Un igualador de expresiones regulares

Brian Kernighan

LAS EXPRESIONES REGULARES SON NOTAS PARA DESCRIBIR PATRONES DE TEXTO y, en efecto, constituyen un lenguaje de propósito especial para la coincidencia de patrones. Aunque existen innumerables variantes, todas comparten la idea de que la mayoría de los caracteres en un patrón coinciden con ocurrencias literales de sí mismos, pero algunos metacaracteres tienen un significado especial, como \* para indicar algún tipo de repetición o [...] significar cualquier carácter de El conjunto dentro de los corchetes.

En la práctica, la mayoría de las búsquedas en programas como los editores de texto son palabras literales, por lo que las expresiones regulares suelen ser cadenas literales como print, que coincidirán con printf o sprint o papel de impresora en cualquier lugar. En los llamados comodines utilizados para especificar nombres de archivo en Unix y Windows, a \* coincide con cualquier número de caracteres, por lo que el patrón \* .c coincide con todos los nombres de archivo que terminan en .c. Hay muchas, muchas variantes de expresiones regulares, incluso en contextos donde uno esperaría que fueran iguales. El dominio de las expresiones regulares de Jeffrey Friedl (O’Reilly) es un estudio exhaustivo del tema.

Stephen Kleene inventó las expresiones regulares a mediados de la década de 1950 como una notación para autómatas finitos; de hecho, son equivalentes a autómatas finitos en lo que representan. Aparecieron por primera vez en un programa en la versión de Ken Thompson del editor de texto QED a mediados de la década de 1960. En 1967, Thompson solicitó una patente sobre un mecanismo para la coincidencia rápida de texto basado en expresiones regulares. La patente fue otorgada en 1971, una de las primeras patentes de software [EE. UU. Patente 3.568.156, Algoritmo de coincidencia de texto, 2 de marzo de 1971].

Las expresiones regulares se movieron de QED al editor de Unix ed, y luego a la herramienta grep por excelencia de Unix, que Thompson creó al realizar una cirugía radical en ed. Estos programas ampliamente utilizados ayudaron a las expresiones regulares a familiarizarse en toda la comunidad Unix temprana.

El emparejador original de Thompson fue muy rápido porque combinó dos ideas independientes. Una era generar instrucciones de la máquina sobre la marcha durante el emparejamiento para que funcionara a la velocidad de la máquina en lugar de por interpretación. El otro era llevar adelante todos los partidos posibles en cada etapa, por lo que no tenía que retroceder para buscar posibles partidos alternativos. En editores de texto posteriores que Thompson escribió, como ed, el código de coincidencia utilizó un algoritmo más simple que retrocedió cuando fue necesario. En teoría, esto es más lento, pero los patrones que se encuentran en la práctica rara vez implican retroceso, por lo que el algoritmo y el código ed y grep fueron lo suficientemente buenos para la mayoría de los propósitos.

Los siguientes emparejadores de expresiones regulares como egrep y fgrep agregaron clases más ricas de expresiones regulares, y se centraron en la ejecución rápida sin importar el patrón. Las expresiones regulares cada vez más elegantes se hicieron populares y se incluyeron no solo en las bibliotecas basadas en C, sino también como parte de la sintaxis de los lenguajes de script como Awk y Perl.

**La práctica de la programación**

En 1998, Rob Pike y yo escribíamos La práctica de la programación (Addison-Wesley). El último capítulo del libro, "Notación", recopiló una serie de ejemplos en los que una buena notación condujo a mejores programas y una mejor programación. Esto incluyó el uso de especificaciones de datos simples (printf, por ejemplo) y la generación de código a partir de tablas. Debido a nuestros antecedentes Unix y casi 30 años de experiencia con herramientas basadas en la notación de expresiones regulares, naturalmente queríamos incluir una discusión sobre expresiones regulares, y parecía obligatorio incluir también una implementación. Dado nuestro énfasis en las herramientas, también parecía mejor centrarse en la clase de expresiones regulares que se encuentran en grep, en lugar de, por ejemplo, las de comodines de shell, ya que también podríamos hablar sobre el diseño de grep en sí. El problema era que cualquier paquete de expresión regular existente era demasiado grande. El grep local tenía más de 500 líneas de largo (alrededor de 10 páginas de libros) e incrustaciones de percebes. Los paquetes de expresiones regulares de código abierto tienden a ser enormes, aproximadamente del tamaño de todo el libro, porque fueron diseñados para generalidad, flexibilidad y velocidad; ninguno era remotamente adecuado para la pedagogía. Le sugerí a Rob que encontráramos el paquete de expresión regular más pequeño que ilustrara las ideas básicas sin dejar de reconocer una clase de patrones útiles y no triviales. Idealmente, el código cabría en una sola página. Rob desapareció en su oficina. Como lo recuerdo ahora, emergió en no más de una hora o dos con las 30 líneas de código C que posteriormente aparecieron en el Capítulo 9 de The Practice of Programming. Ese código implementa un emparejador de expresiones regulares que maneja las siguientes construcciones.

Carácter Significado

c Coincide con cualquier carácter literal c.

. (punto) Coincide con cualquier personaje individual.

^ Coincide con el comienzo de la cadena de entrada.

$ Coincide con el final de la cadena de entrada. \* Coincide con cero o más ocurrencias del personaje anterior.

Esta es una clase bastante útil; en mi propia experiencia de usar expresiones regulares a diario, representa fácilmente el 95 por ciento de todas las instancias. En muchas situaciones, resolver el problema correcto es un gran paso hacia la creación de un hermoso programa. Rob merece un gran crédito por elegir un conjunto de características muy pequeño pero importante, bien definido y extensible entre un amplio conjunto de opciones.

La implementación de Rob en sí misma es un excelente ejemplo de código hermoso: compacto, elegante, eficiente y útil. Es uno de los mejores ejemplos de recursión que he visto, y muestra el poder de los punteros en C. Aunque en ese momento estábamos más interesados ​​en transmitir el importante papel de la buena notación para hacer que un programa sea más fácil de usar (y tal vez también más fácil de escribir), el código de expresión regular también ha sido una excelente manera de ilustrar algoritmos, estructuras de datos, pruebas, mejora del rendimiento y otros temas importantes.

Implementación

En The Practice of Programming, el comparador de expresiones regulares es parte de un programa independiente que imita grep, pero el código de expresión regular es completamente separable de su entorno. El programa principal no es interesante aquí; Al igual que muchas herramientas de Unix, lee su entrada estándar o una secuencia de archivos e imprime las líneas que contienen una coincidencia de la expresión regular.

Este es el código correspondiente:

/\* match: search for regexp anywhere in text \*/ int match(char \*regexp, char \*text)

{

if (regexp[0] == '^')

return matchhere(regexp+1, text);

do { /\* must look even if string is empty \*/

if (matchhere(regexp, text))

return 1;

} while (\*text++ != '\0');

return 0;

}

/\* matchhere: search for regexp at beginning of text \*/

int matchhere(char \*regexp, char \*text)

{

if (regexp[0] == '\0')

return 1;

if (regexp[1] == '\*')

return matchstar(regexp[0], regexp+2, text);

if (regexp[0] == '$' && regexp[1] == '\0') return \*text == '\0';

if (\*text!='\0' && (regexp[0]=='.' || regexp[0]==\*text))

return matchhere(regexp+1, text+1); return 0; }

/\* matchstar: search for c\*regexp at beginning of text \*/

int matchstar(int c, char \*regexp, char \*text)

{

do { /\* a \* matches zero or more instances \*/

if (matchhere(regexp, text))

return 1;

} while (\*text != '\0' && (\*text++ == c || c == '.')); return 0;

}

**Discusión**

La función match (regexp, text) prueba si existe una expresión regular en algún lugar del texto; devuelve 1 si se encuentra una coincidencia y 0 si no. Si hay más de una coincidencia, encuentra la más a la izquierda y la más corta.

La operación básica del partido es sencilla. Si el primer carácter de la expresión regular es ^ (una coincidencia anclada), cualquier posible coincidencia debe ocurrir al comienzo de la cadena. Es decir, si la expresión regular es ^ xyz, coincide con xyz solo si xyz aparece al comienzo del texto, no en el medio. Esto se prueba haciendo coincidir el resto de la expresión regular con el texto que comienza en el principio y en ningún otro lugar. De lo contrario, la expresión regular podría coincidir en cualquier lugar dentro de la cadena. Esto se prueba haciendo coincidir el patrón con cada posición de carácter del texto a su vez. Si hay varias coincidencias, solo se identificará la primera (más a la izquierda). Es decir, si la expresión regular es xyz, coincidirá con la primera aparición de xyz independientemente de dónde ocurra.

Observe que avanzar sobre la cadena de entrada se realiza con un ciclo do-while, una construcción relativamente inusual en los programas en C. La ocurrencia de un tiempo de espera en lugar de un tiempo siempre debería plantear una pregunta: ¿por qué no se prueba la condición de terminación del bucle al comienzo del bucle, antes de que sea demasiado tarde, en lugar de al final después de que se haya hecho algo? Pero la prueba es correcta aquí: dado que el operador \* permite coincidencias de longitud cero, primero tenemos que verificar si es posible una coincidencia nula.

La mayor parte del trabajo se realiza en la función matchhere (regexp, text), que prueba si la expresión regular coincide con el texto que comienza aquí. La función matchhere opera al intentar hacer coincidir el primer carácter de la expresión regular con el primer carácter del texto. Si la coincidencia falla, no puede haber coincidencia en esta posición de texto y la coincidencia aquí devuelve 0. Sin embargo, si la coincidencia tiene éxito, es posible avanzar al siguiente carácter de la expresión regular y al siguiente carácter del texto. Esto se hace llamando a matchhere de forma recursiva.

La situación es un poco más complicada debido a algunos casos especiales y, por supuesto, a la necesidad de detener la recurrencia. El caso más fácil es que si la expresión regular está al final (regexp [0] == '\ 0'), todas las pruebas anteriores han tenido éxito y, por lo tanto, la expresión regular coincide con el texto. Si la expresión regular es un carácter seguido de un \*, se llama a matchstar para ver si el cierre coincide. La función matchstar (c, regexp, text) intenta hacer coincidir las repeticiones del carácter de texto c, comenzando con cero repeticiones y contando hacia arriba, hasta que encuentra una coincidencia con el resto del texto o falla y concluye que hay Sin coincidencia. Este algoritmo identifica una "coincidencia más corta", que está bien para la coincidencia de patrones simple como en grep, donde lo único que importa es encontrar una coincidencia lo más rápido posible. Una "coincidencia más larga" es más intuitiva y es casi seguro que sea mejor para un editor de texto donde se reemplazará el texto coincidente. La mayoría de las bibliotecas modernas de expresiones regulares ofrecen ambas alternativas, y The Practice of Programming presenta una variante simple de matchstar para este caso, que se muestra a continuación. Si la expresión regular consiste en un $ al final de la expresión, el texto coincide solo si también está al final:

if (regexp[0] == '$' && regexp[1] == '\0')

return \*text == '\0';

De lo contrario, si no estamos al final de la cadena de texto (es decir, \* texto! = '\ 0'), y si el primer carácter de la cadena de texto coincide con el primer carácter de la expresión regular, hasta ahora todo bien; continuamos para probar si el siguiente carácter de la expresión regular coincide con el siguiente carácter del texto haciendo una llamada recursiva a matchhere. Esta llamada recursiva es el corazón del algoritmo y la razón por la cual el código es tan compacto y limpio.

Si todos estos intentos de coincidencia fallan, no puede haber coincidencia en este punto entre la expresión regular y el texto, por lo que la coincidencia aquí devuelve 0.

Este código utiliza punteros en C de forma intensiva. En cada etapa de la recursión, si algo coincide, la llamada recursiva que sigue utiliza aritmética de puntero (por ejemplo, regexp + 1 y texto + 1) para que la función posterior se llame con el siguiente carácter de la expresión regular y del texto. La profundidad de recursión no es más que la longitud del patrón, que en uso normal es bastante corto, por lo que no hay peligro de quedarse sin espacio.

**Alternativas**

Este es un código muy elegante y bien escrito, pero no es perfecto. ¿Qué podríamos hacer de manera diferente? Podría reorganizar matchhere para tratar con $ before \*. Aunque no hace ninguna diferencia aquí, se siente un poco más natural, y una buena regla es hacer casos fáciles antes que los difíciles.

En general, sin embargo, el orden de las pruebas es crítico. Por ejemplo, en esta prueba de matchstar:

    } while (\* text! = '\ 0' && (\* text ++ == c || c == '.'));

debemos avanzar sobre un carácter más de la cadena de texto sin importar qué, por lo que el incremento en text ++ siempre debe realizarse.

Este código tiene cuidado con las condiciones de terminación. En general, el éxito de una coincidencia está determinado por si la expresión regular se agota al mismo tiempo que el texto. Si se agotan juntos, eso indica una coincidencia; si uno se agota antes que el otro, no hay coincidencia. Esto es quizás más obvio en una línea como:

if (regexp[0] == '$' && regexp[1] == '\0') return \*text == '\0';

La versión de matchstar que implementa la coincidencia más larga a la izquierda comienza identificando una secuencia máxima de ocurrencias del carácter de entrada c. Luego usa matchhere para intentar extender la coincidencia al resto del patrón y al resto del texto. Cada falla reduce el número de cs en uno e intenta nuevamente, incluido el caso de cero ocurrencias:



/\* matchstar: leftmost longest search for c\*regexp \*/ int matchstar(int c, char \*regexp, char \*text) { char \*t;

for (t = text; \*t != '\0' && (\*t == c || c == '.'); t++) ; do { /\* \* matches zero or more \*/ if (matchhere(regexp, t)) return 1; } while (t-- > text); return 0; }

Considere la expresión regular (. \*), Que coincide con el texto arbitrario entre paréntesis. Dado el texto de destino:

    para (t = texto; \* t! = '\ 0' && (\* t == c || c == '.'); t ++)

una coincidencia más larga desde el principio identificará toda la expresión entre paréntesis, mientras que una coincidencia más corta se detendrá en el primer paréntesis derecho. (Por supuesto, una coincidencia más larga que comienza desde el segundo paréntesis izquierdo se extenderá hasta el final del texto).

Basándose en eso

El propósito de La práctica de la programación era enseñar una buena programación. En el momento en que se escribió el libro, Rob y yo todavía estábamos en los Laboratorios Bell, por lo que no teníamos experiencia de primera mano sobre cómo se utilizaría mejor el libro en el aula. Ha sido gratificante descubrir que parte del material funciona bien en las clases. He usado este código desde 2000 como un vehículo para enseñar puntos importantes sobre la programación.

Primero, muestra cómo la recursividad es útil y conduce a un código limpio en un entorno novedoso; ni siquiera es otra versión de Quicksort (¡o factorial!), ni es una especie de caminata por el árbol.

También es un buen ejemplo para los experimentos de rendimiento. Su rendimiento no es muy diferente de las versiones del sistema de grep, lo que demuestra que la técnica recursiva no es demasiado costosa y que no vale la pena intentar ajustar el código.

Por otro lado, también es una buena ilustración de la importancia de un buen algoritmo. Si un patrón incluye varias secuencias. \*, La implementación sencilla requiere una gran cantidad de retroceso y, en algunos casos, se ejecutará muy lentamente.

El grep estándar de Unix tiene las mismas propiedades de retroceso. Por ejemplo, el comando:

    grep 'a. \* a. \* a. \* a.a'

Tarda unos 20 segundos en procesar un archivo de texto de 4 MB en una máquina típica.

Una implementación basada en la conversión de un autómata finito no determinista en un autómata determinista, como en egrep, tendrá un rendimiento mucho mejor en casos difíciles; puede procesar el mismo patrón y la misma entrada en menos de una décima de segundo, y el tiempo de ejecución en general es independiente del patrón.

Las extensiones a la clase de expresión regular pueden formar la base de una variedad de tareas. Por ejemplo:

1. Agregue otros metacaracteres, como + para una o más ocurrencias del carácter anterior, o? para cero o uno coincide. Agregue alguna forma de citar metacaracteres, como \ $ para representar una ocurrencia literal de $.

2. Separe el procesamiento de expresiones regulares en una fase de compilación y una fase de ejecución. La compilación convierte la expresión regular en una forma interna que simplifica el código de coincidencia o permite que la coincidencia posterior se ejecute más rápido. Esta separación no es necesaria para la clase simple de expresiones regulares en el diseño original, pero tiene sentido en aplicaciones similares a grep donde la clase es más rica y la misma expresión regular se usa para una gran cantidad de líneas de entrada.

3. Agregue clases de caracteres como [abc] y [0-9], que en notación grep convencional coinciden con a o b o c y un dígito, respectivamente. Esto se puede hacer de varias maneras, la más natural de las cuales parece ser reemplazar las variables char \* del código original con una estructura:

typedef struct RE { int type; /\* CHAR, STAR, etc. \*/ int ch; /\* the character itself \*/ char \*ccl; /\* for [...] instead \*/ int nccl; /\* true if class is negated [^...] \*/ } RE;

y modificando el código básico para manejar una matriz de estos en lugar de una matriz de caracteres. No es estrictamente necesario separar la compilación de la ejecución para esta situación, pero resulta ser mucho más fácil. Los estudiantes que siguen los consejos para precompilar en una estructura de este tipo siempre obtienen mejores resultados que aquellos que intentan interpretar una estructura de datos de patrones complicada sobre la marcha. Escribir especificaciones claras e inequívocas para las clases de caracteres es difícil, e implementarlas perfectamente es peor, ya que requiere una codificación tediosa y poco instructiva. He simplificado esta asignación a lo largo del tiempo, y hoy en día pido con frecuencia las abreviaturas de Perllike como \ d para dígitos y \ D para no dígitos en lugar de los rangos entre corchetes originales.

4. Use un tipo opaco para ocultar la estructura RE y todos los detalles de implementación. Esta es una buena manera de mostrar la programación orientada a objetos en C, que no admite mucho más que esto. En efecto, esto crea una clase de expresión regular que usa nombres de funciones como RE\_new () y RE\_match () para los métodos en lugar del azúcar sintáctico de un lenguaje orientado a objetos.

5. Modifique la clase de expresiones regulares para que sean como los comodines en varios shells: las coincidencias están implícitamente ancladas en ambos extremos, \* coincide con cualquier número de caracteres y? coincide con cualquier personaje individual. Se puede modificar el algoritmo o asignar la entrada al algoritmo existente.

6. Convierta el código a Java. El código original usa muy bien los punteros en C, y es una buena práctica descubrir las alternativas en un lenguaje diferente. Las versiones de Java usan String.charAt (indexación en lugar de punteros) o String.substring (más cercano a la versión de puntero). Ninguno de los dos parece tan claro como el código C, y ninguno es tan compacto. Aunque el rendimiento no es realmente parte de este ejercicio, es interesante ver que la implementación de Java se ejecuta aproximadamente seis o siete veces más lento que las versiones C.

7. Escriba una clase de contenedor que convierta las expresiones regulares de esta clase a las clases Pattern y Matcher de Java, que separan la compilación y la coincidencia de una manera bastante diferente. Este es un buen ejemplo del patrón Adaptador o Fachada, que pone una cara diferente en una clase o conjunto de funciones existente.

También he usado este código ampliamente para explorar técnicas de prueba. Las expresiones regulares son lo suficientemente ricas como para que las pruebas estén lejos de ser triviales, pero lo suficientemente pequeñas como para poder escribir rápidamente una colección sustancial de pruebas que se realizarán mecánicamente. Para extensiones como las que acabamos de enumerar, les pido a los estudiantes que escriban una gran cantidad de pruebas en un lenguaje compacto (otro ejemplo más de "notación") y que usen esas pruebas en su propio código; naturalmente, también uso sus pruebas en el código de otros estudiantes.

**Conclusión**

Me sorprendió lo compacto y elegante que era este código cuando Rob Pike lo escribió por primera vez: era mucho más pequeño y poderoso de lo que creía posible. En retrospectiva, uno puede ver una serie de razones por las cuales el código es tan pequeño.

Primero, las características están bien elegidas para ser las más útiles y para dar una mayor comprensión de la implementación, sin ningún tipo de lujos. Por ejemplo, la implementación de los patrones anclados ^ y $ requiere solo tres o cuatro líneas, pero muestra cómo tratar los casos especiales de manera limpia antes de manejar los casos generales de manera uniforme. La operación de cierre \* debe estar presente porque es una noción fundamental en las expresiones regulares y proporciona la única forma de manejar patrones de longitudes no especificadas. Pero no agregaría ninguna idea para proporcionar también + y?, Por lo que se dejan como ejercicios.

En segundo lugar, la recursión es una victoria. Esta técnica de programación fundamental casi siempre conduce a un código más pequeño, más limpio y más elegante que el equivalente escrito con bucles explícitos, y ese es el caso aquí. La idea de despegar un carácter coincidente del frente de la expresión regular y del texto, luego recurrir al resto, hace eco de la estructura recursiva de los ejemplos factoriales tradicionales o de longitud de cadena, pero en un entorno mucho más interesante y útil. Tercero, este código realmente usa el lenguaje subyacente con buenos resultados. Los punteros pueden ser mal utilizados, por supuesto, pero aquí se usan para crear expresiones compactas que expresan naturalmente la extracción de caracteres individuales y avanzar al siguiente carácter. La indexación de matriz o las subcadenas pueden lograr el mismo efecto, pero en este código, los punteros hacen un mejor trabajo, especialmente cuando se combinan con modismos C para autoincremento y conversión implícita de valores de verdad. No conozco otra pieza de código que haga tanto en tan pocas líneas al tiempo que proporciona una fuente tan rica de información e ideas adicionales.

El código más hermoso que nunca escribí

Jon Bentley

IONCE ESCUCHÓ UN PROGRAMADOR MAESTRO ALABADO CON LA FRASE, "Agrega función al borrar el código". Antoine de Saint-Exupéry, el escritor y aviador francés, expresó este sentimiento de manera más general cuando dijo: "Un diseñador sabe que ha logrado la perfección no cuando no hay nada más que agregar, sino cuando no hay nada más que quitar". En el software, el código más bello, las funciones más bellas y los programas más bellos a veces no existen.

Por supuesto, es difícil hablar sobre cosas que no están allí. Este capítulo intenta esta tarea desalentadora presentando un análisis novedoso del tiempo de ejecución del clásico programa Quicksort. La primera sección prepara el escenario al revisar Quicksort desde una perspectiva personal. La siguiente sección es la carne de este capítulo. Comenzaremos agregando un contador al programa, luego manipularemos el código para hacerlo más y más pequeño y aún más y más poderoso hasta que solo unas pocas líneas de código capturen completamente su tiempo de ejecución promedio. La tercera sección resume las técnicas y presenta un análisis particularmente sucinto del costo de los árboles de búsqueda binarios. Las dos secciones finales extraen ideas del capítulo para ayudarlo a escribir programas más elegantes.

**El código más hermoso que jamás escribí**

Cuando Greg Wilson describió por primera vez la idea de este libro, me pregunté cuál era el código más hermoso que jamás había escrito. Después de que esta deliciosa pregunta rodó por mi cerebro durante la mayor parte del día, me di cuenta de que la respuesta era fácil: Quicksort. Desafortunadamente, la única pregunta tiene tres respuestas diferentes, dependiendo de cómo se formule exactamente.

Escribí mi tesis sobre algoritmos de divide y vencerás, y descubrí que C.A.R. El Quicksort de Hoare ("Quicksort", Computer Journal 5) es sin lugar a dudas el abuelo de todos ellos. Es un algoritmo hermoso para un problema fundamental que se puede implementar en código elegante. Me encantó el algoritmo, pero siempre andaba de puntillas en su bucle más interno. Una vez pasé dos días depurando un programa complejo que se basaba en ese ciclo, y durante años copié cuidadosamente ese código cada vez que necesitaba realizar una tarea similar. Solucionó mis problemas, pero realmente no lo entendí.

Eventualmente aprendí un elegante esquema de partición de Nico Lomuto, y finalmente pude escribir un Quicksort que podía entender e incluso probar que era correcto. La observación de William Strunk Jr. de que "la escritura vigorosa es concisa" se aplica tanto al código como al inglés, así que seguí su advertencia de "omitir palabras innecesarias" (Los elementos del estilo). Finalmente reduje aproximadamente 40 líneas de código a una docena par. Entonces, si la pregunta es, "¿Cuál es el código pequeño más hermoso que hayas escrito?" mi respuesta es el Quicksort de mi libro Programming Pearls, Second Edition (Addison-Wesley). Esta función Quicksort, implementada en C, se muestra en el Ejemplo 3-1. Seguiremos estudiando y perfeccionando este ejemplo en la siguiente sección.

EXAMPLE 3-1.Quicksort function

void quicksort(int l, int u) { int i, m; if (l >= u) return; swap(l, randint(l, u)); m = l; for (i = l+1; i <= u; i++) if (x[i] < x[l]) swap(++m, i); swap(l, m); quicksort(l, m-1); quicksort(m+1, u); }

Este código ordena una matriz global x [n] cuando se llama con los argumentos quicksort (0, n-1). Los dos parámetros de la función son los índices de la submatriz a ordenar: l para low y u para upper. El intercambio de llamadas (i, j) intercambia el contenido de x [i] yx [j]. El primer intercambio elige aleatoriamente un elemento de partición uniformemente seleccionado entre l y u.

Programming Pearls contiene una derivación detallada y una prueba de corrección para la función de clasificación rápida. Durante el resto de este capítulo, asumiré que el lector está familiarizado con Quicksort en el nivel presentado en esa descripción y en la mayoría de los libros de texto de algoritmos elementales.

Si cambia la pregunta a "¿Cuál es el código más hermoso que ha escrito que se usó ampliamente?" mi respuesta es nuevamente un Quicksort. Un artículo que escribí con M. D. McIlroy ("Ingeniería de una función de clasificación", Software – Practice and Experience, Vol. 23, No. 11) describe un grave error de rendimiento en la venerable función Unix qsort. Nos propusimos construir una nueva función de ordenación de la biblioteca C, y consideramos muchos algoritmos diferentes para la tarea, incluidos Ordenar por combinación y Ordenar por montón. Después de comparar varias implementaciones posibles, nos decidimos por una versión del algoritmo Quicksort. Ese documento describe cómo diseñamos una nueva función que era más clara, más rápida y más robusta que sus competidores, en parte porque era más pequeña. El sabio consejo de Gordon Bell demostró ser cierto: "Los componentes más baratos, más rápidos y más confiables de un sistema informático son aquellos que no están allí". Esa función se ha utilizado ampliamente durante más de una década sin informes de fallas.

Considerando las ganancias que se podrían lograr al reducir el tamaño del código, finalmente me hice una tercera variante de la pregunta que comenzó este capítulo. "¿Cuál es el código más hermoso que nunca escribiste?" ¿Cómo pude lograr mucho con muy poco? La respuesta estuvo nuevamente relacionada con Quicksort, específicamente, el análisis de su desempeño. La siguiente sección cuenta esa historia.

**Más y más con menos y menos**

Quicksort es un algoritmo elegante que se presta a análisis sutiles. Alrededor de 1980, tuve una maravillosa discusión con Tony Hoare sobre la historia de su algoritmo. Me dijo que cuando desarrolló Quicksort por primera vez, pensó que era demasiado simple de publicar, y solo escribió su clásico artículo "Quicksort" después de que pudo analizar su tiempo de ejecución esperado. Es fácil ver que, en el peor de los casos, Quicksort puede tardar aproximadamente n2 en ordenar una matriz de n elementos. En el mejor de los casos, elige el valor medio como un elemento de partición y, por lo tanto, ordena una matriz en aproximadamente n lg n comparaciones. Entonces, ¿cuántas comparaciones usa en promedio para una matriz aleatoria de n valores distintos? El análisis de Hoare de esta pregunta es hermoso, pero desafortunadamente sobre las cabezas matemáticas de muchos programadores. Cuando enseñé Quicksort a estudiantes universitarios, me sentí frustrado porque muchos simplemente no "obtuvieron" la prueba, incluso después de un esfuerzo sincero. Ahora atacaremos ese problema experimentalmente. Comenzaremos con el programa de Hoare y finalmente terminaremos con un análisis cercano al suyo. Nuestra tarea es modificar el Ejemplo 3-1 del código Quicksort al azar para analizar el número promedio de comparaciones utilizadas para ordenar una matriz de entradas distintas. También intentaremos obtener una visión máxima con un código, tiempo de ejecución y espacio mínimos. Para determinar el número promedio de comparaciones, primero aumentamos el programa para contarlas. Para hacer esto, incrementamos las variables comp antes de la comparación en el ciclo interno (Ejemplo 3-2).

EXAMPLE 3-2.Quicksort inner loop instrumented to count comparisons

for (i = l+1; i <= u; i++) { comps++; if (x[i] < x[l]) swap(++m, i); }

Si ejecutamos el programa para un valor de n, veremos cuántas comparaciones toma esa ejecución en particular. Si repetimos eso para muchas ejecuciones sobre muchos valores de n, y analizamos los resultados estadísticamente, observaremos que, en promedio, Quicksort toma alrededor de 1.4 n lg n comparaciones para ordenar n elementos. Esa no es una mala manera de obtener información sobre el comportamiento de un programa. Trece líneas de código y algunos experimentos pueden revelar mucho. Una famosa cita atribuida a escritores como Blaise Pascal y T. S. Eliot afirma que "si hubiera tenido más tiempo, te habría escrito una carta más corta". Tenemos tiempo, así que experimentemos con el código para intentar crear un programa más corto (y mejor). Jugaremos el juego de acelerar ese experimento, tratando de aumentar la precisión estadística y el conocimiento de la programación. Debido a que el bucle interno siempre realiza comparaciones precisas de u – l, podemos hacer que el programa sea un poco más rápido contando esas comparaciones en una sola operación fuera del bucle. Este cambio produce el Quicksort que se muestra en el Ejemplo 3-3.

EXAMPLE 3-3.Quicksort inner loop with increment moved out of loop

comps += u-l; for (i = l+1; i <= u; i++) if (x[i] < x[l]) swap(++m, i);

Este programa ordena una matriz y cuenta el número de comparaciones utilizadas al hacerlo. Sin embargo, si nuestro objetivo es solo contar las comparaciones, realmente no necesitamos ordenar la matriz. El ejemplo 3-4 elimina el "trabajo real" de ordenar los elementos y mantiene solo el "esqueleto" de las diversas llamadas realizadas por el programa.

EXAMPLE 3-4.Quicksort skeleton reduced to counting

void quickcount(int l, int u) { int m; if (l >= u) return; m = randint(l, u); comps += u-l; quickcount(l, m-1); quickcount(m+1, u); }

Este programa funciona debido a la forma de "aleatorización" en la que Quicksort elige su elemento de partición, y porque se supone que todos los elementos son distintos. Este nuevo programa ahora se ejecuta en un tiempo proporcional a n, y aunque el Ejemplo 3-3 requería un espacio proporcional a n, el espacio ahora se reduce a la pila de recursión, que en promedio es proporcional a lg n.

Si bien los índices (I y U) de la matriz son críticos en un programa real, no importan en esta versión esqueleto. Podemos reemplazar estos dos índices con un único entero (n) que especifica el tamaño de la submatriz que se ordenará (consulte el Ejemplo 3-5).



EXAMPLE 3-5.Quicksort skeleton with single size argument

void qc(int n) { int m; if (n <= 1) return; m = randint(1, n); comps += n-1; qc(m-1); qc(n-m); }

Ahora es más natural reformular este procedimiento como una función de conteo de comparación que devuelve el número de comparaciones utilizadas por una ejecución aleatoria de Quicksort. Esta función se muestra en el Ejemplo 3-6.



EXAMPLE 3-6.Quicksort skeleton implemented as a function

int cc(int n) { int m; if (n <= 1) return 0; m = randint(1, n); return n-1 + cc(m-1) + cc(n-m); }

Los ejemplos 3-4, 3-5 y 3-6 resuelven el mismo problema básico, y lo hacen con el mismo tiempo de ejecución y uso de memoria. Cada sucesor mejora la forma de la función y, por lo tanto, es más claro y un poco más sucinto que su predecesor. Al definir elparadox del inventor (HowToSolveIt, Princeton University Press), George Pólya dice que "el plan más ambicioso puede tener más posibilidades de éxito". Ahora intentaremos explotar esa paradoja en el análisis de Quicksort. Hasta ahora hemos preguntado: "¿Cuántas comparaciones hace Quicksort en una serie de tamaño n?" Ahora haremos la pregunta más ambiciosa: "¿Cuántas comparaciones hace Quicksort, en promedio, para una matriz aleatoria de tamaño n?" Podemos extender el Ejemplo 3-6 para obtener el pseudocódigo en el Ejemplo 3-7.

EXAMPLE 3-7.Quicksort average comparisons as pseudocode

float c(int n) if (n <= 1) return 0 sum = 0 for (m = 1; m <= n; m++) sum += n-1 + c(m-1) + c(n-m) return sum/n

Si la entrada tiene un máximo de un elemento, Quicksort no utiliza comparaciones, como en el Ejemplo 3-6. Para n más grande, este código considera cada valor de partición m (desde el primer elemento hasta el último, cada uno igualmente probable) y determina el costo de la partición allí. Luego, el código calcula la suma de estos valores (resolviendo así recursivamente un problema de tamaño m-1 y un problema de tamaño n-m), y luego divide esa suma por n para devolver el promedio.

Si pudiéramos calcular este número, nuestros experimentos serían mucho más poderosos. En lugar de tener que realizar muchos experimentos con un solo valor de n para estimar la media, un solo experimento nos daría la media verdadera. Desafortunadamente, ese poder tiene un precio: el programa se ejecuta en un tiempo proporcional a 3n (es un ejercicio interesante, aunque autorreferencial, analizar ese tiempo de ejecución utilizando las técnicas descritas en este capítulo).

El ejemplo 3-7 toma el tiempo que hace porque calcula las sub-respuestas una y otra vez. Cuando un programa hace eso, a menudo podemos usar programación dinámica para almacenar las sub-respuestas para evitar volver a calcularlas. En este caso, presentaremos la tabla t [N + 1], en la que t [n] almacena c (n), y calcularemos sus valores en orden creciente. Dejaremos que N denote el tamaño máximo de n, que es el tamaño de la matriz que se ordenará. El resultado se muestra en el Ejemplo 3-8.

EXAMPLE 3-8.Quicksort calculation with dynamic programming

t[0] = 0 for (n = 1; n <= N; n++) sum = 0 for (i = 1; i <= n; i++) sum += n-1 + t[i-1] + t[n-i] t[n] = sum/n

Este programa es una transcripción aproximada del Ejemplo 3-7 y reemplaza c (n) con t [n]. Su tiempo de ejecución es proporcional a N2 y su espacio es proporcional a N. Uno de sus beneficios es que al final de la ejecución, la matriz t contiene los valores promedio verdaderos (no solo la estimación de las medias de muestra) para los elementos de matriz 0 a N. Esos valores se pueden analizar para obtener información sobre la forma funcional del número esperado de comparaciones utilizadas por Quicksort. Ahora simplificaremos más ese programa. El primer paso es mover el término n-1 fuera del ciclo, como se muestra en el Ejemplo 3-9.

EXAMPLE 3-9.Quicksort calculation with code moved out of the loop

t[0] = 0 for (n = 1; n <= N; n++) sum = 0 for (i = 1; i <= n; i++) sum += t[i-1] + t[n-i] t[n] = n-1 + sum/n

Ahora ajustaremos aún más el bucle explotando la simetría. Cuando n es 4, por ejemplo, el bucle interno calcula la suma:

t [0] + t [3] + t [1] + t [2] + t [2] + t [1] + t [3] + t [0]

En la secuencia de pares, los primeros elementos aumentan mientras que los segundos elementos disminuyen. Por lo tanto, podemos reescribir la suma como:

2 \* (t [0] + t [1] + t [2] + t [3])

Podemos usar esa simetría para obtener el Quicksort que se muestra en el Ejemplo 3-10.

EXAMPLE 3-10.Quicksort calculation with symmetry

t[0] = 0 for (n = 1; n <= N; n++) sum = 0 for (i = 0; i < n; i++) sum += 2 \* t[i] t[n] = n-1 + sum/n

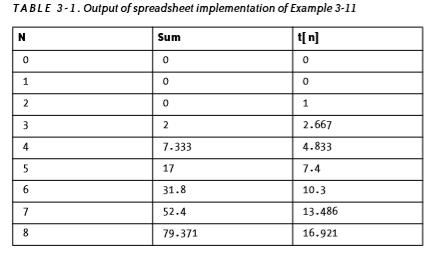
Sin embargo, este código es una vez más derrochador porque vuelve a calcular la misma suma una y otra vez. En lugar de agregar todos los términos anteriores, podemos inicializar la suma fuera del ciclo y agregar el siguiente término para obtener el Ejemplo 3-11.



EXAMPLE 3-11.Quicksort calculation with the inner loop removed

sum = 0; t[0] = 0 for (n = 1; n <= N; n++) sum += 2\*t[n-1] t[n] = n-1 + sum/n

Este pequeño programa es realmente útil. En un tiempo proporcional a N, produce una tabla de los tiempos de ejecución reales esperados de Quicksort para cada número entero de 1 a N. El ejemplo 3-11 es fácil de implementar en una hoja de cálculo, donde los valores están disponibles de inmediato para su posterior análisis. La Tabla 3-1 muestra las primeras filas



La primera fila de números en esta tabla se inicializa con las tres constantes del código. En la notación de hoja de cálculo, la siguiente fila de números (la tercera fila de la hoja de cálculo) se calcula utilizando las siguientes relaciones:

A3 = A2 + 1 B3 = B2 + 2 \* C2 C3 = A3-1 + B3 / A3

Arrastrando esas relaciones (relativas) hacia abajo completa la hoja de cálculo. Esa hoja de cálculo es un verdadero contendiente para "el código más hermoso que jamás haya escrito", utilizando el criterio de lograr mucho con solo unas pocas líneas de código.

¿Pero qué pasa si no necesitamos todos los valores? ¿Qué pasaría si preferiríamos analizar solo algunos de los valores en el camino (por ejemplo, todas las potencias de 2 de 20 a 232)? Aunque el Ejemplo 3-11 construye la tabla t completa, usa solo el valor más reciente de esa tabla.

Por lo tanto, podemos reemplazar el espacio lineal de la tabla t [] con el espacio constante de la variable t, como se muestra en el Ejemplo 3-12.



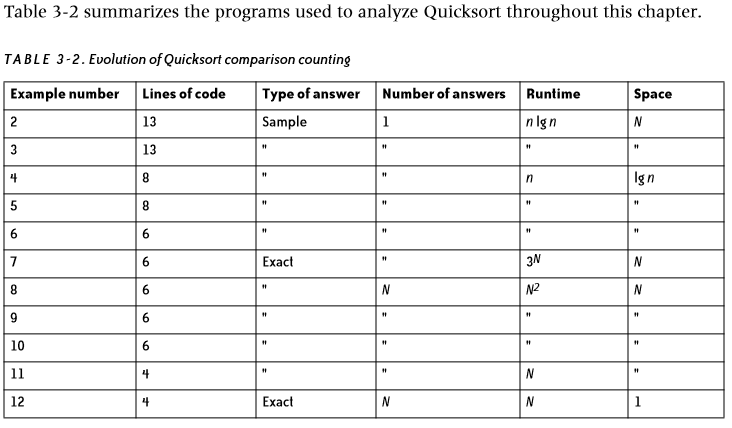
EXAMPLE 3-12.Quicksort calculation—final version

sum = 0; t = 0 for (n = 1; n <= N; n++) sum += 2\*t t = n-1 + sum/n

Luego podríamos insertar una línea adicional de código para probar la idoneidad de n e imprimir esos resultados según sea necesario.

Este pequeño programa es el paso final en nuestro largo camino. La observación de Alan Perlis es adecuada en consideración del camino que ha tomado este capítulo: "La simplicidad no precede a la complejidad, sino que la sigue" ("Epigramas sobre programación", Avisos de Sigplan, Vol. 17, Número 9).

**Perspectiva**

****

Cada paso individual en la evolución de nuestro código fue bastante sencillo; La transición de la muestra en el Ejemplo 3-6 a la respuesta exacta en el Ejemplo 3-7 es probablemente la más sutil. En el camino, a medida que el código se hizo más rápido y más útil, también disminuyó de tamaño. A mediados del siglo XIX, Robert Browning observó que "menos es más", y esta tabla ayuda a cuantificar una instancia de esa filosofía minimalista.

Hemos visto tres tipos de programas fundamentalmente diferentes. Los ejemplos 3-2 y 3-3 están trabajando en Quicksorts, instrumentados para contar comparaciones a medida que clasifican una matriz real. Los ejemplos 3-4 a 3-6 implementan un modelo simple de Quicksort: imitan una ejecución del algoritmo, sin hacer realmente el trabajo de clasificación. Los ejemplos 3-7 a 3-12 implementan un modelo más sofisticado: calculan el número promedio real de comparaciones sin rastrear ninguna ejecución en particular.

Las técnicas utilizadas para lograr cada programa se resumen a continuación:

• Ejemplos 3-2, 3-4, 3-7: Cambio fundamental de la definición del problema.

• Ejemplos 3-5, 3-6, 3-12: ligero cambio en la definición de la función.

• Ejemplo 3-8: Nueva estructura de datos para implementar programación dinámica.

Estas técnicas son típicas. A menudo podemos simplificar un programa preguntando: "¿Qué problema realmente necesitamos resolver?" o "¿Hay una mejor función para resolver ese problema?"

Cuando presenté este análisis a estudiantes universitarios, el programa finalmente se redujo a cero líneas de código y desapareció en una nube de humo matemático. Podemos reinterpretar el Ejemplo 3-7 como la siguiente relación de recurrencia:

Este es precisamente el enfoque adoptado por Hoare y luego presentado por D. E. Knuth en su clásico The Art of Computer Programming, Volume 3: Sorting and Searching (Addison-Wesley). Los trucos de programación de reexpresión y simetría que dan lugar al Ejemplo 3-10 nos permiten simplificar la parte recursiva para:

La técnica de Knuth para eliminar el signo de suma da (aproximadamente) el Ejemplo 3-11, que puede reexpresarse como un sistema de dos relaciones de recurrencia en dos incógnitas como:

Knuth utiliza la técnica matemática de un "factor sumador" para lograr la solución:

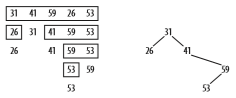
donde Hn denota el enésimo número armónico, 1 + 1/2 + 1/3 + ... 1 / n. Por lo tanto, hemos progresado sin problemas desde la experimentación en un programa al aumentarlo con sondas hasta un análisis completamente matemático de su comportamiento.

Con esta fórmula, terminamos nuestra búsqueda. Hemos seguido el famoso consejo de Einstein de "hacer que todo sea lo más simple posible, pero no más simple".

Un análisis de bonificación

Goethe dijo que "la arquitectura es música congelada". Exactamente en ese sentido, afirmo que "las estructuras de datos son algoritmos congelados". Y si congelamos el algoritmo Quicksort, obtenemos la estructura de datos de un árbol de búsqueda binario. La publicación de Knuth presenta esa estructura y analiza su tiempo de ejecución con una relación de recurrencia similar a la de Quicksort.

Si quisiéramos analizar el costo promedio de insertar un elemento en un árbol de búsqueda binario, podríamos comenzar con el código, aumentarlo para contar las comparaciones y luego realizar experimentos con los datos que recopilamos. Entonces podríamos simplificar ese código (y expandir su poder) de una manera que recuerda mucho a la sección anterior. Una solución más simple es definir un nuevo Quicksort que use un método de partición ideal que deje los elementos en el mismo orden relativo en ambos lados. Ese Quicksort es isomorfo a los árboles de búsqueda binarios, como se ilustra en la Figura 3-1.

****

Los cuadros de la izquierda muestran un Quicksort de partición ideal en progreso, y el gráfico de la derecha muestra el árbol de búsqueda binario correspondiente que se ha creado a partir de la misma entrada. Los dos procesos no solo hacen el mismo número de comparaciones, sino que hacen exactamente el mismo conjunto de comparaciones. Nuestro análisis anterior para el rendimiento promedio de la asignación aleatoria de Quicksort en un conjunto de elementos distintos nos da el número promedio de comparaciones para insertar elementos distintos permutados aleatoriamente en un árbol de búsqueda binario.

¿Qué es escribir?

En un sentido débil, "escribí" los ejemplos 3-2 a 3-12 del programa. Los escribí primero en notas garabateadas, luego en una pizarra frente a estudiantes de pregrado y, finalmente, en este capítulo. Derivé los programas sistemáticamente, pasé un tiempo considerable analizándolos y creo que son correctos. Sin embargo, aparte de la implementación de la hoja de cálculo del Ejemplo 3-11, nunca he ejecutado ninguno de los ejemplos como un programa de computadora.

En casi dos décadas en Bell Labs, aprendí de muchos maestros (y especialmente de Brian Kernighan, cuyo capítulo sobre la enseñanza de la programación aparece como el Capítulo 1 de este libro) que "escribir" un programa para exhibirlo en público implica mucho más que escribiendo símbolos Uno implementa el programa en código, lo ejecuta primero en algunos casos de prueba, luego construye andamios completos, controladores y una biblioteca de casos para superarlo sistemáticamente. Idealmente, uno incluye mecánicamente el código fuente compilado en el texto sin intervención humana. Escribí el Ejemplo 3-1 (y todo el código en Programming Pearls) en ese sentido fuerte.

Como punto de honor, quería mantener mi título honesto al nunca implementar los Ejemplos 3-2 a 3-12. Casi cuatro décadas de programación informática me han dejado profundamente

respeto por la dificultad de la embarcación (bueno, más precisamente, miedo a los insectos). Me comprometí al implementar el Ejemplo 3-11 en una hoja de cálculo, y agregué una columna adicional que daba la solución de forma cerrada. ¡Imagina mi deleite (y alivio) cuando los dos coinciden exactamente! Y entonces le ofrezco al mundo estos hermosos programas no escritos, con cierta confianza en su corrección, pero dolorosamente consciente de la posibilidad de un error no descubierto. Espero que la belleza profunda que encuentre en ellos no se vea afectada por manchas superficiales.

En mi incomodidad al presentar estos programas no escritos, me consuela la idea de Alan Perlis, quien dijo: "¿Es posible que el software no sea como cualquier otra cosa, que está destinado a ser descartado: que todo el punto es verlo?" como una burbuja de jabón?

Conclusión

La belleza tiene muchas fuentes. Este capítulo se ha concentrado en la belleza conferida por la simplicidad, la elegancia y la concisión. Los siguientes aforismos expresan este tema general:

• Esforzarse por agregar funciones eliminando el código.

• Un diseñador sabe que ha logrado la perfección no cuando no queda nada más que agregar, sino cuando no hay nada más que quitar. (Saint-Exupéry)

• En el software, el código más bello, las funciones más bellas y los programas más bellos a veces no existen.

• La escritura vigorosa es concisa. Omitir palabras innecesarias. (Strunk y blanco)

• Los componentes más baratos, más rápidos y más confiables de un sistema informático son aquellos que no están allí. (Campana)

• Esforzarse por hacer más y más con menos y menos.

• Si tuviera más tiempo, te habría escrito una carta más corta. (Pascal)

• La paradoja del inventor: el plan más ambicioso puede tener más posibilidades de éxito. (Pólya)

• La simplicidad no precede a la complejidad, sino que la sigue. (Perlis)

• Menos es más. (Browning)

• Haga todo lo más simple posible, pero no más simple. (Einstein)

• El software a veces debe verse como una burbuja de jabón. (Perlis)

• Busque belleza a través de la simplicidad.

Aquí termina la lección. Ve tú y haz lo mismo.

Para aquellos que desean pistas más concretas, aquí hay algunas ideas agrupadas en tres categorías principales.

Análisis de programas Una forma de obtener información sobre el comportamiento de un programa es instrumentarlo y luego ejecutarlo en datos representativos, como en el Ejemplo 3-2. Sin embargo, a menudo nos preocupa menos el programa en su conjunto que los aspectos individuales. En este caso, por ejemplo, consideramos solo el número de comparaciones que Quicksort usa en promedio e ignoramos muchos otros aspectos. Sedgewick ("El análisis de los programas Quicksort", Acta Informatica, Vol. 7) estudia cuestiones como el espacio que requiere y muchos otros componentes del tiempo de ejecución para una variedad de variantes de Quicksort. Al concentrarnos en los temas clave, podemos ignorar (por un tiempo) otros aspectos del programa. Uno de mis artículos, "Un estudio de caso en diseño de algoritmo aplicado" (IEEE Computer, Vol. 17, No. 2) describe cómo una vez enfrenté el problema de evaluar el rendimiento de una tira heurística para encontrar un recorrido de vendedor ambulante aproximado a través de N puntos en el cuadrado de la unidad. Calculé que un programa completo para la tarea podría tomar 100 líneas de código. Después de una serie de pasos similares en espíritu a lo que hemos visto en este capítulo, utilicé una simulación de doce líneas para dar mucha más precisión (y después de completar mi pequeña simulación, descubrí que Beardwood et al. ["The Shortest Path Through Muchos puntos ", Proc. Cambridge Philosophical Soc., Vol. 55] había reexpresado mi simulación como una integral doble, y por lo tanto había resuelto el problema matemáticamente unas dos décadas antes).

Pequeñas piezas de código Creo que la programación de computadoras es una habilidad práctica, y estoy de acuerdo con Pólya en que "adquirimos cualquier habilidad práctica por imitación y práctica". Los programadores que anhelan escribir código hermoso, por lo tanto, deben leer programas hermosos e imitar las técnicas que aprenden mientras escriben sus propios programas. Me parece que uno de los lugares más útiles para practicar es en pequeños fragmentos de código, digamos de solo una o dos docenas de líneas. Fue un trabajo duro pero muy divertido preparar la segunda edición de Programming Pearls. Implementé cada fragmento de código y trabajé para reducir cada uno a su esencia. Espero que otros disfruten leyendo el código tanto como yo disfruté escribirlo.

Sistemas de software Por especificidad, he descrito una pequeña tarea en detalles insoportables. Creo que la gloria de estos principios no radica en pequeños fragmentos de código, sino en grandes programas y enormes sistemas informáticos. Parnas ("Diseño de software para facilitar la extensión y la contracción", IEEE T. Ingeniería de Software, Vol. 5, Núm. 2) ofrece técnicas para reducir un sistema a lo esencial. Para una aplicabilidad inmediata, no olvide la visión profunda de Tom Duff: "Siempre que sea posible, robe el código".

**Expresiones de gratitud**

Estoy agradecido por los perspicaces comentarios de Dan Bentley, Brian Kernighan, Andy Oram y David Weiss.