0 o 1, ya que si estamos en el estado 1 y encontramos una `/`, terminamos inmediatamente. En consecuencia, el estado puede representarse mediante una variable booleana que es cierta cuando la máquina está en el estado 1. Si no acabamos, volvemos al estado 1 cuando encontramos un `*`, o al estado 0, en otro caso. Esto se hace en la línea 23.

Si nunca se encuentra el token de finalización de comentario, `siguienteCar` acabará devolviendo `false` y el bucle `while` terminará, provocando un mensaje de error. `saltarCadena`, mostrada en la Figura 11.6, es similar. Aquí, el parámetro es el carácter de comienzo de cadena, que puede ser `"` o `'`. En cada caso, debemos examinar el carácter que termina la cadena. Además, debemos estar preparados para tratar el carácter `\`; en otro caso, el programa produciría errores cuando se ejecutara, por ejemplo, sobre su propio código. Vamos, pues, tratando caracteres. Cuando el carácter leído es una comilla de cierre, hemos terminado. Si encontramos un fin de línea, tenemos una constante carácter o cadena sin terminar. Por último, si encontramos una `\`, leemos el siguiente carácter sin examinarlo.

Una vez escrita la rutina de salto, es fácil escribir `siguienteSimbolo`. Si el carácter actual es `/`, llamamos a `procesarBarra` para leer el siguiente carácter,

```
1  /**
2   * Precondición: Estamos a punto de procesar una cadena; ya hemos
3   *               encontrado la primera comilla.
4   * Postcondición: La entrada seguirá leyendo justo a partir
5   *                de la comilla de cierre correspondiente.
6   */
7  private void saltarCadena( char tipoCadena )
8  {
9      while( siguienteCar( ) )
10     {
11         if( car == tipoCadena )
12             return;
13         if( car == '\n' )
14         {
15             errores++;
16             System.out.println( "Falta cerrar cadena en la línea " +
17                                 lineaActual );
18             return;
19         }
20         else if( car == '\\' )
21             siguienteCar( );
22     }
23 }
```

Figura 11.6 Rutina `saltarCadena`, para colocarnos justo después de una constante carácter o cadena ya empezada.

comprobando si hemos encontrado un comentario; si no, deshacemos la segunda lectura. Si tenemos una comilla, llamamos a `procesarCadena`. Si tenemos un símbolo de apertura o cierre, podemos acabar. En otro caso, seguimos leyendo hasta que se acabe la entrada o se encuentre un símbolo de apertura o cierre. La rutina completa se muestra en la Figura 11.7.

`comprobarEquilibrados` está implementado en la Figura 11.8. Sigue la descripción del algoritmo casi al pie de la letra. Los símbolos de apertura se apilan en la pila junto con el número de línea actual. Cuando se encuentra un símbolo de cierre y la pila está vacía, el símbolo es inesperado; en otro caso, eliminamos el elemento de la cima y comprobamos si el símbolo de apertura que estaba en la pila casa con el símbolo de finalización que se acaba de leer. Esto se hace en la rutina `comprobarPareja`, mostrada en la Figura 11.9. Una vez alcanzado el final del fichero, todos los símbolos en la pila no tendrían pareja; éstos, de existir alguno, son mostrados en la salida por el bucle `while` que empieza en la línea 41. Se devuelve el número total de errores detectados.

`comprobarEquilibrados` hace todo el trabajo del algoritmo.

Observe que la implementación actual permite varias llamadas a `comprobarEquilibrados`. Sin embargo, si la entrada no se reinicializa externamente, lo único que ocurrirá es que se detectará inmediatamente el fin de fichero, acabando inmediatamente. La Figura 11.10 muestra que lo que se espera es que se cree un objeto de la clase `AnalizadorJava` y se llame a `comprobarEquilibrados`. En nuestro ejemplo, si no hay argumentos en la línea de comandos, un objeto de la clase `PushbackReader` se asocia con `System.in`. En otro caso, se utilizan repetidamente objetos de la clase `PushbackReader` asociados a los ficheros indicados en la lista de argumentos suministrada en la línea de comandos.

```
1  /**
2   * Después de encontrar la barra /, tratar el siguiente
3   * carácter. Si comienza un comentario, tratarlo;
4   * en caso contrario, devolver el carácter si no se trata
5   * del carácter de nueva línea
6   */
7  private static final int SLASH_SLASH = 0;
8  private static final int SLASH_STAR = 1;
9
10 private void procesarBarra( )
11 {
12     if( siguienteCar( ) )
13     {
14         if( car == '*' )
15         {
16             // comentario para Javadoc
17             if( siguienteCar( ) && car != '*' )
18                 devolverCar( );
19             saltarComentario( SLASH_STAR );
20         }
21         else if( car == '/' )
22             saltarComentario( SLASH_SLASH );
23         else if( car != '\n' )
24             devolverCar( );
25         }
26     }
27
28     /**
29      * Llegar al siguiente símbolo de apertura o cierre.
30      * Devolver false si se acaba el fichero.
31      * Saltar comentarios y constantes carácter o cadena
32      */
33     private boolean siguienteSimbolo( )
34     {
35         while( siguienteCar( ) )
36         {
37             if( car == '/' )
38                 procesarBarra( );
39             else if( car == '\'' || car == '"' )
40                 saltarCadena( car );
41             else if( car == '(' || car == '[' || car == '{' ||
42                      car == ')' || car == ']' || car == '}' )
43                 return true;
44         }
45         return false; // Final de fichero
46 }
```

Figura 11.7 Rutina `siguienteSimbolo`, para saltar comentarios y cadenas, y devolver el siguiente símbolo de apertura o cierre.

```
1  /**
2   * Imprime un mensaje de error en caso de detectar desequilibrios.
3   * @return número de errores detectados.
4   */
5  public int comprobarEquilibrados ( )
6  {
7      Simbolo pareja = null;
8
9      errores = 0;
10     lineaActual = 1;
11     while( siguienteSimbolo( ) )
12     {
13         char ultimoCar = car;
14         Simbolo ultimoSimbolo = new Simbolo( ultimoCar,
15                                              lineaActual );
16
17         switch( ultimoCar )
18         {
19           case '(': case '[': case '{':
20             tokensPendientes.apilar( ultimoSimbolo );
21             break;
22           case ')': case ']': case '}':
23             try
24             {
25                 pareja = (Simbolo)
26                           tokensPendientes.cimaYDesapilar( );
27                 comprobarPareja( pareja, ultimoSimbolo );
28             }
29             catch( DesbordamientoInferior e )
30             {
31                 errores++;
32                 System.out.println( "Inesperado " + ultimoCar +
33                     " en línea " + lineaActual );
34             }
35             break;
36           default: // no puede ocurrir
37             break;
38         }
39     }
40
41     while( !tokensPendientes.esVacia( ) )
42     {
43         errores++;
44         try
45           { pareja = (Simbolo) tokensPendientes.cimaYDesapilar( ); }
46         catch( DesbordamientoInferior e ) { } // No puede ocurrir
47         System.out.println( pareja.token + " no emparejado " +
48                             " en la línea " + pareja.laLinea );
49     }
50     return errores;
51 }
```

Figura 11.8 `comprobarEquilibrados`, el algoritmo principal.

```
1  // Imprime un mensaje de error si simAper no casa con simFin.
2  // Actualiza los errores.
3  private void comprobarPareja( Simbolo simAper, Simbolo simFin )
4  {
5      if( simAper.token == '(' && simFin.token != ')' ||
6          simAper.token == '[' && simFin.token != ']' ||
7          simAper.token == '{' && simFin.token != '}' )
8      {
9          System.out.println( "Encontrado " + simAper.token +
10           " en la línea " + lineaActual +"; no casa con " +
11           simFin.token + " en la línea " + simAper.laLinea );
12         errores++;
13     }
14 }
```

Figura 11.9 Rutina `comprobarPareja`, para comprobar que el símbolo de terminación casa con el símbolo de apertura.

```
1  // Rutina principal del comprobador de símbolos desequilibrados.
2  public static void main( String [ ] args )
3  {
4      AnalizadorJava p;
5
6      if( args.length == 0 )
7      {
8          p = new AnalizadorJava( new PushbackReader(
9                  new InputStreamReader( System.in ) ) );
10         if( p.comprobarEquilibrados( ) == 0 )
11             System.out.println( "¡No hay errores!" );
12         return;
13     }
14
15     for( int i = 0; i < args.length; i++ )
16     {
17         try
18         {
19             FileReader f = new FileReader( args[ i ] );
20
21             System.out.println( args[ i ] + ": " );
22             p = new AnalizadorJava( new
23                                     PushbackReader( f ) );
24             if( p.comprobarEquilibrados( ) == 0 )
25                 System.out.println( " ...sin errores" );
26             f.close( );
27         }
28         catch( IOException e )
29           { System.err.println( e + args[ i ] ); }
30     }
31 }
```

Figura 11.10 Rutina `main`, con argumentos de la línea de comandos.

11.2 Una calculadora sencilla

Algunas de las técnicas utilizadas en el diseño de compiladores pueden utilizarse a pequeña escala en la implementación de una calculadora de bolsillo típica. Las calculadoras evalúan expresiones infijas, como `1+2`, que consisten en un operador binario junto con argumentos a su izquierda y a su derecha. Este formato, bastante fácil de evaluar en un principio, puede hacerse más complejo. Considere la expresión

```
1 + 2 * 3
```

Matemáticamente, la expresión se evalúa a `7`, porque el operador de multiplicación tiene mayor precedencia que la suma. Sin embargo, algunas calculadoras darían como respuesta `9`. Esto ilustra que una simple evaluación de izquierda a derecha no es suficiente. No podemos empezar evaluando `1+2`. Considere también las expresiones

En una *expresión infija*, un operador binario tiene argumentos a su izquierda y a su derecha.

```
10 - 4 - 3
2 ^ 3 ^ 3
```

donde `^` es el operador de exponenciación. ¿Qué resta y qué exponenciación debe evaluarse primero? Las restas se evalúan de izquierda a derecha, dando como resultado `3`. Por el contrario, las potencias se evalúan, generalmente, de derecha a izquierda, por lo que la expresión reflejaría 2^{3^3} en lugar de $(2^3)^3$. Es decir, la resta asocia de izquierda a derecha, mientras que la exponenciación asocia de derecha a izquierda. Todas estas posibilidades sugieren que la evaluación de una expresión como

Cuando hay varios operadores, la precedencia y la asociatividad determinan en qué orden se procesan las operaciones.

```
1 - 2 - 4 ^ 5 * 3 * 6 / 7 ^ 2 ^ 2
```

representaría un serio reto.

Si los cálculos se realizan con matemática entera (es decir, redondeando hacia abajo en la división), la respuesta sería `-8`. Para probar esto, introducimos paréntesis para ilustrar el orden correcto de los cálculos:

```
(1 - 2) - ( ( ( 4 ^ 5 ) * 3 ) * 6 ) / ( 7 ^ ( 2 ^ 2 ) )
```

Aunque los paréntesis desambiguan el orden de evaluación, es difícil argumentar que hacen más claro el mecanismo de evaluación. Resulta que una forma de expresiones diferente, denominada *expresión postfija*, proporciona un mecanismo directo de evaluación. Las siguientes secciones muestran cómo funciona. En la primera estudiamos la notación postfija, mostrando cómo las expresiones escritas con esta notación pueden evaluarse con un simple recorrido de izquierda a derecha. La siguiente sección muestra cómo las expresiones originales, que utilizan la notación habitual infija, se pueden convertir a notación postfija. Finalmente, se presenta un programa Java, que evalúa expresiones infijas que contengan operadores aditivos, multiplicativos y de exponenciación, así como paréntesis. Se utiliza un algoritmo de evaluación de expresiones con precedencia entre operadores.

11.2.1 Máquinas postfijas

Una *expresión postfija* puede evaluarse de la siguiente manera: los operandos se van apilando en una pila; tras encontrar un operador se desapilan sus operandos y se apila el resultado. Al finalizar la evaluación, la pila debería contener exactamente un valor, que representa el resultado.

Una *expresión postfija* está formada por una serie de operadores y operandos. Se evalúa utilizando una *máquina postfija*, en la siguiente forma: cuando se encuentra un operando, se apila en la pila; cuando se encuentra un operador, el número apropiado de operandos son desapilados de la pila, se evalúa la operación, y el resultado se apila de nuevo en la pila. Para operadores binarios, que son los más comunes, dos operandos son desapilados. Cuando la expresión postfija completa ha sido procesada, el resultado debería ser el único elemento en la pila. La notación postfija es una forma natural de evaluar expresiones pues con ella no son necesarias reglas de precedencia.

Presentamos a continuación un ejemplo sencillo. Consideremos la expresión postfija

```
1 2 3 * +
```

La evaluación procede de la siguiente manera: el `1`, el `2` y el `3` son apilados, en ese orden, en la pila. Para procesar el `*`, se desapilan los dos elementos superiores de la pila: esto es, el `3` y después el `2`. Observe que el primer elemento desapilado se convierte en el parámetro derecho del operador, y el segundo en el izquierdo; por tanto, los parámetros se obtienen de la pila en orden inverso al natural. Para la multiplicación, esto no importa, pero para la resta y la división, desde luego que sí. El resultado de la multiplicación es `6`, y es apilado en la pila. En este momento la cima de la pila es un `6`; y debajo hay un `1`. Para procesar el `+`, se desapilan el `6` y el `1`, y su suma, `7`, se apila. En este punto, la expresión se ha leído completamente, y la pila tiene sólo un elemento. Por tanto, la respuesta final es `7`.

Toda expresión infija válida puede convertirse a notación postfija. Por ejemplo, la expresión infija vista antes puede escribirse en notación postfija como

```
1 2 - 4 5 ^ 3 * 6 * 7 2 2 ^ ^ / -
```

La evaluación de una expresión postfija requiere un tiempo lineal.

La Figura 11.11 muestra los pasos utilizados por la máquina postfija para evaluarla. Cada paso involucra un apilamiento. Por tanto, ya que hay 9 operandos y 8 operadores, hay 17 pasos y 17 apilamientos. Claramente, el tiempo necesario para evaluar una expresión postfija es lineal.

El punto que falta por estudiar es un algoritmo que convierta notación infija en notación postfija. Una vez tengamos uno, tendremos un algoritmo para evaluar expresiones infijas.

El algoritmo de *análisis sintáctico de expresiones con precedencia entre operadores* convierte una expresión infija en una postfija, para luego poder evaluarla fácilmente.

11.2.2 Conversión de notación infija a postfija

El principio básico involucrado en el algoritmo de *análisis sintáctico de expresiones con precedencia entre operadores*, que convierte una expresión infija en una postfija, es el siguiente. Cuando se encuentra un operando, podemos pasarlo inmediatamente a la salida. Sin embargo, cuando encontramos un operador, no podemos pasarlo todavía a la salida, pues al efecto debemos esperar a encontrar su se-

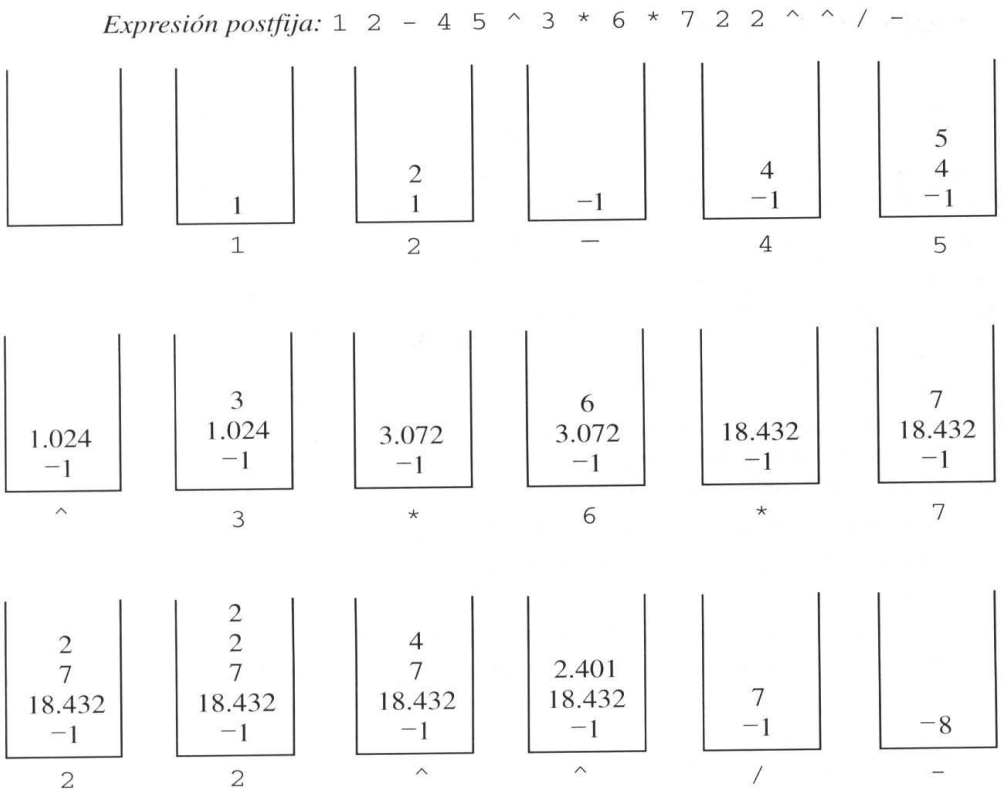


Figura 11.11 Pasos para la evaluación de una expresión postfija.

gundo operando. Por consiguiente, debemos guardarlo temporalmente en una estructura adecuada. Si consideramos una expresión como

```
1 + 2 * 3 ^ 4
```

la cual en notación postfija es

```
1 2 3 4 ^ * +
```

observamos que en ocasiones los operadores pueden aparecer en orden inverso a como aparecían en la expresión infija. Por supuesto, esto sólo es cierto si la precedencia de los operadores involucrados crece al ir de izquierda a derecha. Aún así, el hecho comentado sugiere que una pila es la estructura apropiada para almacenar los operadores pendientes. Siguiendo esta lógica, cuando encontramos un operador, debe ser colocado de alguna manera en la pila. El resto del algoritmo tiene que ver con cuándo los operadores entran y salen de la pila.

Consideramos ahora otra expresión infija más simple:

```
2 ^ 5 - 1
```

Cuando alcanzamos el operador `-`, el `2` y el `5` se han incorporado ya a la salida, y `^` está en la pila. Ya que `-` tiene menor precedencia que `^`, `^` tiene que aplicarse al `2` y al `5`. Por lo que debemos desapilar `^` y, en general, cualquier otro operador

Se utiliza una pila de operadores para almacenar los operadores vistos que no se han pasado a la salida.

Cuando se encuentra un operador en la entrada, los operadores de mayor prioridad (u operadores que asocien por la izquierda de la misma prioridad) se eliminan de la pila, lo que significa que deben ser aplicados. Tras ello apilamos el operador leído de la entrada.

que hubiese en la cima de la pila con mayor precedencia que -. En consecuencia, la expresión resultante postfija es

```
2 5 ^ 1 -
```

En general, cuando procesamos un operador de la entrada, debemos sacar de la pila aquellos operadores que deban ser procesados atendiendo a las reglas de precedencia y asociatividad.

Un segundo ejemplo, algo más complejo, lo representa la expresión infija

```
3 * 2 ^ 5 - 1
```

Cuando se alcanza el operador ^, el 3 y el 2 han sido incorporados a la salida, y el * está en la pila. Ya que ^ tiene mayor precedencia que *, no se desapila nada, apilándose encima el ^. Tras ello, el 5 se muestra inmediatamente, para luego encontrarnos con el -. Las reglas de precedencia nos dicen que desapilemos el ^, seguido del *. En este punto, no hay nada más que desapilar, por lo que dejamos de hacerlo, apilando el -. Después incorporamos a la salida el 1. Cuando se alcanza el final de la expresión, se deben sacar el resto de los operadores en la pila. La expresión postfija resultante es

```
3 2 5 ^ * 1 -
```

Antes de resumir el algoritmo, deben responderse algunas preguntas. En primer lugar, si el símbolo actual es un + y la cima de la pila es un +, ¿debería desapilarse el + de la pila, o debería quedarse ahí? La respuesta se obtiene decidiendo si el + encontrado en la entrada nos indica que el + de la pila cuenta ya con sus operandos. Ya que el + asocia de izquierda a derecha, la respuesta es afirmativa. Sin embargo, si se tratara del operador ^, que asocia de derecha a izquierda, la respuesta sería negativa. Por tanto, cuando nos topamos con dos operadores de igual precedencia, hay que examinar las normas de asociatividad para decidir correctamente, como muestran los siguientes ejemplos:

Expresión infija	Expresión postfija	Asociatividad
2 +3 + 4	2 3 + 4 +	Asociatividad por la izquierda: el + en la entrada es menor que el + de la pila.
2 ^ 3 ^ 4	2 3 4 ^ ^	Asociatividad por la derecha: el ^ en la entrada es mayor que el ^ de la pila.

Un paréntesis izquierdo se trata como un operador de máxima precedencia cuando es un símbolo de la entrada, pero como un operador de precedencia mínima cuando está en la pila. Así un paréntesis izquierdo sólo es eliminado por un paréntesis derecho.

¿Qué ocurre con los paréntesis? Un paréntesis izquierdo puede considerarse como un operador de máxima precedencia cuando está en la entrada, pero de precedencia mínima cuando está en la pila. En consecuencia, el paréntesis izquierdo de la entrada, se apilará siempre sin más. Cuando encontremos un paréntesis derecho en la entrada, desapilaremos hasta encontrar el correspondiente paréntesis izquierdo. Por otra parte, en la salida no aparecen paréntesis.

Presentamos ahora un resumen de los diferentes casos del algoritmo de análisis sintáctico de expresiones con precedencia de operadores. Todo lo que se desapila es mostrado en la salida, a excepción de los paréntesis.

- *Operandos*: Pasan inmediatamente a la salida.
- *Paréntesis derecho*: Desapilar símbolos hasta encontrar un paréntesis izquierdo.
- *Operador*: Desapilar todos los símbolos hasta que encontremos un símbolo de menor precedencia o un símbolo de igual precedencia con asociatividad por la derecha. Apilar entonces el operador encontrado.
- *Fin de la entrada*: Desapilar el resto de símbolos en la pila.

Como ejemplo, la Figura 11.12 muestra cómo el algoritmo procesa la expresión

```
1 - 2 ^ 3 ^ 3 - ( 4 + 5 * 6 ) * 7
```

Debajo de cada pila se muestra el símbolo leído. A la derecha, en negrita, se muestra el estado de la salida.

11.2.3 Implementación

La clase `Evaluador` hará el análisis y evaluará una expresión infija.

Contamos ahora con el fundamento teórico necesario para implementar una calculadora simple. Nuestra calculadora soportará sumas, restas, multiplicaciones, divisiones y exponenciaciones. Desarrollaremos una clase denominada `Evaluador`. Haremos antes una hipótesis simplificadora: no se permitirán números negativos. La

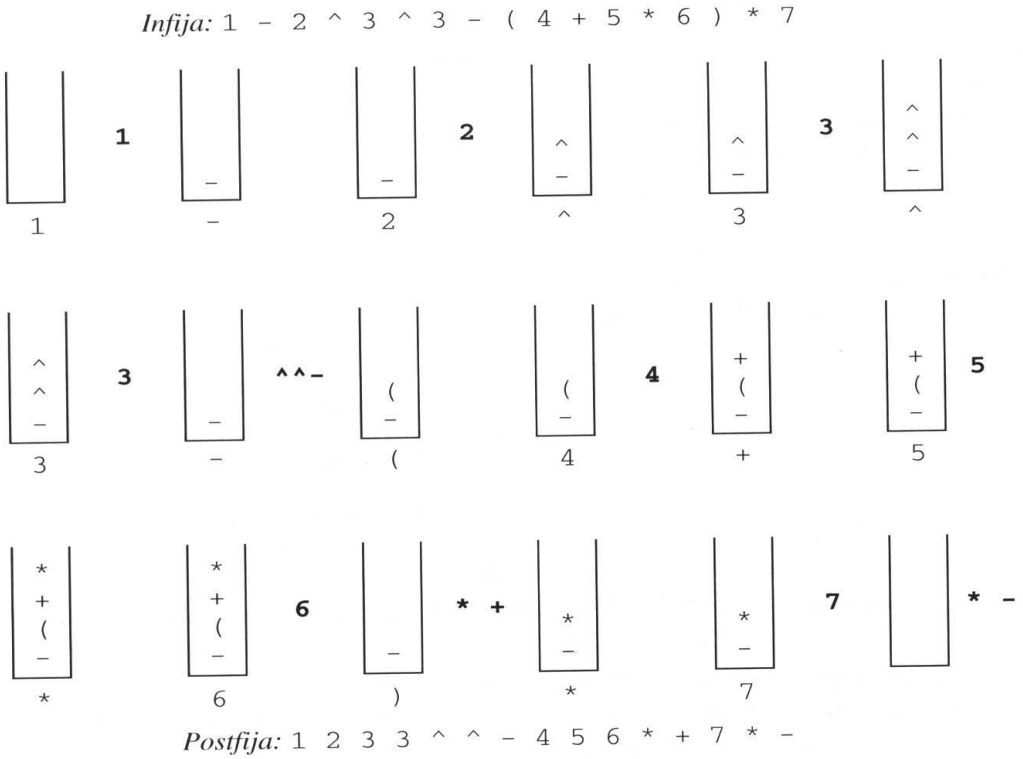


Figura 11.12 Conversión de notación infija o postfija.

distinción entre el operador menos binario y el menos unario requiere un trabajo extra en la rutina de análisis. También complica las cosas, al deber ser tratado como un operador que no es binario. Es cierto que la incorporación de operadores unarios no es demasiado difícil; sin embargo, el código extra no ilustra ningún concepto nuevo, por lo que se deja como ejercicio.

Necesitamos dos pilas: una pila de operadores y una pila para la máquina postfija.

La Figura 11.13 muestra el esqueleto de la clase `Evaluador`, utilizada para leer una cadena de la entrada. El algoritmo básico de evaluación requiere dos pilas. La

```
1  // Clase Evaluador: evalúa expresiones infijas
2  //
3  // CONSTRUCCIÓN: con un valor de tipo String
4  //
5  // ********************OPERACIONES PÚBLICAS*********************
6  // long obtenerValor( ) --> Devuelve el valor de una expresión infija
7  // ************************ERRORES*****************************
8  // Se realiza alguna comprobación de errores
9
10 import EstructurasDatos.*;
11 import Excepciones.*;
12 import java.io.*;
13 import java.util.StringTokenizer;
14
15 public class Evaluador
16 {
17     static final int EOL      = 0;
18     static final int VALOR    = 1;
19     static final int PAREN_A  = 2;
20     static final int PAREN_C  = 3;
21     static final int EXP      = 4;
22     static final int MULT     = 5;
23     static final int DIV      = 6;
24     static final int MAS      = 7;
25     static final int MENOS    = 8;
26
27     public Evaluador( String s )
28     {
29         pilaOper     = new PilaVec( );
30         pilaPostfija = new PilaVec( );
31         entrada = new StringTokenizer( s, "+*-/^() ", true );
32         pilaOper.apilar( new Integer( EOL ) );
33     }
34
35     public long obtenerValor( )
36       { /* Figura 11.15 */ }
37
38     private Pila pilaOper;     // Pila de operadores para la
                                     conversión
39     private Pila pilaPostfija; // Pila para la máquina postfija
40     StringTokenizer entrada;   // La cadena de caracteres
41     long valorActual;          // Operando actual
42     int ultimoToken;           // Último token leído
43
44     private int obtenerToken( )
45       { /* Figura 11.16 */ }
46     private void opBinario( int opCima )
47       { /* Figura 11.18 */ }
48     private void procesarToken( )
49       { /* Figura 11.20 */ }
50 }
```

Figura 11.13 Esqueleto de la clase `Evaluador`.

primera pila se utiliza para evaluar la expresión infija y generar la expresión postfija. Es la pila de operadores declarada en la línea 38. En vez de mostrar directamente por la salida la expresión postfija, mandamos cada símbolo a la máquina postfija, según se van generando. Necesitamos, por tanto, una segunda pila, que almacena operandos. Ésta es la pila de la máquina postfija, declarada en la línea 39. Observe que si no tuviéramos un mecanismo de programación genérica, tendríamos problemas, pues las dos pilas almacenan elementos de diferente tipo. La Figura 11.14 contiene dos métodos para acceder a las dos pilas, y realizar las conversiones de tipos necesarias. El resto de atributos son un objeto de la clase `StringTokenizer` utilizado para recorrer la entrada, y atributos para almacenar el token actual y, cuando el token es un operando, el valor del operando.

Las líneas de la 17 a la 25 generan una serie de constantes. El constructor, mostrado en las líneas de la 27 a la 33, construye las dos pilas y el objeto de la clase `StringTokenizer`. Los parámetros para realizar la división en tokens indican los símbolos considerados como delimitadores. Indican también que estos delimitadores son tokens que deben tratarse (en vez de meros delimitadores que se saltan)[1].

El único método visible públicamente es `obtenerValor`. Mostrado en la Figura 11.15, `obtenerValor` lee repetidamente un token y lo procesa hasta que detecta el final de línea. En ese momento, el elemento en la cima de la pila es la respuesta. La Figura 11.16 muestra la rutina `obtenerToken`. Comenzamos saltando todos los blancos (llamando recursivamente a `obtenerToken`). Si no hemos alcanzado el final de la línea, comprobamos si tenemos alguno de los operadores de un carácter, y si es así, devolvemos el token apropiado. En otro caso, se alcanza la línea 24, y como quiera que esperamos que lo que tengamos sea un operando, lo introducimos en `valorActual`.

```
1  /**
2   * Método interno que oculta la conversión de tipos.
3   */
4  private long cimaYDesapilarPostfija( ) throws
   DesbordamientoInferior
5  {
6      return ( (Long) ( pilaPostfija.cimaYDesapilar( ) ) ).
7                                                    longValue( );
8  }
9
10 /**
11  * Otro método interno que oculta la conversión de tipos.
12  */
13 private int cimaPilaOper( ) throws DesbordamientoInferior
14 {
15     return ( (Integer) ( pilaOper.cima( ) ) ).intValue( );
16 }
```

Figura 11.14 Métodos internos para ocultar la conversión de tipos en las pilas genéricas.

[1] El Apéndice C.3.2 describe con más detalle la clase `StringTokenizer`.

```
1  /**
2   * Rutina pública que realiza la evaluación. Examina la
3   * máquina postfija para ver si aparece un único resultado,
4   * y si es así, lo devuelve; en otro caso, se produce un error.
5   */
6  public long obtenerValor( )
7  {
8      long elResultado = 0;
9
10     do
11     {
12         ultimoToken = obtenerToken( );
13         procesarToken( );
14     } while( ultimoToken != EOL );
15
16     try
17       { elResultado = cimaYDesapilarPostfija( ); }
18     catch( DesbordamientoInferior e )
19     {
20         System.err.println( "¡Falta un operando!" );
21         return 0;
22     }
23
24     if( !pilaPostfija.esVacia( ) )
25         System.err.println( "Aviso: falta un operador!" );
26
27     return elResultado;
28 }
```

Figura 11.15 Rutina `obtenerValor`, para leer y procesar tokens y devolver el elemento en la cima de la pila.

```
1  /**
2   * Encuentra el siguiente token, saltando blancos, y lo devuelve.
3   * Para un token VALOR, coloca el valor en valorActual.
4   * Imprime un mensaje de error si no se reconoce la entrada.
5   */
6  private int obtenerToken( )
7  {
8      String s = "";
9
10     try
11       { s = entrada.nextToken( ); }
12     catch( java.util.NoSuchElementException e )
13       { return EOL; }
14
15     if( s.equals( " " ) ) return obtenerToken( );
16     if( s.equals( "^" ) ) return EXP;
17     if( s.equals( "/" ) ) return DIV;
18     if( s.equals( "*" ) ) return MULT;
19     if( s.equals( "(" ) ) return PAREN_A;
20     if( s.equals( ")" ) ) return PAREN_C;
21     if( s.equals( "+" ) ) return MAS;
22     if( s.equals( "-" ) ) return MENOS;
23
24     try
25       { valorActual = Long.parseLong( s ); }
26     catch( NumberFormatException e )
27     {
28         System.err.println( "Error" );
29         return EOL;
30     }
31     return VALOR;
32 }
```

Figura 11.16 Rutina `obtenerToken`, que devuelve el siguiente token de la cadena.

Las Figuras 11.17 y 11.18 muestran las rutinas necesarias para implementar la máquina postfija. `obtenerCima` devuelve y elimina el elemento de la cima de la pila postfija. La rutina `opBinario` aplica `opCima` (que se espera sea el elemento de la cima de la pila de operadores) a los dos elementos de la cima de la pila postfija y los reemplaza por el resultado obtenido. También desapila de la pila de operadores, dado que se ha completado el procesamiento de `opCima`.

```
1  /*
2   * cimaYDesapilar de la pila de la máquina postfija; devuelve el
3   * resultado. Si la pila está vacía, se genera un error.
4   */
5  private long obtenerCima( )
6  {
7      try
8        { return cimaYDesapilarPostfija( ); }
9      catch( DesbordamientoInferior e )
10       { System.err.println( "Falta un operando" ); }
11     return 0;
12 }
```

Figura 11.17 Rutina `obtenerCima`, que devuelve el elemento en la cima de la pila postfija y lo elimina.

```
1  /**
2   * Procesa un operador tomando dos elementos de la pila
3   * postfija, aplicando el operador, y apilando el resultado.
4   * Imprime un error si faltan paréntesis derechos o se divide por 0.
5   */
6  private void opBinario( int opCima )
7  {
8      if( opCima == PAREN_A )
9      {
10         System.err.println( "Paréntesis desequilibrados" );
11         try
12           { pilaOper.desapilar( ); }
13         catch( DesbordamientoInferior e ) { } // No puede ocurrir
14         return;
15     }
16     long lder = obtenerCima( );
17     long lizq = obtenerCima( );
18
19     if( opCima == EXP )
20         pilaPostfija.apilar( new Long( potencia( lizq, lder ) ) );
21     else if( opCima == MAS )
22         pilaPostfija.apilar( new Long( lizq + lder ) );
23     else if( opCima == MENOS )
24         pilaPostfija.apilar( new Long( lizq - lder ) );
25     else if( opCima == MULT )
26         pilaPostfija.apilar( new Long( lizq * lder ) );
27     else if( opCima == DIV )
28         if( lder != 0 )
29             pilaPostfija.apilar( new Long( lizq / lder ) );
30         else
31         {
32             System.err.println( "División por cero" );
33             pilaPostfija.apilar( new Long( lizq ) );
34         }
35     try
36       { pilaOper.desapilar( ); }
37     catch( DesbordamientoInferior e ) { }
38 }
```

Figura 11.18 Rutina `opBinario` para aplicar `opCima` a la pila postfija.

Una *tabla de precedencias* es utilizada para decidir qué operadores se han de eliminar de la pila. Los operadores que asocian por la izquierda tienen como precedencia en la pila una unidad más que la precedencia del símbolo en la entrada. Los operadores que asocian por la derecha se comportan justo al revés.

La Figura 11.19 declara una *tabla de precedencias*, que almacena la precedencia de operadores y es utilizada para decidir qué operadores se han de eliminar de la pila. Se introduce en memoria cuando se carga la clase `Evaluador`. Los operadores se listan en el mismo orden que las constantes básicas de la clase.

Queremos asignar un número a cada nivel de precedencia. Cuanto mayor es el número, mayor es la precedencia. Podríamos limitarnos a asignar a los operadores aditivos la precedencia 1, a los multiplicativos la precedencia 3, a la exponenciación la precedencia 5, y a los paréntesis la precedencia 99. Sin embargo, también necesitamos tener en cuenta la asociatividad. Para lograrlo, asignamos a cada operador un número que representa su precedencia cuando es un símbolo de la entrada, y otro que representa su precedencia cuando es un operador de la pila. Un operador que asocia por la izquierda tiene como precedencia en la pila una unidad más que la original, mientras que un operador que asocie por la derecha tiene precedencia en la entrada una unidad más que la original. Por tanto, la precedencia de un operador + que esté en la pila es 2.

Una consecuencia de esta regla es que cualquiera de los dos operandos que originalmente tengan diferente precedencia, siguen estando ordenados de la misma forma. Sin embargo, si encontramos sendos operadores aditivos en la pila y como símbolo de la entrada, el operador en la cima de la pila tendrá mayor precedencia, y por tanto será desapilado. Esto es lo que queremos para los operadores que asocian por la izquierda.

De forma similar, si un ^ está en la pila de operadores y también en la entrada, el operador de la pila tendrá menor precedencia, por lo que no será desapilado, lo que es correcto para un operador que asocia por la derecha. El token `VALOR` nunca se apila, por lo que su precedencia carece de significado. El token de fin de línea

```
1  class Precedencia
2  {
3      int simboloEntrada;
4      int cimaPila;
5
6      Precedencia( int simEnt, int simCima )
7      {
8          simboloEntrada = simEnt;
9          cimaPila       = simCima;
10     }
11 }
12         // Esto es parte de la clase Evaluador
13         // tablaPrec genera el orden de evaluación de los Tokens
14     static Precedencia [ ] tablaPrec = new Precedencia[ 9 ];
15     static
16     {
17         tablaPrec[ 0 ] = new Precedencia(   0, -1 ); // EOL
18         tablaPrec[ 1 ] = new Precedencia(   0,  0 ); // VALOR
19         tablaPrec[ 2 ] = new Precedencia( 100,  0 ); // PAREN_A
20         tablaPrec[ 3 ] = new Precedencia(   0, 99 ); // PAREN_C
21         tablaPrec[ 4 ] = new Precedencia(   6,  5 ); // EXP
22         tablaPrec[ 5 ] = new Precedencia(   3,  4 ); // MULT
23         tablaPrec[ 6 ] = new Precedencia(   3,  4 ); // DIV
24         tablaPrec[ 7 ] = new Precedencia(   1,  2 ); // MAS
25         tablaPrec[ 8 ] = new Precedencia(   1,  2 ); // MENOS
26     }
```

Figura 11.19 Tabla de precedencias utilizada para evaluar una expresión infija.

tiene la menor precedencia, de modo que se colocará en la pila (lo que se hace en el constructor) para que actúe como centinela. Si se trata como un operador que asocia por la derecha, se manejará en el caso de operadores generales.

El método que falta es `procesarToken`, mostrado en la Figura 11.20. Cuando encontramos un operando, se apila en la pila postfija. Si encontramos un paréntesis derecho, desapilamos y procesamos repetidamente el operador en la cima de la pila de operadores hasta que aparezca el paréntesis izquierdo (líneas de la 19 a la 21). El paréntesis izquierdo se desapila en la línea 23. (Observe que el test de la línea 22 se utiliza para evitar desapilar el centinela cuando falta el paréntesis izquierdo.) En caso contrario, tenemos el caso general de operador, que se describe con el código de las líneas de la 29 a la 34. En la Figura 11.21 se muestra una rutina `main` sencilla: repetidamente lee una línea de la entrada, instancia un objeto de la clase `Evaluador`, y calcula su valor.

```
1  /**
2   * Después de leer un token, se utiliza el algoritmo de análisis de
3   * expresiones con precedencia de operadores para evaluarlo.
4   * Se detecta la falta de paréntesis abiertos.
5   */
6  private void procesarToken( )
7  {
8      int opCima;
9
10     try
11     {
12         switch( ultimoToken )
13         {
14           case VALOR:
15             pilaPostfija.apilar( new Long( valorActual ) );
16             return;
17
18           case PAREN_C:
19             while( ( opCima = cimaPilaOper( ) ) != PAREN_A
20                    && opCima != EOL )
21                 opBinario( opCima );
22             if( opCima == PAREN_A )
23                 pilaOper.cima( ); // Eliminar paréntesis abierto
24             else
25                 System.err.println( "Falta paréntesis abierto" );
26             break;
27
28           default: // Caso de operador general
29             while( tablaPrec[ ultimoToken ].simboloEntrada <=
30                    tablaPrec[ opCima = cimaPilaOper( ) ].
31                                                    cimaPila )
32                 opBinario( opCima );
33             if( ultimoToken != EOL )
34                 pilaOper.apilar( new Integer( ultimoToken ) );
35             break;
36         }
37     }
38     catch( DesbordamientoInferior e ) { } // No puede ocurrir
39 }
```

Figura 11.20 Rutina `procesarToken` para procesar `ultimoToken` utilizando el algoritmo de análisis de expresiones con precedencia de operadores.

```
1  /**
2   * Rutina main sencilla, aunque un tanto chapucera
3   */
4  public static void main( String [ ] args )
5  {
6      String cad;
7      BufferedReader in = new BufferedReader( new
8                              InputStreamReader( System.in ) );
9
10     try
11     {
12         System.out.println( "Introduzca expresiones," +
13                                " una por línea:" );
14         while( ( cad = in.readLine( ) ) != null )
15         {
16             System.out.println( "Leída: " + cad );
17             Evaluador ev = new Evaluador( cad );
18             System.out.println( ev.obtenerValor( ) );
19             System.out.println( "Introduzca otra expresión:" );
20         }
21     }
22     catch( IOException e ) { }
23 }
```

Figura 11.21 Una rutina `main` simple, para evaluar expresiones repetidamente.

11.2.4 Árboles sintácticos de expresiones

En el *árbol sintáctico de una expresión* las hojas contienen los operandos y el resto de nodos los operadores.

La Figura 11.22 muestra un ejemplo de *árbol sintáctico de una expresión*. Las hojas del árbol sintáctico de una expresión son operandos, como constantes o nombres de variables, mientras que el resto de nodos contienen operadores. Este árbol particular es binario, porque todos los operadores que aparecen en la expresión lo son. Aunque éste es el caso más simple, es posible que los nodos tengan más de dos hijos, así como un único hijo, como ocurre en el caso del operador menos unario.

El árbol sintáctico de una expresión se evalúa aplicando el operador en la raíz a los valores obtenidos evaluando de forma recursiva los subárboles izquierdo y derecho. En el ejemplo anterior, el hijo izquierdo se evalúa a `(a+b)` y el hijo derecho a `(a-b)`. Por tanto, el árbol completo representa `((a+b)*(a-b))`. Es evidente que podemos producir una expresión infija (sobrecargada de paréntesis) produ-

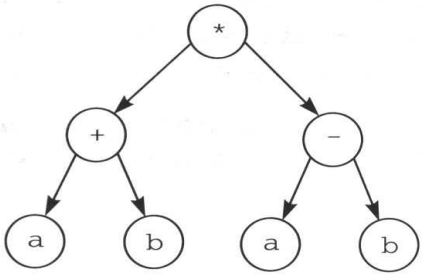


Figura 11.22 Árbol sintáctico de la expresión `(a + b) * (a - b)`.

ciendo recursivamente la expresión parentizada izquierda, incorporando entonces el operador en la raíz, y produciendo finalmente la expresión parentizada derecha. Esta estrategia general (izquierda, nodo, derecha) se denomina *recorrido en orden simétrico*. Este tipo de recorrido es fácil de recordar por el tipo de expresiones que produce.

Una segunda estrategia consiste en imprimir recursivamente el subárbol izquierdo, seguido del subárbol derecho, y terminando con el operador. Obtenemos de este modo la expresión en forma postfija. Este recorrido se denomina *recorrido en postorden del árbol*. Una tercera estrategia para recorrer el árbol tiene como resultado la expresión prefija. Todas estas estrategias se discutirán en el Capítulo 17. Los árboles sintácticos de expresiones (y sus generalizaciones) son estructuras de datos útiles en el diseño de compiladores, pues permiten manejar globalmente las expresiones, teniendo en cuenta su estructura lógica. Esto hace más sencilla la generación de código, y en algunos casos facilita significativamente los esfuerzos de optimización.

Se puede utilizar la evaluación recursiva del árbol abstracto de una expresión para producir expresiones equivalentes en forma infija, postfija y prefija.

Es de interés también la construcción del árbol sintáctico de una expresión a partir de su forma infija. Como ya se ha visto, siempre se puede convertir una expresión infija en una postfija, por lo que es suficiente ver cómo se puede construir un árbol sintáctico a partir de la expresión postfija. Y esto es sencillo. Se mantiene una pila de (referencias a) árboles. Cuando nos encontramos un operando, creamos un árbol con un único nodo y lo apilamos en la pila. Cuando encontramos un operador, los dos árboles en la cima de la pila se desapilan y combinan. En el nuevo árbol, la raíz es el operador, el hijo derecho es el primer árbol desapilado y el hijo izquierdo el segundo árbol desapilado. El resultado se apila de nuevo en la pila. Éste es, esencialmente, el mismo algoritmo que utilizábamos para la evaluación postfija, sustituyendo el cálculo de operadores binarios por la creación de árboles.

Los árboles sintácticos de expresiones pueden construirse a partir de una expresión postfija, de forma similar a como se realiza la evaluación postfija.

Resumen

Este capítulo examina dos usos de las pilas en el área general de lenguajes de programación y diseño de compiladores. Nos enseña que las pilas son muy potentes, aunque sean una estructura bastante simple. Las pilas pueden utilizarse para comprobar si una secuencia de símbolos está correctamente equilibrada. El algoritmo resultante consume un tiempo lineal y, lo que es igualmente importante, sólo hace un recorrido secuencial de la entrada. El análisis de expresiones con precedencia de operadores es una técnica que puede utilizarse para analizar expresiones infijas. También necesita un tiempo lineal y un solo recorrido de la entrada, utilizando dos pilas. Aunque las pilas almacenan valores de diferente tipo, el mecanismo de tipos genéricos nos permite utilizar una única implementación de las pilas para ambos tipos de objetos.

Elementos del juego

análisis de operadores con precedencia Un algoritmo que convierte una expresión infija en una postfija para evaluar la expresión infija.

análisis léxico El proceso de reconocimiento de tokens en una cadena de símbolos.

árbol sintáctico de una expresión Un árbol en el cual las hojas contienen operandos y el resto de nodos contiene operadores.

expresión infija Una expresión en la cual cada operador binario tiene sus argumentos a su izquierda y a su derecha. Cuando hay varios operadores, la precedencia y la asociatividad determinan en qué orden deben procesarse.

expresión postfija Una expresión que puede ser evaluada por la máquina postfija sin utilizar ninguna regla de precedencia.

máquina de estados Técnica común utilizada para analizar símbolos. En cada momento, la máquina está en un determinado estado. Cada carácter de entrada nos conduce a un nuevo estado. Cuando todo va bien, la máquina de estados acaba por alcanzar un estado en el cual un símbolo ha sido reconocido.

máquina postfija Máquina utilizada para evaluar una expresión postfija. El algoritmo que utiliza es el siguiente: los operandos se apilan en una pila y al encontrarnos con un operador desapilamos sus operandos y apilamos el resultado. Al final de la evaluación, la pila debería contener exactamente un elemento, que representa el resultado.

tabla de precedencias Una tabla utilizada para decidir qué operadores deben eliminarse de la pila de operadores. Los operadores que asocian por la izquierda tienen como precedencia en la pila de operadores una unidad más que su precedencia como símbolo de la entrada. Los operadores que asocian por la derecha se comportan justo al revés.

Errores comunes

1. Los errores en la entrada deben tratarse con el máximo cuidado. Es un error grave de programación ser negligente en este área.
2. En la rutina de comprobación de símbolos equilibrados, el tratamiento incorrecto de las comillas es un error habitual.
3. En el algoritmo de conversión de notación infija a postfija, la tabla de precedencias debe reflejar la precedencia y asociatividad correctas.

En Internet

Las dos aplicaciones están disponibles en el directorio **Chapter11**. Recomendamos que descargue el programa de comprobación de símbolos equilibrados; puede ayudarle a depurar programas en Java. Los nombres de los ficheros son:

JavaAnalizer.java	Traducido por `AnalizadorJava.java`, contiene el programa de comprobación de símbolos equilibrados. En el directorio **Part3** puede encontrarse una versión que incluye un generador de referencias cruzadas (*véase* Capítulo 12).
Evaluator.java	Traducido por `Evaluador.java`, contiene el programa para evaluar expresiones.

Ejercicios

Cuestiones breves

11.1. Muestre el resultado de ejecutar el programa de comprobación de símbolos desequilibrados sobre las siguiente entradas:

a) `}`
b) `( }`
c) `[ [ [`
d) `) (`
e) `[ ) ]`

11.2. Muestre las siguientes expresiones en notación postfija:

a) `1 + 2 - 3 ^ 4`
b) `1 ^ 2 - 3 * 4`
c) `1 + 2 * 3 - 4 ^ 5 + 6`
d) `( 1 + 2 ) * 3 - ( 4 ^ ( 5 - 6 ) )`

11.3. Para la expresión infija `a +b ^ c * d ^ e ^ f - g - h / ( i + j )`, haga lo siguiente:

a) Muestre cómo el algoritmo de análisis de expresiones con precedencia de operadores genera la correspondiente expresión postfija.
b) Muestre cómo la máquina postfija evalúa la expresión postfija resultante.
c) Dibuje el árbol sintáctico correspondiente.

Problemas teóricos

11.4. Para el programa de comprobación de símbolos equilibrados, explique cómo imprimir un mensaje de error que refleje la causa probable de cada error posible.

11.5. Explique, en términos generales, cómo se incorporarían operadores unarios al evaluador de expresiones. Asuma que los operadores unarios preceden a su operando, y que tienen máxima precedencia. Incluya una descripción de cómo serían reconocidos por la máquina de estados.

Problemas prácticos

11.6. Utilizar para la exponenciación el operador `^` puede confundir a los programadores Java (pues en dicho lenguaje se utiliza para denotar el operador O-exclusivo bit a bit). Rescriba la clase `Evaluador` tomando como operador de exponenciación `**`.

11.7. El evaluador infijo acepta ciertas expresiones ilegales, en las cuales los operadores están mal colocados. Se pide hacer lo siguiente:

a) ¿Cómo se evaluaría `1 2 3 + *`?
b) ¿Cómo podemos detectar estas ilegalidades?
c) Modifique la clase `Evaluador` para que las detecte.

Prácticas de programación

11.8. Modifique el evaluador de expresiones de modo que admita números negativos en la entrada.

11.9. Implemente un evaluador completo de expresiones de Java. Trate todos los operadores de Java que puedan aceptar constantes y tengan sentido aritmético.

11.10. Implemente un evaluador de expresiones Java que incluya variables. Suponga que existen como mucho 27 variables, es decir, de la `a` a la `z`, y que se puede asignar un valor a una variable con el operador `=` de mínima precedencia (como en Java).

11.11. Escriba un programa que lea una expresión infija y genere una expresión postfija.

11.12. Escriba un programa que lea una expresión postfija y genere una expresión infija.

11.13. Diseñe un applet que implemente una calculadora que utilice el teclado para la entrada.

11.14. Escriba un applet que ilustre cómo las dos pilas van cambiando durante la ejecución del evaluador de expresiones infijas.

11.15. Escriba una aplicación Java que proporcione una GUI con el programa de comprobación de símbolos desequilibrados. Utilice una caja de diálogo de ficheros, y coloque la salida en un área de texto.

Bibliografía

El algoritmo de análisis de expresiones con precedencia de operadores utilizado para convertir una expresión infija en una postfija fue descrito por primera vez en [3]. [1] y [2] son dos buenos libros sobre la construcción de compiladores.

1. A. V. Aho, R. Sethi, y J. D. Ullman, *Compiler Design*: *Principles*, *Techniques*, *and Tools*, Addison-Wesley, Reading, Mass. (1986).
2. C. N. Fischer y R. J. LeBlanc, *Crafting a Compiler with C*, Benjamin/Cummings, Redwood City, Calif. (1991).
3. R. W. Floyd, «Syntactic Analysis and Operator Precedence», *Journal of the ACM* **10:3** (1963), 316-333.