Chương 5

GỠ RỐI

Trong bốn chương trước đã trình bày nhiều đoạn mã nguồn và chúng ta lầm tưởng rằng chúng đã chạy tốt ngay từ lần đầu tiên. Tất nhiên điều này không đúng bởi lẽ chúng đã có nhiều lỗi. Từ "bug" không có nguồn gốc cùng lúc với sự ra đời của lập trình viên, nhưng nó là một trong những thuật ngữ thông dụng trong lĩnh vực tin học. Tại sao phần mềm lại khó như vậy?

Sự phức tạp của chương trình liên quan tới cách thức tương tác giữa các thành phần của chương trình đó, đồng thời một phần mềm lại bao gồm nhiều thành phần và các tương tác giữa chúng. Nhiều kỹ thuật làm giảm mối liên hệ giữa các thành phần trong một chương trình nhờ thế làm giảm số lượng các thành phần tương tác; chẳng hạn như kỹ thuật che giấu thông tin, trừu tượng hóa và giao tiếp, và các đặc trưng của ngôn ngữ hỗ trợ cho các kỹ thuật này. Cũng có các kỹ thuật nhằm đảm bảo tính toàn vẹn của một bản thiết kế phần mềm – các tài liệu của chương trình, việc lập mô hình, phân tích các yêu cầu, kiểm tra hình thức – nhưng chưa có một kỹ thuật nào trong số này làm thay đổi cách thức xây dựng phần mềm; chúng chỉ giải quyết được một số vấn đề nhỏ. Thực tế cho thấy rằng lỗi luôn xuất hiện thông qua quá trình kiểm chứng và được loại bỏ bằng việc gỡ rối.

Các lập trình viên có khả năng tốt thường nhận thấy rằng họ tốn cùng lượng thời gian cho việc gỡ rối và việc viết chương trình, do đó họ cố gắng rút kinh nghiệm từ các lỗi trước đó. Mỗi một lỗi khi được tìm ra sẽ giúp bạn tránh lặp lại lỗi tương tự, cũng như giúp bạn nhận ra lỗi đó khi gặp lại.

Công viêc gỡ rối thường khó khăn và có thể tốn một lượng thời gian

không thể đoán trước được. Do đó, mục tiêu cần đạt được là tránh gây ra nhiều lỗi. Một số cách giúp làm giảm thời gian gỡ rối là: thiết kế cho thật tốt, phong cách lập trình tốt, kiểm tra các điều kiện biên, kiểm tra các khẳng định và tính đúng đắn trong mã nguồn, thiết kế các giao tiếp tốt, giới hạn việc sử dụng dữ liệu toàn cục và dùng các công cụ kiểm tra. Phương châm là phòng lỗi hơn chữa lỗi.

Vai trò của ngôn ngữ lập trình là gì? Động lực chính cho việc cải tiến các ngôn ngữ lập trình là cố gắng ngăn chặn các lỗi thông qua các đặc trưng của ngôn ngữ. Một số đặc trưng hạn chế bớt lỗi là: việc kiểm tra các giới hạn biên của các chỉ số, hạn chế hoặc không dùng con trỏ, bộ dọn dẹp, các kiểu dữ liệu chuỗi, xác định kiểu nhập/xuất, và kiểm tra kiểu dữ liệu chặt chẽ. Ngược lại, có một đặc tính dễ gây ra lỗi như: các lệnh `goto`, các biến toàn cục, các con trỏ trỏ đến những vùng không biết trước, và việc chuyển kiểu tự động. Các lập trình viên nên biết được các phần gây ra lỗi tiềm ẩn trong các ngôn ngữ lập trình và đặc biệt thận trọng khi sử dụng chúng. Họ cũng nên kích hoạt tất cả các chức năng kiểm tra của trình biên dịch và cũng nên quan tâm đến các cảnh báo.

Mỗi đặc trưng ngăn chặn lỗi của ngôn ngữ đều có ưu điểm và khuyết điểm. Nếu một ngôn ngữ cấp cao hơn tránh được các lỗi đơn giản một cách tự động, thì giá phải trả là nó dễ gây ra các lỗi ở mức cao hơn. Không một ngôn ngữ nào có thể ngăn hoàn toàn việc gây ra lỗi của bạn.

Phần lớn thời gian lập trình được dùng cho việc kiểm chứng và sửa lỗi mặc dù chúng ta không muốn như vậy. Chương này sẽ trình bày những phương pháp giúp bạn giảm thời gian gỡ rối một cách có hiệu quả; vấn đề kiểm chứng sẽ được trình bày trong Chương 6.

5.1. Trình gỡ rối

Các trình biên dịch cho các ngôn ngữ lớn thường kèm theo các trình gỡ rối phức tạp và được đóng gói như là một phần của môi trường phát triển phần mềm. Môi trường này tích hợp tất cả các công việc tạo lập, hiệu chỉnh mã nguồn, biên dịch, thực thi, và gỡ rối chương trình vào một hệ thống duy

nhất. Các trình gỡ rối với các giao diện đồ họa cho phép chạy chương trình từng bước qua từng lệnh hoặc từng hàm tại một thời điểm, dừng lại ở các dòng lệnh đặc biệt hay dừng lại khi xuất hiện một điều kiện đặc biệt. Chúng cũng hỗ trợ các công cụ cho phép định dạng và hiển thị các giá trị của các biến.

Trình gỡ rối có thể được kích hoạt một cách trực tiếp khi có một lỗi được biết trước. Một số trình gỡ rối kích hoạt một cách tự động khi xảy ra lỗi bất ngờ trong quá trình chạy chương trình. Thường để tìm ra được nơi mà chương trình bị hỏng khi đang chạy, chúng ta sẽ xem xét thứ tự các hàm đã được kích hoạt (theo vết ngăn xếp) và hiển thị các giá trị của các biến cục bộ và toàn cục. Những thông tin đó có thể đủ để xác định được lỗi. Nếu như vẫn không tìm ra được lỗi, thì dùng các điểm ngắt và việc chạy từng bước cho phép chúng ta chạy lại chương trình lỗi từng bước để xác định nơi đầu tiên gây ra lỗi.

Với một môi trường phù hợp và một người có kinh nghiệm thì một trình gỡ rối tốt là một phương tiện giúp gỡ rối hiệu quả và ít tốn kém. Với các công cụ mạnh như thế trong tay, tại sao ta không dùng chúng? Tại sao ta phải cần đến cả một chương nói về vấn đề gỡ rối?

Có nhiều lý do khách quan và chủ quan (dựa trên kinh nghiệm cá nhân) để trả lời cho câu hỏi trên. Một số ngôn ngữ ngoài luồng không có trình gỡ rối hoặc chỉ hỗ trợ các khả năng gỡ rối thô sơ. Các trình gỡ rối lại phụ thuộc hệ thống, nên có thể bạn sẽ không tìm thấy một trình gỡ rối quen thuộc khi làm việc trên một hệ thống khác. Các trình gỡ rối cũng hoạt động không tốt trên một số chương trình: các chương trình đa xử lý hay đa tiến trình, các hệ điều hành và các hệ thống phân tán thường phải được gỡ rối thông qua các tiếp cận ở mức thấp hơn. Trong các tình huống như vậy, bạn sẽ phải tự xoay sở lấy, bạn chỉ có thể dựa vào các câu lệnh xuất ra, kinh nghiệm và khả năng suy luận dựa trên mã nguồn của bạn.

Theo kinh nghiệm cá nhân, ta không nên dùng các trình gỡ rối bằng cách theo vết ngăn xếp hoặc lấy giá trị của một hoặc hai biến. Nguyên nhân là do chúng ta rất dễ đi lạc vào các chi tiết của các cấu trúc dữ liệu và các

luồng điều khiển phức tạp; chúng ta nhận thấy rằng việc chạy từng bước trong một chương trình thì không hiệu quả bằng việc suy nghĩ sâu hơn và thêm các lệnh xuất kết quả ra và tự kiểm tra mã nguồn tại những nơi có nguy cơ gây ra lỗi. Chạy qua các lệnh sẽ tốn nhiều thời gian hơn là duyệt qua các kết quả xuất ra của nó được đặt cẩn thận ở nơi nhạy cảm với lỗi. Quá trình quyết định vị trí đặt các lệnh xuất kết quả ra sẽ tốn ít thời gian hơn so với việc nhảy từng bước tới vùng có nguy cơ gây lỗi của mã nguồn ngay cả khi giả sử là ta đã biết trước được vùng gây lỗi. Và điều quan trọng là việc sửa lỗi các câu lệnh luôn tồn tại cùng với chương trình; còn các phiên bản của trình gỡ rối chỉ được dùng nhất thời.

Việc sử dụng trình gỡ rối một cách tự phát sẽ không có lợi. Sẽ hữu ích hơn nếu sử dụng trình gỡ rối để khám phá trạng thái của một chương trình khi nó có lỗi, tiếp đến suy nghĩ đến cách thức gây ra lỗi. Các trình gỡ rối có thể bí hiểm và khó dùng, và đặc biệt là đối với những người mới bắt đầu, chúng gây ra nhiều bối rối hơn là giúp đỡ họ. Nếu bạn đưa ra một câu hỏi sai, chúng có thể sẽ cho bạn một câu trả lời, tuy nhiên bạn có thể không biết được là nó đang bị sai.

Tuy nhiên, một trình gỡ rối có thể rất có giá trị, và bạn nên đưa một trình gỡ rối vào bộ công cụ gỡ rối của bạn. Nhưng nếu bạn không có một trình gỡ rối, hay nếu bạn phải đối phó với một vấn đề đặc biệt khó khăn thì các kỹ thuật trong chương này sẽ giúp bạn gỡ rối hiệu quả và đỡ tốn kém. Chúng cũng giúp bạn sử dụng hiệu quả hơn trình gỡ rối vì chúng tập trung chủ yếu vào việc cung cấp cho bạn các suy luận về lỗi và các nguyên nhân có thể gây ra lỗi.

5.2. Có đầu mối, phát hiện ra lỗi dễ dàng

Khi chương trình bị hỏng, hay xuất hiện các thông báo vô nghĩa, hoặc bị treo. Vì sao lại như vậy? Những người bắt đầu thường có khuynh hướng đổ lỗi cho trình biên dịch, thư viện, hay bất kỳ một nguyên nhân nào khác thay vì phải xem lại mã nguồn của họ. Các lập trình viên kinh nghiệm cũng thích làm như thế, nhưng họ biết rằng thật ra hầu hết các lỗi là do

chính các sai sót của họ.

May mắn là hầu hết các lỗi đều đơn giản và có thể được tìm thấy bằng các kỹ thuật đơn giản. Hãy khảo sát các đầu mối của việc xuất ra kết quả có lỗi và cố gắng suy ra nguyên nhân gây ra nó. Hãy quan sát bất kỳ việc gỡ rối nào của thao tác xuất ra kết quả trước khi xảy ra lỗi, nếu có thể hãy theo vết ngăn xếp từ trình gỡ rối. Bây giờ bạn có được một số thông tin về lỗi, và nơi xảy ra lỗi. Tạm dừng để ngẫm nghĩ. Lỗi xảy ra như thế nào? Suy luận ngược trở lại trạng thái của chương trình bị hỏng để xác định nguyên nhân gây ra lỗi này.

Gỡ rối liên quan đến việc lập luận lùi, giống như việc tìm kiếm các bí mật của một vụ ám sát. Một số vấn đề không thể xảy ra và chỉ có những thông tin xác thực mới đáng tin cậy. Do đó, chúng ta phải suy nghĩ ngược từ kết quả để khám phá ra các nguyên nhân. Khi ta có lời giải thích đầy đủ, ta sẽ biết được vấn đề cần sửa và có thể phát hiện ra một số vấn đề khác mà ta chưa dự trù từ trước.

Tìm các lỗi tương tự

Hãy tự hỏi liệu đây có phải là một lỗi tương tự hay không? "Tôi đã gặp qua rồi" thường là câu khởi đầu cho việc hiểu vấn đề, hay thậm chí là đã biết toàn bộ câu trả lời. Các lỗi phổ biến thì có các đặc điểm phân biệt. Chẳng hạn như, các lập trình viên C thiếu kinh nghiệm thường viết

```
?       int n;
?       scanf("%d", n);
```

thay vì

```
int n;
scanf("%d", &n);
```

và đây là một ví dụ đặc trưng về nguyên nhân gây ra một cố gắng truy cập tới bộ nhớ ngoài phạm vi khi đọc dữ liệu vào. Người dạy C sẽ nhận ra vấn

đề này ngay lập tức.

Các chuyển đổi kiểu và các kiểu không hợp lý trong hàm `printf` và `scanf` là một thường là nguyên nhân gây ra các lỗi đơn giản.

```
?     int n = 1;
?     double d = PI;
?     printf("%d %f\n", d, n);
```

Một lỗi khác thường thấy nữa là sử dụng định dạng `%f` thay vì dùng định dạng `%lf` để đọc một giá trị có kiểu `double` bằng hàm `scanf`. Một vài trình biên dịch bắt các lỗi như thế bằng cách xác minh rằng các kiểu của các đối số của hàm `scanf` và `printf` phải phù hợp với các định dạng của chúng: nếu tất cả các cảnh báo được bật lên, đối với hàm `prinf` ở trên, thì trình biên dịch sẽ thông báo như sau:

```
x.c:9: warning: int format, double arg (arg 2)
x.c:9: warning: double format, different type arg (arg 3)
```

Không khởi tạo biến cục bộ dẫn đến một dạng lỗi đặc biệt khác, kết quả thường là một giá trị cực kỳ lớn, giá trị rác trước đó được lưu lại trong cùng vị trí bộ nhớ. Một số trình biên dịch sẽ cảnh báo bạn, mặc dù có thể bạn phải kích hoạt chức năng kiểm tra tại thời điểm biên dịch, và chúng có thể không bao giờ bắt hết tất cả các trường hợp. Bộ nhớ được trả về bởi các hàm cấp phát như `malloc`, `realloc`, và `new` cũng thường chứa giá trị rác; phải nhớ khởi tạo nó.

Kiểm tra sự thay đổi mới nhất

Thay đổi mới nhất là gì? Nếu bạn chỉ thay đổi mỗi lúc một vấn đề khi phát triển chương trình thì lỗi thường có khuynh hướng xảy ra hoặc nằm trong đoạn mã mới hoặc do đoạn mã mới tác động đến chương trình. Khảo sát cẩn thận các thay đổi mới sẽ có ích cho việc khoanh vùng. Nếu lỗi xuất hiện trong phiên bản mới và không xuất hiện trong phiên bản cũ thì đoạn mã mới chính là phần gây ra lỗi. Điều này có nghĩa là bạn nên giữ lại ít nhất là

phiên bản ngay trước đó của chương trình, mà bạn tin rằng nó chạy đúng, và như thế bạn có thể so sánh giữa các phiên bản với nhau. Nó cũng có nghĩa là bạn nên lưu lại việc sửa đổi và các lỗi được sửa, như thế bạn không phải xét lại các thông tin này trong khi cố gắng sửa lỗi. Các hệ thống quản lý mã nguồn và các cơ chế lưu lại quá trình sửa đổi sẽ rất hữu ích trong trường hợp này.

Tránh mắc cùng một lỗi hai lần

Sau khi sửa một lỗi, bạn phải nghĩ xem bạn có phạm lỗi như thế ở nơi nào khác nữa không. Xét đoạn chương trình minh họa sau:

```
?   for (i = 1; i < argc; i++) {
?           if (argv[i][0] != '-')  /*các chọn lựa được
kết thúc */
?                   break;
?           switch (argv[i][1]) {
?           case 'o':   /* tên tập tin output */
?                   outname = argv[i];
?                   break;
?           case 'f':
?                   from = atoi(argv[i]);
?                   break;
?           case 't':
?                   to = atoi(argv[i]);
?                   break;
?           ...
```

luôn có -o gắn vào trước tên tập tin xuất ra. Lỗi này được sửa dễ dàng, mã nguồn được viết lại như sau:

```
outname = &argv[i][2];
```

và sau đó lại phát hiện ra chương trình bị sai khi xử lý đối số có dạng -f123 một cách chính xác: giá trị số được chuyển đổi luôn là 0. Đây là lỗi giống như lần trước; trường hợp tiếp theo trong lệnh switch nên viết là:

```
from = atoi(&argv[i][2]);
```

Mã nguồn dễ có thể có các lỗi nếu như chúng ta viết cẩu thả vì thấy nó quen thuộc. Thậm chí khi mã nguồn đơn giản đến mức bạn có thể viết trong giấc ngủ thì cũng đừng có ngủ gật trong lúc viết.

Gỡ lỗi ngay khi gặp

Quá vội vã cũng có thể gây ảnh hưởng đến các tình huống khác. Đừng bỏ qua lỗi khi chúng xuất hiện; hãy sửa ngay, vì nó có thể không xuất hiện ra lại cho đến khi quá trễ để thực hiện điều này.

Theo vết ngăn xếp

Mặc dù các trình gỡ rối có thể kiểm tra các chương trình đang chạy, nhưng một trong các ứng dụng phổ biến nhất của chúng ta là kiểm tra trạng thái của một chương trình sau khi hỏng. Số dòng mã nguồn có lỗi (thông thường được lưu vết trong ngăn xếp) là một phần hữu ích nhất cung cấp thông tin gỡ rối; các giá trị bất thường của các đối số cũng là một đầu mối lớn (các con trỏ 0, các số nguyên có giá trị lớn mà lẽ ra nó phải có giá trị nhỏ hoặc nó có giá trị âm thay vì nó phải có giá trị dương, các chuỗi ký tự không thuộc bảng mẫu tự).

Sau đây là một ví dụ đặc trưng, dựa vào phần trình bày về việc sắp xếp trong Chương 2. Để sắp xếp một mảng các số nguyên, chúng ta nên gọi hàm qsort với hàm so sánh số nguyên icmp:

```
int arr[N];

qsort(arr, N, sizeof(arr[0]), icmp);
```

nhưng giả sử tên của hàm so sánh chuỗi scmp được truyền vào một cách vô ý như sau:

```
?       int arr[N];
?       qsort(arr, N, sizeof(arr[0]), scmp);
```

Trình biên dịch không thể nhận biết được sự không phù hợp kiểu ở đây, do đó sẽ xảy ra lỗi. Việc chạy chương trình sẽ bị hỏng do cố gắng truy cập tới một vị trí bộ nhớ không hợp lệ.

Đọc trước khi gõ vào

Một kỹ thuật gỡ rối hiệu quả nhưng không được ủng hộ lắm là đọc đoạn mã cẩn thận và bỏ ra một chút thời gian để suy nghĩ về nó mà không thực hiện thay đổi. Nếu như có một sự thôi thúc khiến chúng ta chạm ngay vào bàn phím và bắt đầu sửa đổi chương trình thì hãy thử xem có sửa được lỗi hay không? Nhưng khi chúng ta không biết được điều gì thực sự gây ra lỗi và sửa không đúng chỗ sẽ có nguy cơ gây ra các lỗi khác. Việc liệt kê các phần có khuynh hướng gây lỗi của chương trình ra giấy có thể cho chúng ta một cái nhìn khác hơn là xem trên màn hình, và khuyến khích chúng ta bỏ ra nhiều thời gian hơn cho việc ngẫm nghĩ. Không nên thực hiện các liệt kê như vậy cho các phần không có khuynh hướng gây ra lỗi. In ra toàn bộ chương trình sẽ phá vỡ cây cấu trúc vì rất khó để thấy được cấu trúc khi nó được trải qua nhiều trang và các liệt kê trở nên không còn hữu ích khi bạn bắt đầu soạn thảo lại.

Hãy nghỉ ngơi một chút; đôi khi những thứ được viết trong mã nguồn không đúng như những gì bạn nghĩ, và những lúc nghỉ giải lao giúp bạn tỉnh táo trở lại và khi quay trở lại viết bạn sẽ nhận ra điều đó.

Hãy chống lại sự thôi thúc gõ vào; hãy nên suy nghĩ nhiều.

Giải thích cho người khác về đoạn mã của bạn

Một kỹ thuật hiệu quả khác là giải thích cho người khác về đoạn mã của bạn. Điều này thường chỉ ra được lỗi của bạn. Đôi khi chỉ cần một vài câu mở đầu thì theo sau sẽ là một câu đầy bối rối: "Ồ! Tôi phát hiện ra lỗi rồi. Xin lỗi đã làm phiền anh". Cách này hoạt động đặc biệt hiệu quả; thậm chí bạn có thể giải thích cho những người không phải là lập trình viên.

5.3. Không có đầu mối, khó phát hiện ra lỗi

Làm cho lỗi xuất hiện lại

Bước đầu tiên phải chắc chắn rằng bạn có thể làm cho lỗi xuất hiện khi cần. Thật là phiền toái để bắt các lỗi không xảy ra trong khi chạy chương trình. Bỏ ra một ít thời gian để cấu trúc lại các thiết lập dữ liệu nhập và các đối số được cho là gây ra lỗi, tiếp đến gom các cách trên lại sao cho có thể chạy nó thông qua nút nhấn hay một vài cái gõ phím. Nếu nó là một lỗi khó, bạn có thể làm cho nó lặp đi lặp lại nhiều lần trong khi bạn tìm nguyên nhân gây ra nó, và như thế bạn sẽ tiết kiệm được thời gian bằng cách làm cho nó xuất hiện lại một cách dễ dàng.

Nếu không thể làm cho lỗi xuất hiện lại qua mỗi lần chạy thì cố gắng tìm nguyên nhân tại sao lại không được. Có phải lỗi sẽ nhạy cảm trên một tập các điều kiện nào đó hay không? Thậm chí nếu bạn không thể làm cho lỗi xuất hiện qua mỗi lần chạy, nếu bạn có thể làm giảm được thời gian chờ xử lý thì bạn sẽ tìm thấy lỗi nhanh hơn.

Nếu một chương trình có hỗ trợ việc xuất kết quả gỡ lỗi, thì hãy kích hoạt nó lên. Các chương trình giả lập như chương trình chuỗi *Markov* trong Chương 3 nên đưa vào các chọn lựa như thế để biết thêm nhiều thông tin chẳng hạn như việc chọn số của bộ phát sinh số ngẫu nhiên để kết quả xuất ra có thể được tái sử dụng; các tùy chọn khác nên cho phép việc tạo lại số được chọn. Nhiều chương trình thêm vào các tùy chọn và một ý tưởng hay nữa là thêm vào các công cụ tương tự vào chương trình của chính bạn.

Chia để trị

Có thể thu hẹp phạm vi hơn hoặc chỉ tập trung vào dữ liệu nhập gây lỗi cho chương trình không? Thu hẹp các khả năng bằng cách giới hạn lại tối đa dữ liệu nhập sao cho lỗi vẫn còn xuất hiện. Cần những sửa đổi gì để giải quyết được lỗi? Cố gắng tìm ra các trường hợp kiểm tra tiêu biểu tập trung vào lỗi. Mỗi trường hợp kiểm tra nên hướng đến một kết quả cụ thể để từ đó xác nhận hay từ chối một vấn đề đặc thù nào đó được cho là gây ra lỗi.

Tiến trình thực hiện theo cách tìm kiếm nhị phân. Giới hạn dữ liệu nhập còn một nửa và xem kết quả có còn sai nữa không; nếu không, quay về trạng thái ngay trước đó và kiểm tra trên nửa kia của dữ liệu nhập. Cùng một tiến trình tìm kiếm nhị phân có thể được áp dụng trên toàn bộ chương trình: loại bỏ một số phần của chương trình không liên quan tới lỗi và kiểm tra xem có còn lỗi hay không.

Hiển thị kết quả để định vị khu vực tìm lỗi

Nếu bạn không hiểu chương trình đang làm gì, việc thêm vào các lệnh để hiển thị thêm thông tin là cách dễ dàng và hiệu quả nhất để tìm được lỗi. Đưa chúng vào để xác định sự hiểu biết của bạn hoặc để hình thành ý tưởng cho bạn về vấn đề gây lỗi. Ví dụ, hiển thị "không thể đến được đây" nếu bạn nghĩ là không có khả năng để thâm nhập vào một điểm nào đó trong đoạn mã; tiếp đến nếu bạn thấy xuất hiện thông điệp đó thì chuyển các câu lệnh xuất ngược về đầu để tìm ra nơi đầu tiên gây lỗi. Hoặc là hiển thị các thông điệp "đến đây" và di chuyển xuống dưới để xác định nơi cuối cùng mà ở đó công việc dường như là hoạt động đúng. Mỗi thông điệp nên được phân biệt để bạn có thể nhận biết được thông điệp bạn đang quan sát.

Hiển thị các thông điệp theo một định dạng cố định rõ ràng sao cho chúng có thể được duyệt qua bằng mắt hay bằng các chương trình như công cụ xác định mẫu `grep`. (Một chương trình như `grep` rất hữu ích cho việc tìm kiếm văn bản. Chương 9 có trình bày một cài đặt đơn giản). Nếu bạn cho hiển thị giá trị của một biến thì hãy định dạng nó cùng một cách qua mỗi lần hiển thị. Trong ngôn ngữ C và C++ trình bày các con trỏ như là các số thập lục phân với `%x` hay `%p`: điều này giúp bạn nhận ra việc hai con trỏ có cùng giá trị hay có liên quan nhau. Hãy học cách đọc các giá trị con trỏ và nhận biết điểm giống nhau và khác nhau, như 0, các số âm, các số lẻ, và các số nhỏ. Sự quen thuộc định dạng của các địa chỉ cũng sẽ hữu ích khi bạn sử dụng một trình gỡ rối.

Nếu kết quả xuất ra có khuynh hướng rất lớn thì có thể chỉ cần in ra các ký tự đơn như A, B, C,... để đánh dấu nơi mà chương trình đi qua.

Viết mã tự kiểm tra

Nếu cần nhiều thông tin hơn thì bạn có thể viết hàm kiểm tra cho riêng bạn để kiểm tra một điều kiện, bỏ đi các biến có liên quan, và kết thúc chương trình

```
/* Hàm check: kiểm tra diều kiện, in ra và hỏng */

void check(char *s)

{

    if (var1 > var 2) {

        printf("%s: var1 %d var2 %d\n", s, var1, var2);

        fflush(stdout) /* bảo đảm tất cả kết quả xuất ra được sạch */

        abort();        /* tín hiệu kết thúc bất thường*/

    }

}
```

Ta viết hàm `check` để gọi hàm `abort`, một hàm trong thư viện chuẩn C có chức năng làm cho chương trình đang thi hành kết thúc một cách bất thường. Nhưng trong các ứng dụng khác, bạn có thể muốn hàm `check` vẫn thi hành sau khi in ra kết quả.

Kế tiếp, thêm vào các lời gọi tới hàm `check` vào bất kỳ nơi đâu bạn thấy là hữu ích trong mã nguồn của bạn:

```
check("trước điểm nghi ngờ");

/*…mã nguồn nghi ngờ…*/

check("sau điểm nghi ngờ");
```

Sau khi một lỗi được sửa, đừng vứt bỏ hàm `check`. Cứ lưu nó trong mã nguồn chương trình, đóng nó lại bằng dấu chú thích hay điều khiển nó bằng một lựa chọn gỡ rối, để sau này nó có thể được kích hoạt lại khi xuất hiện vấn đề khó khác.

Đối với các vấn đề khó hơn, hàm `check` có thể được phát triển lên để thực hiện việc kiểm tra và hiển thị của các cấu trúc dữ liệu. Cách tiếp cận này có thể được tổng quát hoá thành các hàm thực hiện kiểm tra tính thống nhất của các cấu trúc dữ liệu và các thông tin khác. Trong một chương trình có các cấu trúc dữ liệu phức tạp, một ý tưởng hay là viết các kiểm tra này trước khi các vấn đề xảy ra, như các thành phần của chương trình hoàn chỉnh; như thế chúng có thể được kích hoạt khi bắt đầu phát sinh lỗi. Đừng dùng chúng chỉ khi gỡ rối; cài đặt chúng trong suốt các giai đoạn phát triển chương trình. Nếu không quá tốn kém thì tốt nhất là nên để nó luôn trong trạng thái hoạt động. Các chương trình lớn như các hệ thống chuyển điện thoại thường được dùng nhiều mã nguồn để "kiểm tra tính đúng đắn" của các hệ thống con có chức năng theo dõi thông tin và thiết bị, và thông báo hoặc thậm chí sửa lỗi nếu xảy ra.

Tạo log file

Một chiến lược khác là tạo ra *log file* chứa một loạt các kết quả gỡ rối với định dạng cố định. Khi xảy ra lỗi, *log file* sẽ lưu lại những việc thực hiện ngay trước khi xảy ra lỗi. Các *web server* và các chương trình mạng khác duy trì các *log file* dài của việc lưu thông tin giúp chúng theo dõi chính chúng và các *client* của chúng; phân đoạn sau đây (đã được hiệu chỉnh cho vừa) được trích ra từ một hệ thống cục bộ:

```
[Sun Dec 27 16:19:24 1998]
    HTTPd: access to /usr/local/httpd/cgi-bin/test.html
        failed for m1.cs.bell-labs.com,
        reason: client denied by server (CGI non-executable)
```

from http://m2.cs.bell-labs.com/cgi-bin/test.pl

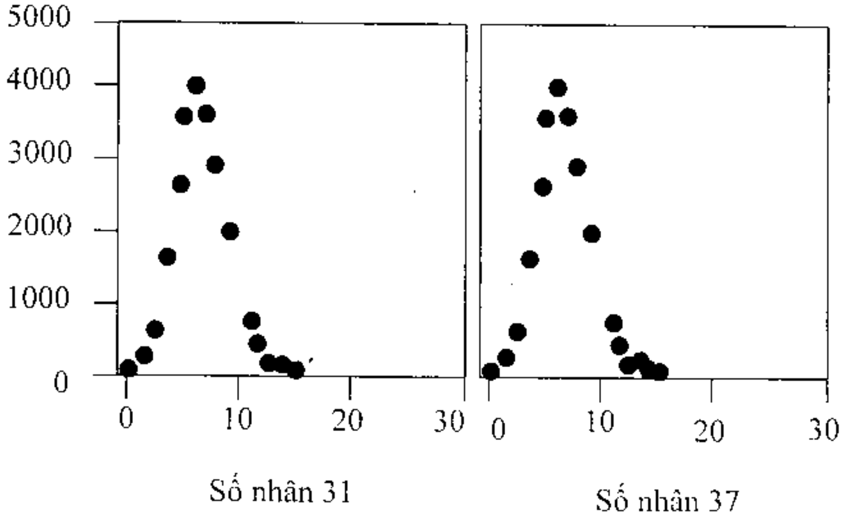
Hãy chắc chắn là các `buffer` nhập/xuất được làm sạch để các thông tin mới nhất được lưu vào *log file*. Các hàm xuất kết quả như `printf` thông thường đưa kết quả của chúng vào `buffer` để in ra hiệu quả hơn; việc kết thúc bất thường có thể tiêu huỷ kết quả đã được đưa vào `buffer` này. Trong C một lời gọi tới hàm `fflush` bảo đảm rằng tất cả các kết quả được xuất ra trước khi chương trình hỏng; trong C++ và Java có các hàm `flush` tương tự như thế cho việc xuất kết quả. Hoặc nếu bạn không sợ tốn kém thì có thể tránh làm sạch toàn bộ vấn đề bằng cách sử dụng các hàm nhập/xuất không sử dụng `buffer` cho các *log file*. Các hàm chuẩn `setbuf` và `setvbuf` điều khiển việc sử dụng `buffer`; `setbuf(fp, NULL)` làm tắt chức năng dùng `buffer` trên luồng `fp`. Các luồng xử lý lỗi chuẩn `(stderr, cerr, System.err)` mặc định thường là tắt chức năng dùng `buffer`.

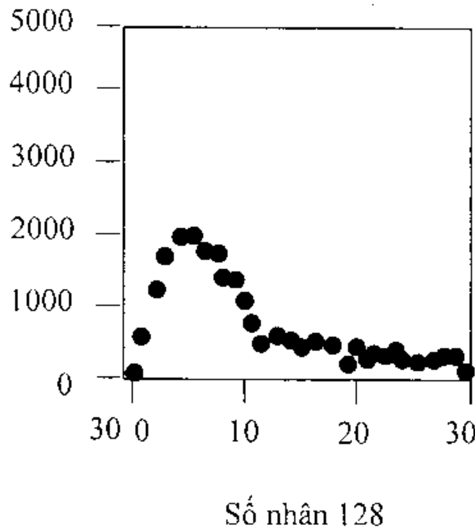
Vẽ hình

Đối với việc kiểm chứng và gỡ rối đôi khi các hình ảnh hiệu quả hơn là văn bản. Trong Chương 2, ta đã thấy các hình ảnh rất hữu ích để thể hiện các cấu trúc dữ liệu một cách dễ hiểu, và dĩ nhiên là khi đang viết phần mềm đồ họa, nhưng thật ra chúng có thể được dùng cho tất cả các loại chương trình. Các dấu chấm phân bố lộn xộn minh họa các giá trị không chính xác hiệu quả hơn là các cột của các con số. Một biểu đồ của dữ liệu cho thấy các sự bất thường trong điểm thi, các số ngẫu nhiên, các kích thước khối trong các điều khiển cấp phát và bảng băm, và v.v...

Nếu bạn không hiểu được điều gì đang diễn ra bên trong chương trình của bạn thì thử thêm chú thích cho các cấu trúc dữ liệu với các thống kê và vẽ đồ thị biểu diễn kết quả. Các đồ thị sau đây minh họa cho chương trình *Markov* trong C ở Chương 3, trục x biểu thị các chiều dài của chuỗi băm, và trục y biểu thị số lượng tương ứng của các phần trong các chuỗi của chiều dài đó. Dữ liệu vào là quyển sách 42685 từ, và 22482 tiếp đầu ngữ. Hai đồ thị đầu tiên minh họa cho hàm băm tốt ứng với các số nhân là 31 và 37, và đồ thị thứ ba minh họa cho hàm băm tồi với số nhân là 128. Trong

hai trường hợp đầu tiên, không có chuỗi nào có nhiều hơn 15 hoặc 16 phần tử và hầu hết các phần tử thuộc các chuỗi có chiều dài 5 hoặc 6. Trong trường hợp thứ 3, sự phân bố trải rộng hơn, chuỗi dài nhất có 187 phần tử, và có hàng ngàn phần tử trong các chuỗi dài hơn 20.





Sử dụng các công cụ

Hãy tận dụng các công cụ trong môi trường bạn đang gỡ rối. Ví dụ, một chương trình so sánh tập tin như `diff` so sánh các kết quả xuất ra từ các lần gỡ rối thành công và thất bại để bạn có thể tập trung vào điều được thay đổi. Nếu kết quả của việc gỡ rối dài, hãy dùng `grep` để tìm kiếm nó hoặc dùng một trình soạn thảo để kiểm tra nó. Tránh gửi các kết quả gỡ rối ra máy in; máy tính duyệt qua lượng kết quả lớn tốt hơn con người. Sử dụng các giao diện *script* và các công cụ khác để thực hiện tự động tiến trình xử lý dữ liệu xuất ra từ các lần chạy gỡ rối.

Viết các chương trình đơn giản để kiểm tra tính đúng đắn hay xác nhận sự hiểu biết của bạn về vấn đề hoạt động như thế nào. Ví dụ, việc giải phóng một con trỏ `NULL` có hợp lý hay không?

```
int main(void)

{

        free(NULL);

        return 0;

}
```

Các chương trình điều khiển mã nguồn như RCS theo dõi các phiên bản của mã nguồn sao cho bạn có thể nhìn thấy điều được thay đổi và có thể chuyển nó về các phiên bản trước đó để phục hồi một trạng thái trước đó. Bên cạnh việc chỉ ra những điều được thay đổi gần đây, chúng cũng có thể xác định được các phần của mã nguồn thường xuyên bị sửa đổi; các phần này thường là nơi tiềm tàng của lỗi.

Lưu vết

Nếu việc tìm lỗi được tiến hành sau một thời gian thì bạn sẽ bắt đầu mất dấu vết về những gì bạn đã thử qua và những gì bạn đã xem xét. Nếu bạn lưu lại các kiểm tra và các kết quả của bạn thì bạn ít khi xem lại vấn đề bạn đã xét hoặc bạn nghĩ rằng đã kiểm tra một số khả năng mà thực ra bạn

chưa kiểm tra. Thao tác lưu vết sẽ giúp bạn ghi nhớ được vấn đề trong lần kế tiếp khi có xuất hiện vấn đề tương tự, và nó cũng sẽ hữu ích khi bạn giải thích vấn đề cho người khác.

5.4. Phương sách cuối cùng

Bạn sẽ làm gì nếu không có cách nào giúp bạn giải quyết được vấn đề? Đã đến lúc bạn phải dùng một trình gỡ rối tốt để chạy chương trình từng bước. Nếu bạn nghĩ rằng cách thức hoạt động của chương trình là sai quá rõ ràng, như vậy bạn hoàn toàn xem xét không đúng chỗ, hoặc đúng chỗ nhưng không thấy được vấn đề, thì trình gỡ rối sẽ giúp bạn suy nghĩ một cách đúng đắn. Các lỗi do "mô hình trí óc" này là một trong số các lỗi khó phát hiện nhất; cơ chế sửa lỗi này là vô giá.

Sự quan niệm sai đôi khi rất đơn giản: độ ưu tiên toán tử không đúng, hay toán tử sai, hay việc canh hàng không khớp với cấu trúc thực tế, hay lỗi về phạm vi của biến, hay việc lẫn lộn cách dùng giữa biến cục bộ và biến toàn cục. Ví dụ, các lập trình viên thường quên rằng toán tử & và | có độ ưu tiên thấp hơn toán tử == và !=. Khi viết một đoạn mã nguồn như sau:

```
?       if (x & 1 ==0)
?               ...
```

và không biết được tại sao điều này luôn luôn sai. Đôi khi một cái trượt ngón trên bàn phím cũng làm chuyển đổi một dấu = thành hai hay ngược lại:

```
?       while ((c == getchar()) != EOF)
?               if (c = '\n')
?                       break;
```

Hay mã nguồn dư thừa được bỏ qua trong quá trình soạn thảo

```
?       for (i   0; i < n ; i++);
?               a[i++] = 0;
```

Hay gây ra lỗi do gõ vào cẩu thả:

```
?     switch (c) {
?             case '<':
?                     mode = LESS;
?                     break;
?             case '>':
?                     mode = GREATER;
?                     break;
?             defualt:
?                     mode = EQUAL;
?                     break;
?     }
```

Đôi khi lỗi liên quan đến các đối số được đưa vào không đúng thứ tự trong một tình huống mà việc kiểm tra kiểu không phát hiện được, như sau:

```
?     memset(p, n, 0);  /* lưu n số 0 vào p*/
```

thay vì

```
      memset(p, 0, n);  /* lưu n số 0 vào p*/
```

Đôi khi có một vài thay đổi bạn không kiểm soát được – các biến toàn cục bị thay đổi và bạn không nhận biết được rằng một vài tiến trình khác có thể truy xuất đến chúng.

Đôi khi giải thuật hoặc cấu trúc dữ liệu của bạn có một rất nghiêm trọng và bạn không thể nhìn thấy được. Trong khi chuẩn bị các thao tác trên các danh sách liên kết, ta đã viết một tập các hàm về danh sách cho phép tạo mới một phần tử ở đầu hoặc cuối các danh sách...; các hàm này được trình bày trong Chương 2. Dĩ nhiên chúng tôi đã viết một chương trình kiểm tra để chắc chắn rằng mọi việc được thực hiện chính xác. Một số các kiểm tra

đầu tiên hoạt động tốt nhưng sau đó có một kiểm tra không chạy được. Nguyên nhân là do chương trình kiểm tra:

```
        ?     while (canf("%s %d", name, &value) != EOF)
    {
        ?           p = newitem(name, value);
        ?           list1 = addfront(list1, p);
        ?           list2 = addend(list2, p);
        ?     }
        ?     for (p = list1; p!= NULL; p = p->next)
        ?           printf("%s %d\n", p->name, p->value);
```

Thật ngạc nhiên khi thấy rằng vòng lặp đầu tiên đặt cùng nút p vào cả hai danh sách, do đó các con trỏ được cố gắng truy cập một cách vô vọng vào lúc chúng ta thực hiện in.

Cũng thật khó khăn để tìm thấy được lỗi loại này bởi vì ý nghĩ của bạn vạch ra một hướng suy nghĩ sai lệch, bạn cứ cho điều không đúng là đúng. Vì thế trình biên dịch sẽ giúp bạn, nó giúp bạn đi theo một hướng khác, theo những gì mà chương trình thực hiện, chứ không phải những gì bạn nghĩ là nó sẽ thực hiện. Thông thường vấn đề được chú ý là những gì không đúng so với cấu trúc của toàn bộ chương trình, và để tìm được lỗi bạn cần phải quay trở lại các suy luận ban đầu của bạn.

Hãy chú ý rằng trong ví dụ về danh sách này, lỗi xảy ra trong mã nguồn kiểm tra, khiến cho việc tìm lỗi khó hơn nhiều. Rất hay gặp trường hợp tìm kiếm lỗi ở nơi mà lỗi không xuất hiện, vì chương trình kiểm tra sai, hoặc do kiểm tra trên phiên bản sai của chương trình, hoặc do không cập nhật hay biên dịch lại trước khi kiểm tra.

Nếu bạn không thể tìm thấy lỗi sau khi đã tìm kiếm một cách kỹ lưỡng, hãy nghỉ giải lao. Hãy thư giãn và làm chuyện khác. Thảo luận với

một người bạn và yêu cầu được giúp đỡ. Bạn có thể có được câu trả lời, nhưng nếu không, bạn sẽ không đi theo con đường cũ ở lần gỡ rối kế tiếp.

Nếu như trong một thời gian dài mà vẫn không phát hiện được lỗi thì vấn đề thực sự là trình biên dịch hay thư viện hay hệ điều hành hay thậm chí phần cứng, đặc biệt là nếu có một vài thay đổi trong môi trường ngay trước khi một lỗi xuất hiện. Bạn đừng bao giờ vội vàng đổ lỗi cho một trong các vấn đề trên, nhưng sau khi đã loại ra mọi thứ, thì mới có thể nghĩ tới chúng. Có lần chúng tôi phải di chuyển một chương trình định dạng văn bản lớn từ máy Unix nguyên thủy của nó sang một máy PC. Chương trình được biên dịch không có một sự khác biệt nào, nhưng lại hoạt động theo một cách thức rất khác thường: nó cắt xuống dòng gần như là mỗi hai ký tự của dữ liệu nhập. Suy nghĩ đầu tiên của chúng tôi là một vài thuộc tính có kiểu số nguyên 16 bit thay vì 32 bit, hay có thể là một vài vấn đề liên quan đến thứ tự khác thường của các byte. Nhưng nhờ vào việc in ra các ký tự trong vòng lặp chính, cuối cùng chúng tôi đi dần đến một lỗi trong tập tin `ctype.h`, được cung cấp bởi nhà cung cấp trình biên dịch. Nó cài đặt `isprint` như là một macro hàm

```
?      #define  isprint(c)  ((c)  >=  040  &&  (c)  <
0177)
```

và vòng lặp nhập liệu như sau:

```
?      while (isprint(c = getchar()))
?              ...
```

Mỗi khi một ký tự nhập vào là khoảng trắng (040, một cách viết tồi là ' ') hay lớn hơn, điều này hầu như lúc nào cũng gặp, hàm `getchar` được gọi lần thứ hai bởi vì macro tính toán đối số của nó hai lần, và ký tự nhập vào đầu tiên luôn luôn mất. Mã nguồn nguyên thủy không được rõ ràng như chúng ta nghĩ -- có quá nhiều sai sót trong điều kiện lặp – nhưng sai sót trong tập tin `header` của nhà cung cấp là không thể tha thứ được.

Ngày nay bạn vẫn có thể tìm thấy các phiên bản của vấn đề này; macro sau đây được lấy từ các tập tin `header` hiện tại của một nhà cung cấp

khác:

```
?     #define iscsym(c) (isalnum(c) || ((c) == '_'))
```

Bộ nhớ "rò rỉ" – không thể trả về bộ nhớ không còn được dùng đến nữa – là nguồn gốc của các thao tác bất thường. Một vấn đề nữa là quên đóng các tập tin, đến lúc bảng quản lý tập tin đang mở đầy thì chương trình không thể mở thêm một tập tin nào nữa. Các chương trình có các lỗi "rò rỉ" thường bị treo một cách bí ẩn bởi vì chúng cạn kiệt một số tài nguyên và ta không thể tạo ra lại được các lỗi đặc trưng này.

Đôi khi chính phần cứng có vấn đề. Năm 1994, lỗi dấu chấm động trong bộ xử lý Pentium làm cho một số tính toán nào đó tạo ra các kết quả sai, là một lỗi được phổ biến rộng rãi và đắt giá trong thiết kế phần cứng, nhưng khi nó được phát hiện, thì dĩ nhiên nó được tạo lại một cách đúng đắn. Một trong các loại lạ nhất mà chúng tôi từng thấy có liên quan đến một trình tính toán, xảy ra trên một hệ thống có hai bộ xử lý, cách đây lâu rồi. Đôi khi diễn đạt ½ sẽ in ra 0,5 và đôi khi nó lại in ra giá trị vẫn đảm bảo tính thống nhất nhưng hoàn toàn sai 0,7432: không có một quy luật nào để cho biết kết quả có được là đúng hay sai. Vấn đề này cuối cùng được truy ra là một lỗi của đơn vị chấm động trong một trong các bộ xử lý. Khi trình tính toán được chạy ngẫu nhiên trên hoặc một bộ xử lý này hoặc một bộ xử lý còn lại, thì các kết quả là hoặc chính xác hoặc vô lý.

5.5. Các lỗi không có khả năng xuất hiện lại

Các lỗi không xuất hiện theo một cách thức nào hết thì rất khó giải quyết, và thông thường vấn đề sẽ không rõ ràng như là lỗi phần cứng. Tuy nhiên hành vi này thật ra là do thông tin của chính chương trình; nghĩa là lỗi này ít có khả năng do sai sót trong thuật toán của bạn, mà do theo một cách thức nào đó, mã nguồn của bạn sử dụng thông tin bị thay đổi qua mỗi lần chạy chương trình.

Hãy kiểm tra xem tất cả các biến được khởi tạo hết chưa; có thể là bạn đã lấy một giá trị ngẫu nhiên từ một giá trị bất kỳ được lưu trước đó trong cùng vị trí bộ nhớ. Các biến cục bộ của các hàm và bộ nhớ được tạo ra

từ các chức năng cấp phát thường là các thủ phạm trong C và C++. Gán các giá trị được biết cho các biến.

Nếu lỗi làm thay đổi hành vi hay thậm chí không xuất hiện khi mã nguồn gỡ rối được thêm vào, thì nó có thể là do lỗi cấp phát bộ nhớ - bạn đã viết ra ngoài vùng nhớ được cấp phát ở một chỗ nào đó, và việc thêm vào mã gỡ rối làm thay đổi hình thức lưu trữ đủ để làm thay đổi sự tác động của lỗi. Phần lớn các hàm xuất dữ liệu, từ hàm `printf` cho đến các hộp thoại, là tự cấp phát bộ nhớ, góp phần làm cho vấn đề phức tạp thêm.

Nếu dường như không có một vấn đề gì có thể bị sai ở nơi xảy ra lỗi, thì vấn đề thường là viết chồng lên bộ nhớ bằng cách lưu vào một vị trí bộ nhớ không được sử dụng trong suốt một khoảng thời gian dài. Đôi khi đây là một vấn đề về quản lý con trỏ lỏng lẻo, ở đó một con trỏ trỏ tới một biến cục bộ được trả về một cách vô tình từ một hàm, và nó được dùng đến. Việc trả về điều khiển của một biến cục bộ là một trong những nguyên nhân dẫn đến loại lỗi này.

```
?       char *msg(int n, char *s)
?       {
?               char buf[100];
?
?               sprintf(buf, "error %d: %s\n", n, s);
?               return buf;
?       }
```

Vào lúc con trỏ được trả về bởi hàm `msg` được sử dụng, thì nó không còn trỏ tới vùng lưu trữ có ý nghĩa nữa. Bạn phải cấp phát vùng nhớ bằng hàm `malloc`, dùng một mảng tĩnh, hay yêu cầu đối tượng gọi cung cấp không gian.

Việc dùng một giá trị đã được cấp phát một cách linh động sau khi nó được giải phóng sẽ có các triệu chứng tương tự. Chúng ta đã đề cập điều

này trong Chương 2 khi viết hàm `freeall`. Đoạn mã sau đây là sai:

```
?       for (p = listp; p != NULL; p = p->next)
?               free(p);
```

Khi bộ nhớ được giải phóng, nó phải không được dùng tới vì các nội dung của nó có thể thay đổi và không đảm bảo `p->next` vẫn còn trỏ tới một vị trí đúng.

Trong một số cài đặt của hàm `malloc` và `free`, việc giải phóng một phần tử hai lần sẽ làm phá vỡ cấu trúc dữ liệu bên trong nhưng phải sau một thời gian thì vấn đề rắc rối mới phát sinh khi một lời gọi sau đó rơi vào các hư hỏng xảy ra trước đó. Một số trình cấp phát có kèm theo các lựa chọn gỡ rối, các lựa chọn này có thể được kích hoạt để kiểm tra tính thống nhất cục bộ tại mỗi lần gọi; chúng được kích hoạt nếu bạn gặp một lỗi không kiểm soát được. Nếu không có chức năng này, bạn có thể viết trình cấp phát của riêng bạn để thực hiện một vài kiểm tra tính thống nhất của chính nó hay để lưu trữ tất cả các lời gọi cho việc phân tích riêng. Một trình cấp phát mà không cần phải chạy nhanh thường dễ viết, do đó chiến lược này là khá thi khi tình huống trở nên quá tồi tệ. Cũng có một số sản phẩm thương mại tốt cho việc kiểm tra việc quản lý bộ nhớ và bắt các lỗi và các "rò rỉ": việc viết hàm `malloc` và `free` cho riêng bạn có thể cho bạn một vài lợi ích nếu bạn không có các hàm này hay không thể truy cập vào bên trong chúng.

Khi một chương trình hoạt động tốt cho một người này nhưng không tốt cho một người khác, là do một số vấn đề phụ thuộc vào môi trường bên ngoài của chương trình. Nó có thể là do tập tin đưa vào làm dữ liệu nhập cho chương trình, các thao tác trên tập tin, các biến môi trường, đường dẫn tới các lệnh, các mặc định, hay các tập tin khởi tạo. Rất khó để thảo luận về các tình huống này.

Bài tập 5-1. Viết một phiên bản của hàm `malloc` và `free` có thể được dùng cho việc gỡ rối các vấn đề về quản lý lưu trữ. Một cách tiếp cận là kiểm tra toàn bộ vùng làm việc trên mỗi lời gọi tới hàm `malloc` và `free`; cách tiếp cận khác là viết các thông tin lưu lại có thể được xử lý bởi một

chương trình khác. Với mỗi cách, thêm các đánh dấu vào đầu và cuối của mỗi khối được cấp phát để phát hiện ra các trường hợp vượt ra ngoài giới hạn của hai đầu.

5.6. Công cụ gỡ rối

Các trình gỡ rối không phải là các công cụ duy nhất để tìm kiếm lỗi. Nhiều chương trình có thể giúp chúng ta tìm trong dòng dữ liệu xuất để lấy ra các bit quan trọng, tìm ra những điều bất thường hoặc sắp xếp lại dữ liệu để xem xét được dễ dàng hơn những gì đang xảy ra. Nhiều chương trình loại này là một phần của các bộ công cụ chuẩn: một số được viết để giúp tìm kiếm lỗi hoặc để phân tích một chương trình nào đó.

Trong phần này ta sẽ mô tả một chương trình đơn giản gọi là các chuỗi mà chúng đặc biệt hữu ích cho việc tìm kiếm các tập tin có chứa nhiều ký tự không thuộc bảng chữ cái, chẳng hạn các tập tin thực thi được (*.EXE) hoặc các tập tin nhị phân được một số chương trình xử lý văn bản hỗ trợ. Thông thường có các thông tin ẩn bên trong như một đoạn văn bản hay các thông báo lỗi và các tùy chọn không thông báo, hoặc tên các tập tin, thư mục hoặc các tên của các hàm mà chương trình có thể gọi.

Ta cũng tìm thấy các chuỗi hữu ích cho việc định vị các văn bản trong các tập tin nhị phân. Các tập tin ảnh thường chứa các chuỗi ASCII để chỉ định chương trình tạo ra chúng, và các tập tin nén và các tập tin lưu trữ (chẳng hạn các tập tin nén) có thể chứa các tên tập tin, các chuỗi cũng có thể tìm thấy các văn bản trong các tập tin này.

Các hệ thống Unix cung cấp sẵn một chương trình các chuỗi, cho dù nó hơi khác so với các chuỗi ở trên. Nó nhận ra khi dữ liệu nhập của nó là một chương trình và chỉ xét những đoạn văn bản và dữ liệu, bỏ qua các bảng biểu tượng. Nhưng nó có tùy chọn, cho phép chọn đọc hết toàn bộ tập tin.

Để hiệu quả, chuỗi trích văn bản ASCII từ tập tin nhị phân vì vậy văn bản có thể được các chương trình khác đọc hay xử lý. Nếu một thông báo lỗi không mang một dấu hiệu nhận diện nào, nó có thể không xác định được chương trình nào đã phát sinh ra nó. Trong trường hợp đó, dùng lệnh

để tìm kiếm trong các thư mục có thể có:

```
% strings *.exe *.dll | grep 'mystery message'
```

có thể tìm ra chương trình tạo ra nó.

Chức năng của chuỗi đọc một tập tin và in ra tất cả các đường chạy với ít nhất có `MINLEN = 6` các ký tự có thể in ra.

```
/* Hàm strings: trích các chuỗi có thể in ra từ stream */
void strings(char *name, FILE *fin)
{
    int c, i;
    char buf[BUFSIZ]
    do {
        for (i = 0; (c = getc(fin)) != EOF; ){
            if (!isprint(c))
                break;
            buf[i++] = c;
            if (i >= BUFSIZ)
                break;
        }
        if (i >= MINLEN) /* in chuỗi buf nếu nó đủ dài */
            printf("%s:%.*s\n", name, i, buf);
    } while (c != EOF);
```

Lệnh `printf` định dạng in chuỗi `%.*s` lấy chiều dài chuỗi từ tham số `i` kế tiếp, vì chuỗi `buf` không kết thúc bằng *null*.

Vòng lặp `do-while` tìm và sau đó in ra từng chuỗi, kết thúc khi gặp `EOF`. Kiểm tra kết thúc tập tin ở cuối giúp cho vòng lặp `getc` và vòng lặp `string` cùng chung điều kiện kết thúc và để cho hàm in chuỗi xử lý kết thúc chuỗi, kết thúc tập tin và chuỗi quá dài.

Một vòng lặp ngoài theo chuẩn với việc kiểm tra ở đầu hoặc một vòng lặp đơn `getc` với thân chương trình phức tạp hơn, sẽ đòi hỏi thêm lệnh `printf`. Hàm này bắt đầu hoạt động theo cách đó, nhưng nó có một lỗi trong câu lệnh `printf`. Ta đã chỉnh sửa ở một chỗ nhưng quên chỉnh ở hai chỗ khác. (Ta có phạm sai lầm tương tự ở chỗ nào khác không?). Đến đây, điều đó trở nên rõ ràng rằng chương trình cần phải được viết lại để có ít đoạn mã lặp lại hơn: điều này dẫn đến vòng lặp `do-while`.

Hàm `main` của `strings` gọi hàm `strings` đối với mỗi tập tin đối số của nó:

```
    /* strings main: tìm chuỗi có thể in được trong
tập tin */
    int main(int argc, char *argv[])
    {
        int i;
        FILE *fin;

        setprogname("strings");
        if (argc == 1)
            eprintf("usage: strings filenames");
        else {
            for (i = 1; i < argc; i++) {
```

```
        if ((fin = fopen(argv[i], "rb")) == NULL )
            weprintf("can't open %s:", argv[i] );
          else {
                strings(argv[i], fin);
                fclose(fin);
            }
        }
    }
    return 0;
}
```

Bạn có thể ngạc nhiên rằng hàm `strings` không dọc dữ liệu nhập chuẩn của nó nếu không có tập tin. Để giải thích tại sao nó không thực hiện, ta cần mô tả việc gỡ rối.

Trường hợp kiểm tra rõ ràng cho hàm `strings` là chạy chính chương trình này. Chương trình chạy tốt trên Unix, tuy nhiên trên nền Windows 95 lệnh

```
C:\> strings <strings.exe
```

cho ra đúng 5 dòng kết quả:

```
!This program cannot be run in DOS mode.
'.rdata
@.data
.idata
.reloc
```

Dòng đầu tiên trông giống như một thông báo lỗi và ta phải tốn một ít thời gian trước khi nhận ra rằng nó chính là một dòng của kết quả chương trình, theo như nó chạy. Nó không biết có một quá trình cố gắng gỡ rối được thực hiện vì hiểu nhầm đó là câu thông báo lỗi.

Nhưng có nhiều kết quả được xuất ra hơn. Nó ở đâu ra? Ta dần dần hiểu được vấn đề. Đây chính là vấn đề tính khả chuyển được mô tả chi tiết hơn trong Chương 8. Ban đầu, chúng ta đã viết chương trình chỉ để đọc từ dữ liệu nhập chuẩn sử dụng hàm `getchar`. Tuy nhiên, trên Windows hàm `getchar` trả về `EOF` khi nó gặp phải byte đặc biệt (`0x1A` hoặc Control-Z) trong dữ liệu nhập dạng văn bản và điều này gây ra việc chương trình dừng sớm.

Đây tất nhiên là một cách xử lý hợp lý, nhưng nó không giống như những gì chúng ta mong đợi trong môi trường Unix. Kết quả là việc mở một tập tin theo nhị phân dùng `"rb"`. Nhưng `stdin` đã mở rồi và không có lệnh C chuẩn nào để thay đổi trạng thái này. (Các hàm như `fdopen` hay `setmode` có thể được dùng nhưng chúng không phải lệnh C chuẩn). Cuối cùng ta gặp phải một loạt các thay thế khó chấp nhận ép buộc người dùng nhập tên tập tin để nó có thể chạy tốt trên môi trường Windows nhưng không tương thích trên Unix; và mặc nhiên đưa ra các kết quả sai nếu người sử dụng Windows thử đọc từ dữ liệu nhập chuẩn; hoặc dùng sự biên dịch có điều kiện để tạo ra các kết quả thích hợp với các hệ thống khác nhau, trả giá cho việc giảm đi tính tương thích. Ta chọn sự lựa chọn đầu vì vậy cùng một chương trình có thể làm việc như nhau ở mọi nơi.

Bài tập 5-2. Chương trình `strings` in ra các chuỗi với `MINLEN` hay nhiều ký tự hơn, điều này thỉnh thoảng xuất ra nhiều kết quả hơn. Hãy nhập các chuỗi với tham số tùy chọn để định nghĩa chiều dài chuỗi tối thiểu.

Bài tập 5-3. Viết chương trình `vis`, chép dữ liệu nhập vào dữ liệu xuất, ngoại trừ nó hiển thị các byte không in được như các ký tự *backspace*, các ký tự điều khiển và các ký tự không phải ASCII như `\Xhh` trong đó `hh` là thể hiện hệ 16 của byte không in được. Trái ngược với `strings`, `vis` thật sự hữu ích cho việc xem xét các dữ liệu nhập chỉ chứa vài ký tự không in được.

Bài tập 5-4. `vis` đưa ra kết quả gì nếu dữ liệu nhập là `\XCA`? Bạn làm thế nào để có thể tạo kết quả của `vis` trở nên mơ hồ?

Bài tập 5-5. Mở rộng chương trình `vis` để xử lý một loạt các tập tin, nhóm các dòng dài tại bất cứ cột nào và loại bỏ các ký tự không in được một cách hoàn toàn. Những đặc trưng nào khác có thể thích hợp với vai trò của chương trình?

5.7. Lỗi của người khác

Thực tế hầu hết các lập trình viên đều không mong muốn việc tạo ra một hệ thống hoàn toàn mới từ đầu. Thay vào đó, họ bỏ ra nhiều giờ sử dụng, bảo trì, thay đổi và sau đó không tránh khỏi việc gỡ rối các đoạn mã do người khác viết.

Khi gỡ rối mã nguồn của người khác, ta sẽ áp dụng mọi thứ như khi ta gỡ rối mã nguồn của chính mình. Trước khi bắt đầu, ta phải hiểu được một chương trình được tổ chức như thế nào và các lập trình viên ban đầu đã nghĩ và viết như thế nào.

Đây là nơi mà các công cụ có thể giúp một cách đáng kể. Các chương trình tìm kiếm văn bản như `grep` có thể tìm thấy tất cả các xuất hiện của các tên. Các tham khảo chéo cho một số ý tưởng cho cấu trúc chương trình. Sự hiển thị của đồ thị của các lời gọi hàm là có giá trị nếu nó không quá lớn. Từng bước duyệt qua chương trình, một lời gọi hàm tại thời điểm khi gỡ rối có thể cho thấy một chuỗi các biến cố. Một quá trình chạy của chương trình có thể có một số mấu chốt thông qua việc hiển thị những gì được thực hiện bởi chương trình qua thời gian. Các thay đổi thường xuyên là một dấu hiệu của dòng lệnh khó hiểu hoặc phải chịu các yêu cầu thay đổi và do đó dễ bị lỗi.

Thỉnh thoảng bạn cần phải theo dõi các lỗi trong phần mềm mà bạn không phải chịu trách nhiệm và không có mã nguồn của nó. Trong trường hợp này, nhiệm vụ của bạn là định rõ đặc điểm của lỗi một cách hiệu quả và bạn có thể thực hiện thông báo lỗi một cách chính xác, và khi đó bạn nên tránh các vấn đề khác nảy sinh.

Nếu bạn nghĩ rằng bạn đã tìm thấy một lỗi trong một chương trình của ai khác, bước đầu tiên là đảm bảo rằng đó là lỗi thực sự, vì thế bạn không phí thời gian của tác giả và uy tín của mình.

Khi bạn tìm thấy một lỗi biên dịch thì đảm bảo rằng lỗi là thật sự của trình biên dịch và không phải trong mã nguồn của bạn. Chẳng hạn, phép toán dịch bit sang phải sẽ điền vào các bit trống (dịch chuyển logic) hoặc truyền bit dấu (dịch số học) là không xác định trong C và C++. Do đó những người mới học thỉnh thoảng nghĩ rằng có lỗi nếu có lệnh như sau:

```
?       i = -1;

?       printf("%d\n", i >> 1);
```

cho ra một kết quả không xác định được. Nhưng điều này là vấn đề tương thích, vì phát biểu trên có thể hoạt động hợp lý nhưng khác nhau trên các hệ thống khác nhau. Thử kiểm tra sự hiểu của bạn trên nhiều hệ thống khác nhau và đảm bảo rằng bạn hiểu những gì xảy ra.

Đảm bảo lỗi là mới. Bạn có phiên bản mới nhất của chương trình không? Bạn đã có danh sách các lỗi đã sửa không? Hầu hết các phần mềm có nhiều phiên bản: nếu bạn tìm thấy một lỗi trong phiên bản 4.0b1, nó có thể được sửa hoặc thay thế bởi phiên bản mới như 4.04b2.

Cuối cùng, bạn phải tự đặt mình trong hoàn cảnh của người nhận báo cáo của bạn. Bạn muốn cung cấp cho người sở hữu nó một chương trình được kiểm tra tốt nhất có thể. Điều này không thật sự hữu ích nếu lỗi có thể được thể hiện với dữ liệu nhập lớn; hoặc trong một môi trường phức tạp, hoặc dữ liệu nhập từ nhiều tập tin. Đối với phiên bản bị lỗi của `isprint` được đề cập trong phần 5.4, ta có thể cung cấp một chương trình kiểm tra như sau:

```
/* Kiểm tra chương trình bị lỗi ở hàm isprint */

int main(void)

{

        int c;
```

```
        while (isprint(c = getchar()) || c != EOF)
        printf("%c", c);
        return 0;
}
```

Bất kỳ dòng nào của văn bản in được cũng đều là trường hợp để kiểm tra, vì dữ liệu xuất chỉ chứa một nửa dữ liệu nhập:

```
% echo 1234567890 | isprint_test
24680
%
```

Các thông báo lỗi tốt là các thông báo mà chỉ cần một hoặc hai dòng dữ liệu nhập trên một hệ thống mà có thể biểu diễn được lỗi và nó bao gồm cả việc sửa lỗi.

5.8. Tổng kết

Với thái độ đúng mực, việc gỡ rối có thể vui, giống như giải quyết một câu đố cho dù bạn có thích hay không, gỡ rối là một nghệ thuật mà chúng ta phải luyện tập thường xuyên. Thế nhưng điều đó rất tốt đẹp nếu các lỗi không xảy ra, vì vậy chúng ta cố gắng tránh chúng bằng cách viết các lệnh thật tốt ngay từ đầu. Mã nguồn được viết tốt có ít lỗi hơn từ khi bắt đầu và chúng sẽ dễ tìm thấy hơn.

Một khi thấy lỗi, điều đầu tiên phải làm là nghĩ kỹ về các nguồn gốc sinh ra nó. Làm thế nào mà gây ra lỗi? Nó có quen không? Có điều gì đó vừa mới đưa vào chương trình không? Có điều gì đặc biệt với dữ liệu nhập mà phát sinh ra lỗi không?

Nếu không tìm được nguồn gốc gây ra lỗi thì việc suy nghĩ kỹ vẫn là bước khởi đầu tốt nhất, sau đó thực hiện các thao tác thử sai một cách hệ thống để thu hẹp lại vị trí của vấn đề gây ra lỗi. Tất cả những điều đó là các

trường hợp của chiến lược tổng quát, chia để trị. điều này rất hữu ích trong quá trình gỡ rối giống như trong chính trị và trong chiến tranh.

Sử dụng các nguồn trợ giúp khác. Giải thích mã lệnh của bạn cho một ai đó là điều thật sự hiệu quả. Sử dụng chương trình gỡ rối để quan sát ngăn sếp. Sử dụng một số công cụ thương mại để kiểm tra việc rò rỉ bộ nhớ, tràn mảng, mã nguồn có khả năng gây ra lỗi và những thứ tương tự. Chạy từng bước chương trình của bạn để xác minh rằng bạn đã có suy nghĩ đúng hoặc sai về mục đích của mã nguồn.

Biết chính mình và tất cả các lỗi mình sai phạm. Một khi bạn tìm thấy và sửa lỗi, đảm bảo rằng bạn loại bỏ tất cả các lỗi tương tự khác. Suy nghĩ về những gì đã xảy ra để mà bạn có thể tránh việc lặp lại các lỗi như thế.

Chương 6

KIỂM CHỨNG

Kiểm chứng và gỡ rối thường được hiểu là như nhau, nhưng thực ra chúng không phải là như nhau. Để đơn giản, gỡ rối là những việc bạn cần phải làm khi chương trình bị lỗi. Kiểm chứng là sự kiểm tra có hệ thống nhằm tìm ra các lỗi của chương trình mà bạn nghĩ là nó đang thực hiện đúng.

Edsger Dijkstra nhận xét rằng việc kiểm chứng có thể giải thích sự hiện diện của các lỗi nhưng không thể cho biết được sự tiềm ẩn của chúng. Ông hy vọng rằng các chương trình có thể được viết một cách đúng đắn trong quá trình phát triển chúng để chương trình không còn lỗi và vì vậy không cần kiểm chứng nữa. Mặc dù đây là một mục tiêu tốt đẹp nhưng nó không thực tế. Do vậy, trong chương này chúng ta sẽ chú trọng đến những cách thức kiểm chứng để tìm ra lỗi nhanh chóng và hiệu quả.

Nghĩ đến các lỗi tiềm ẩn trong khi viết chương trình là một bước khởi đầu tốt. Việc kiểm chứng có hệ thống từ đơn giản đến các phức tạp bảo đảm rằng các chương trình có thể thực hiện đúng đắn và vẫn còn đúng khi mở rộng chúng. Sự tự động hóa giúp loại bỏ các thao tác thủ công và khuyến khích việc kiểm chứng bao quát. Các nhà lập trình đã và đang tích lũy nhiều mẹo lập trình từ kinh nghiệm thực tế.

Một cách để viết mã nguồn không còn lỗi là dùng một chương trình phát sinh ra nó. Nếu nhiệm vụ lập trình được hiểu rõ đến mức có thể viết mã nguồn tự động thì chúng nên được phát sinh một cách tự động. Thông thường, một chương trình có thể được phát sinh từ một đặc tả bằng một số ngôn ngữ đặc trưng. Chẳng hạn, ta biên dịch ngôn ngữ cấp cao sang hợp

ngữ; ta sử dụng những biểu thức thông thường để định rõ những mẫu của văn bản; sử dụng các ký hiệu như `SUM(A1:A50)` để biểu diễn các phép toán trên một các phần tử trong bảng tính. Trong trường hợp này, nếu bộ phát sinh hoặc bộ biên dịch đúng và nếu đặc tả đúng thì kết quả của chương trình cũng sẽ đúng. Chương 9 sẽ trình bày kỹ hơn về vấn đề này; ở đây ta sẽ tóm tắt các cách thực hiện kiểm chứng từ những bản đặc tả đầy đủ.

6.1. Kiểm tra trong khi viết mã nguồn

Một lỗi được tìm thấy càng sớm càng tốt. Nếu bạn nghĩ một cách có hệ thống về những gì bạn sẽ viết, bạn sẽ xác định được các thuộc tính đơn giản của chương trình ngay khi đang xây dựng nó. Kết quả là mã nguồn của bạn sẽ được kiểm chứng ít nhất một lần trước khi biên dịch. Nhờ đó một số loại lỗi xác định sẽ không bao giờ xuất hiện.

Kiểm chứng mã nguồn ở các giá trị biên của nó

Một kỹ thuật được gọi là kiểm tra điều kiện biên: khi viết một đoạn mã nguồn, ví dụ như một vòng lặp hay một biểu thức điều kiện, nếu việc kiểm tra điều kiện là đúng thì rẽ theo nhánh đúng hoặc vòng lặp sẽ thực hiện tiếp. Quá trình này gọi là việc kiểm tra điều kiện biên bởi vì bạn kiểm tra tại những biên tự nhiên bên trong chương trình và dữ liệu, chẳng hạn như giá trị nhập không tồn tại hoặc rỗng, một mảng đầy. Có ý kiến cho rằng hầu hết các lỗi xuất hiện tại biên. Nếu một đoạn mã nguồn sai, nó thường sai ở biên. Ngược lại, nếu nó có thể thực hiện đúng tại biên, nó cũng sẽ thực hiện tại các vị trí khác.

Đoạn chương trình này, mô phỏng hàm `fgets` đọc chuỗi các ký tự cho đến khi gặp ký tự kết thúc chuỗi hay đầy *buffer*:

```
?     int i;
?     char s[MAX];
?
?     for (i = 0; (s[i] = getchar()) != '\n' && i
```

```
< MAX-1; ++i)
?
?     s[--i] = '\0';
```

Tưởng tượng rằng bạn vừa viết xong vòng lặp này. Vòng lặp này sẽ thực hiện việc đọc vào một dòng. Kiểm tra biên đầu tiên là trường hợp đơn giản nhất: một dòng trống. Nếu bạn bắt đầu với một dòng không có ký tự nào, vòng lặp sẽ dừng ngay lần đầu tiên với giá trị của `i` là `0`, do đó giá trị của `i` sẽ giảm xuống và mang giá trị `-1` ở dòng lệnh cuối cùng và do đó sẽ gán giá trị rỗng vào `s[-1]` (`s[-1]` nằm trước vị trí bắt đầu của mảng). Việc kiểm tra điều kiện biên sẽ phát hiện ra lỗi này.

Nếu ta viết lại vòng lặp bằng cách gán các phần tử của mảng với các ký tự nhập vào từ bàn phím như sau:

```
?     for (i = 0; i < MAX-1; i++)
?           if ((s[i] = getchar()) == '\n')
?                 break;
?     s[i] = '\0';
```

Thực hiện lại kiểm tra biên ban đầu, ta dễ dàng chứng minh một dòng chỉ có một ký tự xuống dòng đã được xử lý chính xác khi đó giá trị của `i` là `0`, ký tự đọc được đầu tiên sẽ thoát khỏi vòng lặp, và `'\0'` sẽ được gán vào `s[0]`. Việc kiểm tra tương tự với 1 hoặc 2 ký tự nhập và kết thúc bằng dấu xuống dòng chứng tỏ vòng lặp làm việc tốt gần với biên đó.

Tuy nhiên, có những điều kiện biên khác cần phải kiểm tra. Nếu ta nhập vào một dòng dài hoặc dòng không có ký tự xuống dòng thì ta sẽ kiểm tra giá trị của `i` phải nhỏ hơn `MAX-1`. Nếu nhập vào dòng rỗng thì hàm `getchar()` sẽ trả về giá trị `EOF` ở lần gọi hàm đầu tiên. Ta buộc phải kiểm tra điều này:

```
?     for (i = 0; i < MAX-1; i++)
?         if ((s[i] = getchar()) == '\n' || s[i] == EOF)
```

```
?                   break;
?       s[i]='\0';
```

Việc kiểm tra điều kiện biên có thể tìm ra được nhiều lỗi, nhưng không phải là tìm ra được tất cả. Ta sẽ trở lại ví dụ này trong Chương 8, và chỉ ra rằng vẫn còn những khi đúng khi sai.

Bước tiếp theo là kiểm tra dữ liệu nhập vào tại những biên khác, chẳng hạn tại những biên mà mảng gần đầy, đầy hoặc quá đầy, đặc biệt là nếu ký tự xuống dòng được nhập vào cùng thời điểm. Ta sẽ không viết ra chi tiết ở đây mà xem đây là bài tập hữu ích dành cho các bạn. Suy nghĩ về các biên sẽ đưa ra câu hỏi là cần phải làm gì khi gán các giá trị cho *buffer* trước khi nhập vào ký tự xuống dòng `'\n'`. Sự thiếu sót này trong đặc tả nên được giải quyết sớm, và các biên dùng để kiểm tra sẽ giúp dễ phát hiện nó.

Việc kiểm tra điều kiện biên sẽ có hiệu quả trong việc tìm và loại bỏ từng lỗi một. Trong thực tế nhiều lỗi được loại bỏ trước khi nó xảy ra.

Kiểm tra điều kiện trước và điều kiện sau

Một cách khác để tránh những lỗi là xác định những thuộc tính cần thiết đi trước (điều kiện trước) và sau (điều kiện sau) mã nguồn được thi hành. Các giá trị đầu vào phải thuộc một phạm vi cho trước là một ví dụ thông thường để kiểm tra điều kiện trước. Hàm tính giá trị trung bình của một mảng gồm *n* phần tử sẽ có lỗi nếu *n* nhỏ hơn hoặc bằng 0:

```
?       double avg(double a[], int n)
?       {
?               int i;
?               double sum;
?
?               sum = 0.0;
?               for (i = 0; i < n; i++)
?                   sum += a[i];
```

```
?           return sum / n;
?     ;
```

Hàm `avg` sẽ làm gì nếu `n` có giá trị `0`? Một mảng không có phần tử nào vẫn đầy đủ ý nghĩa mặc dù giá trị trung bình của nó không được xác định. Hàm `avg` có nên để cho hệ thống bắt những lỗi chia cho `0`? Chấm dứt? Thông báo lỗi? Trả về một giá trị bất kỳ nào đó? Còn nếu `n` có giá trị âm, thật vô lý nhưng vẫn có thể xảy ra? Như đã đề cập trong Chương 4, ta nên trả về giá trị `0` nếu `n` nhỏ hơn hay bằng `0`:

```
return n <= 0 ? 0.0 : sum/n;
```

nhưng đó không phải là câu trả lời đúng duy nhất.

Chắc chắn bạn sẽ nhận được câu trả lời sai khi bỏ việc kiểm tra tiền điều kiện. Một bài báo của *Scientific American* vào tháng 11 năm 1998 mô tả sự cố của một chiến hạm có chứa các tên lửa điều khiển *USS Yorktown*. Một thủy thủ đã nhầm lẫn khi nhập `0` cho một giá trị dữ liệu, dẫn đến lỗi chia cho `0`, lỗi xảy ra đã làm chao đảo và thậm chí làm hệ thống đẩy của tàu ngưng hoạt động. Yorktown bị chìm dần sau hai giờ vì chương trình đã không kiểm tra được tính hợp lệ của dữ liệu nhập vào.

Sử dụng các khẳng định (assert)

C và C++ cung cấp cơ chế khẳng định các điều kiện trước và sau dễ dàng trong thư viện `<assert.h>`. Nếu một sự khẳng định bị sai thì nó làm kết thúc chương trình, nó được dành riêng cho các tình huống xảy ra lỗi không ngờ tới hoặc không thể khắc phục được. Ta có thể làm tăng tính đúng đắn của mã nguồn bằng việc thêm khẳng định trước khi vào vòng lặp như sau:

```
assert(n > 0);
```

Nếu sự khẳng định bị vi phạm, nó sẽ làm cho chương trình ngừng thi hành với thông báo chuẩn sau:

```
Assertion failed: n > 0, file avgtest.c, line 7
Abort(crash)
```

Cơ chế khẳng định đặc biệt hữu ích trong việc kiểm tra tính đúng đắn của các thuộc tính của giao tiếp vì nó chú ý đến sự không nhất quán giữa hàm gọi thực hiện và hàm được gọi thực hiện, thậm chí có thể chỉ ra lỗi do hàm nào gây ra. Nếu sự khẳng định n>0 là sai khi hàm được gọi, nó chỉ ra rằng hàm gây ra lỗi là hàm gọi thực hiện chứ không chỉ là lỗi của hàm avg. Nếu một giao tiếp thay đổi nhưng ta quên sửa các hàm phụ thuộc vào nó, sự khẳng định có thể bắt những lỗi này trước khi nó gây ra lỗi thực sự.

Hoàn thiện chương trình

Một kỹ thuật hữu ích là thêm mã nguồn để xử lý các trường hợp "không thể xảy ra", các tình huống không logic vì một trường hợp nào đó không thể xảy ra nhưng nó vẫn có thể xảy ra (do một vài lỗi xảy ra ở một nơi nào đó trong chương trình). Chẳng hạn như, việc kiểm tra số lượng phần tử của mảng được gán trong biến avg có giá trị là 0 hoặc âm. Một ví dụ khác, chương trình xử lý việc xếp loại học sinh cần phải kiểm tra là không có điểm âm hoặc điểm quá lớn:

```
    if (grade < 0 || grade > 100) /* các trường hợp không
thể xảy ra */
                letter = '?';
    else if (grade >= 90)
                letter = 'A';
    else
          ...
```

Đây là một ví dụ về lập trình bảo vệ: bảo đảm chương trình có thể ngăn chặn việc dùng sai hoặc dữ liệu không hợp lệ. Các con trỏ NULL, chỉ số vượt ra ngoài giới hạn, phép chia cho 0, và những lỗi khác có thể được phát hiện sớm và cảnh báo. Lập trình bảo vệ có thể bắt được lỗi của phép chia cho không trong sự cố *Yorktown*.

Kiểm tra lỗi dựa vào giá trị trả về

Một cách ngăn chặn lỗi tổng quát thường được sử dụng là kiểm tra

lỗi dựa trên giá trị trả về của các hàm thư viện và các lời gọi hệ thống. Giá trị trả về của các hàm đọc dữ liệu như `fread`, `fscanf` luôn được kiểm tra lỗi, cũng như việc kiểm tra lỗi của hàm thực hiện mở tập tin như hàm `fopen`. Nếu thao tác đọc hoặc mở tập tin có lỗi, quá trình tính toán không thể nào thực hiện chính xác được.

Kiểm tra giá trị trả về của các hàm xuất kết quả như hàm `fprintf` hay `fwrite` sẽ bắt những lỗi khi thực hiện ghi dữ liệu lên tập tin không còn chỗ trống trên đĩa. Ta cũng cần phải kiểm tra giá trị trả về của hàm `fclose`, nó trả về giá trị là `EOF` nếu có bất cứ lỗi nào xảy ra trong quá trình thao tác hoặc trả về giá trị `0` nếu không có lỗi.

```
fp = fopen(outfile, "w");
while (…)           /* ghi kết quả ra file */
      fprintf(fp,…);
if (fclose(fp) == EOF) {       /* xảy ra lỗi? */
      /* xuất kết quả thông báo lỗi */
}
```

Những lỗi đầu ra có thể rất nghiêm trọng. Nếu tập tin đang được ghi là phiên bản mới của tập tin chứa dữ liệu quan trọng thì việc kiểm tra này sẽ giúp bạn không xóa tập tin cũ nếu tập tin mới chưa được ghi thành công.

Chi phí dành cho việc kiểm chứng khi viết chương trình sẽ thấp và cho kết quả đáng kể. Nghĩ đến việc kiểm chứng trong khi viết chương trình sẽ cho bạn những đoạn mã nguồn tốt hơn, vì khi đó bạn biết chính xác đoạn chương trình đó nên thực hiện công việc như thế nào. Nếu bạn đợi cho đến khi lỗi phát sinh thì có thể bạn đã quên đoạn chương trình đó làm việc ra sao. Vì áp lực công việc, bạn buộc phải tính lại nó, điều này làm mất thời gian, và việc sửa chữa sẽ không triệt để vì bạn không hoàn toàn hiểu lại đoạn chương trình đó như lúc ban đầu.

Bài tập 6-1. Hãy kiểm tra các ví dụ sau ở các biên của chúng, sau đó sửa lại (nếu cần thiết) theo những nguyên tắc chuẩn trong Chương 1 và những lời khuyên trong chương này.

(a) Tính giai thừa:

```
?    int factorial(int n)
?    {
?            int fac;
?            fac = 1;
?            while (n--)
?                    fac *= n;
?            return fac;
?    }
```

(b) In từng ký tự của một chuỗi trên từng dòng:

```
?    i = 0;
?    do {
?            putchar(s[i++]);
?            putchar('\n');
?    } while (s[i] != '\0');
```

(c) Sao chép một chuỗi từ nguồn sang đích:

```
?    void strcpy(char *dest, char *src)
?    {
?            int i;
?
?            for (i = 0; src[i] != '\0'; i++)
?                    dest[i] = src[i];
?    }
```

(d) Sao chép *n* ký tự từ chuỗi `s` sang chuỗi `t`:

```
?    void strncpy(char *t, char *s, int n)
?    {
?            while (n>0 && *s != '\0') {
```

```
?                    *t = *s;
?                    t++;
?                    s++;
?                    n--;
?              }
?      }
```

(e) So sánh các số:

```
?      if (i > j)
?            printf("%d is greater than %d.\n", i, j);
?      else
?            printf("%d is smaller than %d.\n", i, j);
```

(f) Kiểm tra lớp ký tự:

```
?      if (c >= 'A' && c <= 'Z') {
?            if (c <= ' L')
?                  cout <<"first half of alphabet";
?            else
?                  cout <<"second half of alphabet";
?      }
```

Bài tập 6-2. Như ta đã biết, sự cố năm 2000 dường như là sự cố có điều kiện biên lớn nhất.

(a) Ngày tháng nào bạn sử dụng để kiểm tra liệu hệ thống có làm việc vào năm 2000 không? Giả sử rằng việc thực hiện kiểm tra này tốn nhiều chi phí, bạn thực hiện việc kiểm tra theo trật tự nào sau ngày 1.1.2000?

(b) Bằng cách nào để kiểm chứng hàm chuẩn `ctime`, nó trả về một chuỗi đại diện cho ngày tháng theo dạng sau:

```
Fri Dec 31 23:58:27 EST 1999\n\0
```

Giả sử chương trình của bạn gọi hàm `ctime`. Bạn sẽ viết mã nguồn như thế nào để khắc phục lỗi cài đặt này.

(c) Hãy mô tả cách bạn kiểm chứng một chương trình in lịch như sau:

January 2000

S	M	Tu	W	Th	F	S
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31					

(d) Những biên thời gian nào khác mà bạn nghĩ bạn sử dụng trong hệ thống, và bằng cách nào bạn kiểm chứng chúng có được xử lý chính xác hay không?

6.2. Kiểm chứng có hệ thống

Việc kiểm chứng một chương trình một cách có hệ thống rất quan trọng, vì vậy bạn cần phải biết tại mỗi bước bạn cần kiểm chứng cái gì và kết quả bạn muốn làm gì. Bạn cần phải thực hiện một cách có thứ tự và bạn phải lưu lại những việc bạn đã làm để biết bạn đã làm được những gì.

Kiểm chứng tăng dần

Việc kiểm chứng cần phải đi đôi với việc xây dựng chương trình. Một người viết toàn bộ chương trình sau đó mới kiểm chứng tất cả sẽ gặp nhiều khó khăn và tốn nhiều thời gian hơn cách tiếp cận tăng dần. Viết một

phần của chương trình, kiểm chứng nó, viết thêm mã nguồn, kiểm chứng nó và cứ tiếp tục như vậy. Nếu bạn có hai phần được viết và kiểm chứng độc lập, khi bạn tích hợp chúng với nhau bạn chỉ cần kiểm tra chúng có hoạt động tốt với nhau không.

Chẳng hạn như, khi ta kiểm chứng chương trình *CSV* ở Chương 4, bước đầu tiên là viết mã nguồn vừa đủ để đọc giá trị nhập vào: điều này cho phép xử lý thực hiện dữ liệu nhập vào một cách đúng đắn. Bước tiếp theo là chia các dòng dữ liệu nhập này tại dấu phẩy. Khi những phần này hoạt động, ta chuyển sang xử lý phần trích dẫn, và dần dần tiến tới kiểm chứng mọi thứ.

Kiểm chứng những phần đơn giản trước

Cách tiếp cận tăng dần cũng được áp dụng để kiểm chứng các tính năng của chương trình. Việc kiểm chứng cần phải tập trung vào những phần đơn giản và những phần thường được thực hiện nhất của chương trình; chỉ khi chúng thực thi đúng bạn mới chuyển sang phần khác. Bằng cách này, ở mỗi giai đoạn, bạn đưa ra nhiều phần hơn để kiểm chứng và xây dựng với sự tin tưởng rằng nó sẽ hoạt động tốt. Những kiểm chứng đơn giản sẽ giúp tìm ra những lỗi đơn giản. Mỗi lần kiểm chứng góp phần tìm ra lỗi tiềm ẩn tiếp theo. Mặc dù khó phát hiện các lỗi về sau hơn nhưng việc sửa lỗi không hẳn là khó hơn.

Trong phần này, ta bàn về việc chọn những biện pháp kiểm chứng hiệu quả và thứ tự áp dụng chúng; trong hai phần tiếp theo, ta sẽ đề cập đến vấn đề làm sao để tự động hóa quy trình để việc thực hiện có hiệu quả. Đối với những chương trình nhỏ hoặc những hàm riêng lẻ, bước đầu tiên là việc mở rộng các kiểm chứng điều kiện biên ở phần trước và kiểm chứng có hệ thống các trường hợp nhỏ.

Giả sử rằng ta có một hàm thực hiện việc tìm kiếm nhị phân trên một mảng các số nguyên. Ta sẽ bắt đầu với những kiểm chứng sau, được sắp xếp theo mức độ phức tạp tăng dần:

- Tìm kiếm trên mảng không có phần tử nào

- Tìm kiếm trên mảng chỉ có một phần tử và giá trị cần tìm:
    - Nhỏ hơn phần tử duy nhất của mảng
    - Bằng phần tử duy nhất
    - Lớn hơn phần tử duy nhất
- Tìm kiếm trên mảng có hai phần tử và một giá trị cần tìm:
    - Kiểm tra tất cả năm vị trí có khả năng
- Kiểm tra việc thực hiện trên mảng gồm hai phần tử giống nhau và giá trị cần tìm
    - Nhỏ hơn giá trị trong mảng
    - Bằng giá trị trong mảng
    - Lớn hơn giá trị trong mảng
- Tìm kiếm trên mảng có ba phần tử tương tự như với hai phần tử
- Tìm kiếm trên mảng có bốn phần tử tương tự như với hai, ba phần tử

Nếu những kiểm chứng được thực hiện mà chương trình không có lỗi phát sinh thì có vẻ chương trình này chạy tốt nhưng nó vẫn cần được kiểm chứng nhiều hơn nữa.

Vì tập hợp những kiểm chứng này nhỏ nên ta có thể tiến hành thủ công, nhưng tốt hơn là nên tự động hóa quy trình này. Ta có thể quản lý chương trình điều khiển đơn giản. Nó đọc vào các dòng dữ liệu nhập bao gồm một khóa để tìm kiếm và kích thước của mảng. Đồng thời, nó sẽ tạo ra một mảng có kích thước đã cho chứa các giá trị 1, 3, 5, ... và thực hiện việc tìm khóa trên mảng này.

```
/* Hàm main của việc kiểm chứng hàm tìm kiếm nhị phân*/
int main(void)
{
        int i, key, nelem, arr[1000];
```

```
    while (scanf("%d %d", &key, &nelem) !=EOF {
        for (i = 0; i < nelem; i++)
            arr[i] = 2*i + 1;
        printf("%d\n", binsearch(key, arr, nelem));
    }
    return 0;
}
```

Điều này thật đơn giản nhưng nó cho thấy rằng việc thực hiện tự động thì hữu ích không phải cồng kềnh, và dễ dàng mở rộng về sau để thực hiện các kiểm chứng tương tự, và ít đòi hỏi con người can thiệp hơn.

Biết trước dữ liệu xuất

Trong tất cả các kiểm chứng, bạn cần phải biết kết quả đúng là gì; nếu không thì thật là lãng phí thời gian. Điều này dường như hiển nhiên, bởi vì trong nhiều chương trình, thật dễ dàng để xét xem chương trình có làm việc hay không. Chẳng hạn như, thao tác sao chép một tập tin thực hiện việc sao chép một tập tin hoặc không, dữ liệu xuất của một hàm sắp xếp có được sắp xếp hoặc không.

Quả thật là khó khăn để chỉ ra đặc điểm của hầu hết các chương trình - trình biên dịch (dữ liệu đầu ra có thực sự biên dịch đúng dữ liệu đầu vào không?), các thuật toán số học (kết quả có chứa những lỗi có thể bỏ qua hay không?), đồ họa (các pixel có ở đúng vị trí của nó không?)... Vì những lý do này, để đảm bảo dữ liệu xuất đúng ta thực hiện so sánh chúng với các giá trị đã biết.

- Để kiểm chứng một trình biên dịch, ta cần biên dịch và thực thi các tập tin thử nghiệm. Chương trình thử nghiệm ngược lại sẽ phát sinh các dữ liệu xuất, và các kết quả này sẽ được so sánh với những kết quả đã biết.
- Để kiểm chứng một chương trình số học, ta phát sinh các trường hợp thử nghiệm bao quát mọi trường hợp của thuật toán, từ những trường hợp đơn giản cho đến các trường hợp phức tạp. Nếu có thể, bạn viết

đoạn chương trình để kiểm tra tính đúng đắn các thuộc tính của dữ liệu xuất. Chẳng hạn như, giá trị xuất của một phép tính tích phân có thể được kiểm tra tính liên tục, và tính bị chặn của nó.

- Để kiểm chứng một chương trình đồ họa, nếu ta chỉ kiểm tra xem chương trình có thể vẽ một hình hộp hay không thì chưa đủ, thay vào đó ta phải đọc từ màn hình và kiểm tra xem các cạnh của chúng có nằm đúng vị trí hay không.

Nếu một chương trình có khả năng thực hiện theo chiều ngược lại thì cần kiểm tra rằng nó có thể khôi phục lại giá trị nhập hay không. Mã hóa và giải mã là hai quá trình ngược nhau, vì vậy nếu bạn có thể thực hiện mã hóa nhưng lại không thể giải mã được thì chứng tỏ có điều gì đó sai đã xảy ra. Tương tự, thuật toán nén không mất thông tin và giải nén là ngược nhau. Nhờ đó sau khi nén tập tin thì vẫn có thể giải nén tập tin đúng với ban đầu tập tin. Đôi khi có nhiều phương pháp đảo ngược; cần phải kiểm tra tất cả các trường hợp này.

Kiểm tra các thuộc tính lưu trữ

Nhiều chương trình dành lưu trữ các tính chất của dữ liệu nhập. Những công cụ như `wc` (đếm số hàng, số từ, số ký tự) và `sum` (tính tổng) có thể kiểm chứng được các dữ liệu xuất có cùng kích thước, có cùng số từ, có các byte giống nhau theo một thứ tự nào đó. Những chương trình so sánh các tập tin (`cmp`) hoặc đưa ra sự khác nhau của các tập tin (`diff`). Các chương trình này và các chương trình tương tự thích hợp với hầu hết các môi trường, và thu được kết quả tốt.

Một chương trình tuần tự theo byte có thể được dùng để kiểm tra sự lưu trữ dữ liệu và cũng để phát hiện những sự khác thường như những ký tự phi văn bản trong những tập tin văn bản. Sau đây là một phiên bản của chương trình `freq`:

```
#include <stdio.h>
#include <ctype.h>
#include <limits.h>
```

```
unsigned long count[UCHAR_MAX+1];
/*Hàm main của freq: hiển thị tần xuất theo từng byte*/
int main(void)
{
        int c;
        while ((c = getchar()) != EOF)
                count[c]++;
        for (c = 0; c <= UCHAR_MAX; c++)
                if (count[c] != 0)
                    printf("%.2x   %c  %lu\n",
                      c, isprint(c) ? c : '-', count[c]);
        return 0;
}
```

Các thuộc tính lưu trữ cũng có thể được kiểm tra bên trong một chương trình. Một hàm đếm các thành phần trong cấu trúc dữ liệu sẽ cho phép kiểm tra tính nhất quán giản đơn. Một bảng băm có thuộc tính là mỗi phần tử sau khi thêm vào bảng băm đều có thể truy xuất trở lại được. Ta có thể dễ dàng kiểm tra điều kiện này bằng cách tạo một hàm sao chép nội dung của bảng băm vào một tập tin hay một mảng. Tại bất kỳ thời điểm nào, số lượng các phần tử thêm vào trong cấu trúc dữ liệu trừ cho số lượng phần tử bị xoá bỏ phải bằng với số lượng phần tử hiện có trong cấu trúc dữ liệu đó. Ta dễ dàng kiểm tra được điều kiện này.

So sánh các quá trình cài đặt độc lập

Các quá trình cài đặt độc lập của một thư viện hay của một chương trình phải cho ra cùng kết quả. Chẳng hạn như, trong hầu hết các trường hợp, hai trình biên dịch sẽ phải tạo ra các chương trình hoạt động giống nhau trên cùng một máy.

Đôi khi một kết quả có thể được tính theo hai cách khác nhau, hoặc bạn có thể viết một phiên bản bình thường của chương trình sử dụng phép

so sánh chậm nhưng thực hiện độc lập. Nếu hai chương trình độc lập cho ra cùng kết quả thì có thể khẳng định cả hai chương trình đó đều đúng; nhưng nếu hai chương trình đó cho ra các kết quả khác nhau thì có ít nhất một chương trình bị sai.

Kiểm tra tính bao quát của việc kiểm chứng

Một mục tiêu của kiểm chứng là đảm bảo mọi câu lệnh trong chương trình đều được thi hành tại một thời điểm nào đó trong suốt quá trình kiểm chứng; việc kiểm chứng không thể xem là hoàn tất nếu mỗi dòng của chương trình chưa được kiểm tra ít nhất một lần. Tính bao quát hoàn toàn thường khó đạt được. Ngay cả khi bỏ qua các trường hợp "không thể xảy ra", việc sử dụng các dữ liệu nhập thông thường để buộc chương trình thực hiện các câu lệnh nào đó cũng gặp nhiều khó khăn.

Có một số phần mềm thương mại dùng để đo lường tính bao quát. Nó cung cấp cách tính tần xuất thực hiện của mỗi câu lệnh trong chương trình, và cho biết tính bao quát trong các lần kiểm chứng cụ thể.

Ta đã kiểm chứng chương trình *Markov* ở Chương 3 bằng cách kết hợp những kỹ thuật này. Phần cuối cùng của chương này mô tả chi tiết các quá trình kiểm chứng đó.

Bài tập 6-3. Hãy mô tả cách kiểm chứng chương trình `freq`.

Bài tập 6-4. Hãy thiết kế và cài đặt một phiên bản của chương trình `freq` để đo tần số của các kiểu giá trị dữ liệu khác, chẳng hạn các số nguyên 32 bit hoặc các số thực dấu chấm động. Bạn có thể tạo một phiên bản của chương trình để xử lý một số kiểu dữ liệu khác nhau hay không?

6.3. Kiểm chứng tự động

Quá trình kiểm chứng chương trình thủ công rất nhàm chán và không đáng tin cậy; ba quá trình kiểm chứng theo đúng nghĩa gồm: thực hiện việc kiểm chứng nhiều lần, nhiều bộ dữ liệu nhập và nhiều lần so sánh dữ liệu xuất. Vì vậy, quá trình kiểm chứng cần được thực hiện bằng chương trình, nó giúp ta không bị mệt mỏi cũng như tránh được sự bất cẩn có thể

xảy ra. Ta nên viết một chương trình bao gồm tất cả các quá trình kiểm chứng, vì vậy, một bộ chương trình kiểm chứng đầy đủ có thể được thực thi (theo cả nghĩa đen lẫn nghĩa bóng) bằng cách nhấn một nút duy nhất. Bộ chương trình kiểm chứng càng dễ chạy bao nhiêu thì nó càng được sử dụng thường xuyên bấy nhiêu, và càng ít khi bỏ qua nó khi có ít thời gian. Ta viết một bộ kiểm chứng để kiểm tra tất cả chương trình đã viết trong cuốn sách này, và chạy nó mỗi khi có sự thay đổi; các thành phần của bộ phần mềm này chạy một cách tự động sau khi biên dịch thành công.

Tự động hóa việc kiểm chứng lùi

Một dạng cơ bản của tự động là việc kiểm chứng lùi, nó thực hiện tuần tự các kiểm chứng so sánh phiên bản mới với những phiên bản cũ tương ứng. Khi sửa lỗi, khuynh hướng tự nhiên là kiểm tra việc sửa đổi đó có đúng hay không; thật dễ dàng để nhận ra được khả năng các thao tác sửa đổi làm hỏng những thứ khác. Mục đích của việc kiểm chứng lùi là đảm bảo việc sửa lỗi sẽ không làm ảnh hưởng những phần khác trừ khi chúng ta muốn như thế.

Một số hệ thống có nhiều công cụ trợ giúp tự động hóa; các ngôn ngữ *script* cho phép viết các đoạn *script* để thực hiện kiểm chứng tuần tự. Trong hệ điều hành Unix, các trình thao tác trên tập tin như `cmp` và `diff` so sánh các dữ liệu xuất, `sort` sắp xếp các phần tử, `grep` kiểm chứng dữ liệu xuất; `wc`, `sum` và `freq` tổng kết dữ liệu xuất. Nếu kết hợp chúng lại với nhau thì sẽ dễ dàng tạo ra được một mô hình kiểm chứng, nó có thể không kiểm chứng được đầy đủ cho các chương trình lớn nhưng hầu như đủ cho các chương trình được bảo trì bởi từng cá nhân hay một nhóm nhỏ.

Đây là bản mô tả cho việc kiểm chứng lùi một chương trình xóa gọi là `ka`. Nó chạy phiên bản cũ (`old_ka`) và một phiên bản mới (`new_ka`) trên một số lượng lớn các tập tin dữ liệu khác nhau, và đưa ra thông báo về trường hợp mà dữ liệu xuất không đồng nhất. Nó được viết trên giao diện *shell* của Unix nhưng có thể chuyển qua Perl hoặc các ngôn ngữ mô tả khác:

```
for i in ka_data.*              # lặp trên các file dữ
```

```
liệu thu
      do
            old_ka $i > out1          # chạy phiên bản cũ
            new_ka $i > out2          # chạy phiên bản mới
            if ! cmp -s out1 out2     # so sánh các file dữ
liệu xuất
            then
                  echo $i : BAD       # khác nhau: xuất thông
báo lỗi
            fi
      done
```

Đoạn chương trình kiểm chứng thường chạy một cách thầm lặng, việc xuất các thông báo chỉ khi có lỗi xuất hiện. Ta có thể in ra tên tập tin đang được kiểm chứng hoặc đưa ra một thông báo lỗi nếu có lỗi phát sinh. Những sự báo hiệu này giúp xác định các lỗi như vòng lặp vô hạn, hoặc một đoạn chương trình kiểm chứng chạy sai trên những trường hợp đúng, nhưng khi thêm những báo hiệu khác vào thì rất phiền phức nếu việc kiểm chứng thực hiện đúng.

Tham số `-s` buộc `cmp` đưa ra trạng thái nhưng không đưa ra output. Nếu các tập tin so sánh bằng nhau, `cmp` trả về `true`, nên `!cmp` là `false`, và không có gì được in ra. Tuy nhiên, nếu giá trị xuất mới và cũ khác nhau thì `cmp` trả về `false`, khi đó tên tập tin và lời cảnh báo được in ra.

Trong việc kiểm chứng lùi cần phải giả sử rằng phiên bản trước của chương trình cho ra kết quả đúng. Điều này cần được kiểm chứng cẩn thận tại thời điểm bắt đầu, và sự bất biến phải được duy trì một cách thận trọng. Nếu có một kết quả sai bên trong việc kiểm chứng lùi thì rất khó để phát hiện, và mọi thứ phụ thuộc vào nó sẽ không chính xác. Do đó, ta cần phải kiểm tra chính việc kiểm chứng lùi một cách có định kỳ để đảm bảo nó vẫn hợp lệ.

Tạo ra những kiểm chứng độc lập

Các kiểm chứng độc lập có chứa giá trị nhập và giá trị xuất mong đợi sẽ bổ sung cho việc thực hiện kiểm chứng lùi. Nhiều ngôn ngữ có cấu trúc được kiểm chứng bằng cách chạy thử một vài chương trình nhỏ với các bộ dữ liệu thử đặc biệt và thấy rằng dữ liệu xuất ra đúng. Phần kiểm chứng hỗn hợp sau đây sẽ trình bày cách kiểm chứng một biểu thức có độ phức tạp tăng dần. Nó chạy trên phiên bản mới của ngôn ngữ Awk *(newawk)* thực hiện kiểm chứng một chương trình ngắn dữ liệu xuất được ghi vào một tập tin, kết quả đúng được ghi vào một tập tin khác bằng hàm echo, sau đó so sánh 2 tập tin, thông báo lỗi nếu 2 tập tin này khác nhau.

```
    # field increment test : $i++ means  ($i)++, not $(i++)


    echo 3 5 | newawk '{i =1; print $i++; print $1 ,i}'
>out1
    echo '3
    4 1' >out2                # kết quả đúng
    if ! cmp -s out1 out2     # nếu việc so sánh file là khác
nhau
    then
        echo'BAD: field increment test failed'
    fi
```

Câu lệnh đầu tiên là một phần dữ liệu nhập của việc kiểm chứng; nó giải thích việc kiểm chứng đang thực hiện những gì.

Ta có thể thực hiện việc kiểm chứng nhiều lần với chi phí vừa phải. Đối với những biểu thức đơn giản, ta tạo ra một ngôn ngữ chuyên biệt đơn giản để mô tả các kiểm chứng, giá trị dữ liệu nhập, giá trị dữ liệu xuất mong muốn. Sau đây là một mô tả ngắn cho việc kiểm chứng một vài cách thể hiện số 1 trong Awk:

```
try {if ($1 == 1) print "yes"; else print "no"}
1       yes
1.0     yes
1E0     yes
0.1E1   yes
10E-1   yes
01      yes
+1      yes
10E-2   no
10      no
```

Dòng đầu tiên là chương trình cần được kiểm chứng (tất cả những từ sau từ `try`). Mỗi dòng tiếp theo là một tập các đầu vào và đầu ra mong muốn cách nhau bởi dấu *tab*. Kiểm chứng đầu tiên cho thấy nếu giá trị vào là 1 thì giá trị xuất là `yes`. Bảy hàng kiểm chứng đầu tiên đều in ra `yes` và hai hàng còn lại in ra `no`.

Một chương trình Awk chuyển mỗi kiểm chứng thành một chương trình Awk hoàn chỉnh, sau đó chạy với những dữ liệu nhập, so sánh dữ liệu xuất thực tế với dữ liệu xuất mong muốn, và chỉ thông báo những trường hợp có câu trả lời sai.

Những cơ chế tương tự được sử dụng để kiểm chứng những biểu thức thông dụng và những câu lệnh thay thế. Một số ngôn ngữ được dùng để viết kiểm chứng tạo ra các mô tả cho việc kiểm chứng một cách dễ dàng; đó chính là việc dùng một chương trình để viết ra chương trình thực hiện việc kiểm chứng cho chương trình ở mức cao (Chương 9 đề cập nhiều về các ngôn ngữ đặc tả và dùng một số chương trình để viết các chương trình khác).

Tóm lại, có khoảng 1000 kiểm chứng Awk; toàn bộ tập này có thể được chạy với từng lệnh đơn, và nếu mọi thứ đều tốt thì sẽ không xuất ra gì

cả. Bất cứ lúc nào khi một chức năng được thêm vào hay một lỗi được sửa, các kiểm chứng mới được thêm vào để chứng tỏ các thao tác vừa mới được thay đổi là đúng? Mỗi khi một chương trình thay đổi, ngay cả theo những cách thông thường, toàn bộ các kiểm chứng sẽ được thực hiện; nó chỉ mất vài phút thôi. Nhưng đôi khi bắt được những lỗi hoàn toàn không ngờ trước được, và tiết kiệm được nhiều thời gian.

Bạn nên làm gì khi bạn phát hiện ra một lỗi? Nếu nó không được tìm thấy bởi những thao tác kiểm chứng có sẵn, bạn phải tạo một kiểm chứng mới để phát hiện vấn đề và thực hiện việc kiểm chứng bằng cách chạy nó trên phiên bản có lỗi. Một lỗi có thể cần nhiều thao tác kiểm chứng hoặc phải kiểm tra toàn bộ các lớp mới. Hoặc có thể thêm vào những đoạn chương trình bảo vệ để có thể bắt được những lỗi bên trong chương trình.

Đừng bao giờ bỏ qua giai đoạn kiểm chứng. Nó có thể giúp bạn chọn thông báo lỗi đưa ra hợp lý, hoặc mô tả những gì vừa được sửa chữa. Lưu vết các lỗi, các thay đổi và sửa chữa. Trong hầu hết các chương trình thương mại, những bản ghi chép đó là bắt buộc. Đối với các chương trình cá nhân thì những bản này tốn ít chi phí và cho ra kết quả tốt.

Bài tập 6-5. Hãy thiết kế một bộ kiểm chứng cho hàm `printf`, sử dụng các cơ chế hỗ trợ có thể.

6.4. Mô hình kiểm chứng

Những thảo luận của chúng ta phần lớn dựa vào việc kiểm chứng các chương trình độc lập trong dạng hoàn chỉnh của nó. Tuy nhiên, điều này không phải là một loại kiểm chứng tự động, cũng không phải là cách kiểm chứng một chương trình trong lúc xây dựng, nhất là khi bạn thuộc một nhóm lập trình. Nó cũng không là cách hiệu quả nhất để kiểm chứng các thành phần nhỏ ẩn trong các thành phần lớn hơn.

Để kiểm chứng một bộ phận độc lập, ta cần phải tạo ra một số loại mô hình để cung cấp đầy đủ sự hỗ trợ cũng như các giao tiếp cho tất cả các phần có liên quan của hệ thống mà việc kiểm chứng được thực hiện. Như chúng ta đã trình bày một ví dụ nhỏ để kiểm chứng việc tìm kiếm nhị phân ở

phần đầu của chương này.

Thật dễ dàng xây dựng một mô hình để kiểm chứng các hàm toán học, các hàm liên quan đến chuỗi, các hàm sắp xếp .... vì việc tạo mô hình bao gồm phần lớn sự thiết lập các tham số đầu vào, gọi các hàm sẽ được kiểm chứng, sau đó kiểm chứng kết quả. Nhưng để tạo ra một mô hình để kiểm chứng từng phần của một chương trình hoàn chỉnh thì quả thật là một việc tốn nhiều thời gian và công sức.

Để minh họa, ta sẽ xây dựng một kiểm chứng cho hàm `memset`, một trong những hàm liên quan đến bộ nhớ trong thư viện chuẩn của C/C++. Những hàm này thường được viết bằng hợp ngữ cho từng loại máy khác nhau bởi vì sự thực hiện của chúng rất quan trọng. Tuy nhiên, nếu chúng càng được thực hiện cẩn thận thì chúng lại càng sai do vậy chúng cần được kiểm chứng một cách kỹ lưỡng.

Bước đầu tiên là cung cấp những phiên bản C đơn giản nhất có thể chạy được; những phiên bản này cho thấy một sự đánh giá cho việc thực thi và quan trọng hơn là tính chính xác. Khi chuyển sang một môi trường mới chỉ lấy sang những phiên bản đơn giản và chúng chỉ được dùng khi đã được hiệu chỉnh cho phù hợp.

Hàm `memset(s,c,n)` gán ký tự `c` vào *n byte* trong bộ nhớ bắt đầu từ đầu địa chỉ `s`, trả về `s`. Hàm này được thực hiện dễ dàng nếu không chú ý tới tốc độ:

```
/* Hàm memset : gán kí tự c vào n byte đầu tiên của s*/
void *memset(void *s , int c, size_t n)
{
    size_t i;
    char *p;
    p = (char *) s;
    for (i = 0; i < n; i++)
        p[i] = c;
```

```
        return s;
    }
```

Nhưng khi quan tâm đến vấn đề tốc độ thì những mẹo như viết đầy đủ các *word* 32 hoặc 64 bit tại cùng thời điểm được sử dụng. Những điều này có thể gây ra lỗi vì thế bắt buộc phải có những kiểm chứng tổng quát.

Việc kiểm chứng dựa trên sự kết hợp toàn diện và kiểm tra điều kiện biên tại những nơi dễ gây ra lỗi. Đối với hàm `memset`, biên của nó bao gồm các giá trị hiển nhiên của `n` như 0, 1, 2 nhưng cũng bao gồm các giá trị là lũy thừa của hai hoặc những giá trị lân cận, bao gồm các giá trị nhỏ và lớn như 2^{16}, nó thích hợp với giá trị biên tự nhiên của nhiều máy một *word* có 16 *bit*. Lũy thừa của hai cần được chú ý bởi vì một cách để làm cho hàm `memset` thực hiện nhanh hơn là gán nhiều *byte* cùng lúc, điều này được thực hiện bằng các lệnh đặc biệt hoặc bằng cách lưu một *word* tại một thời điểm thay vì chỉ lưu một *byte*. Một cách tương tự, ta muốn kiểm tra mảng với những cách sắp xếp khác nhau trong trường hợp có lỗi phát sinh dựa trên địa chỉ bắt đầu hoặc chiều dài mảng. Ta sẽ đặt mảng đích vào trong một mảng có kích thước lớn hơn, nhờ đó tạo ra một vùng đệm hoặc một vùng an toàn trên mỗi chiều và chỉ ra một cách dễ dàng để thay đổi cách sắp xếp.

Ta cũng muốn kiểm chứng nhiều giá trị của `c`, bao gồm 0, `0x7F` (giá trị có dấu lớn nhất), `0x80` và `0xFF` (tìm những lỗi tiềm ẩn có thể liên quan đến các ký tự có dấu và không dấu), và các giá trị lớn hơn một *byte* (để chắc chắn chỉ có một *byte* được sử dụng). Ta cũng cần phải khởi tạo bộ nhớ theo vài giá trị khác những giá trị đó để có thể kiểm tra hàm `memset` có ghi ra ngoài vùng hợp lệ hay không.

Chúng ta có thể dùng một sự cài đặt đơn giản như một tiêu chuẩn so sánh trong việc kiểm chứng hai mảng, sau đó so sánh các sự thực hiện trên sự kết hợp của `n`, `c` và *offset (địa chỉ tương đối)* trong mảng:

```
    big = maximum left margin + maximum n + maximum right
margin
```

```
        s0 = malloc(big)
        s1 = malloc(big)
        for each combination of test parameters n, c and
offset:
                set all of s0 and s1 to know pattern
                run slow memset(s0 + offset, c, n)
                run fast memset(s1 + offset, c, n)
                check return values
                compare all of s0 and s1 byte by byte
```

Một lỗi gây ra do hàm `memset` là ghi bên ngoài giới hạn của mảng thì ảnh hưởng đến những *byte* ở vị trí bắt đầu hoặc kết thúc của mảng, vì vậy dành ra vùng đệm để dễ dàng phát hiện những *byte* hư hỏng và để phát hiện ra lỗi ghi đè lên một số phần khác của chương trình. Để kiểm chứng việc ghi ra ngoài giới hạn đã định, ta so sánh tất cả các *byte* của `s0` và `s1`, chứ không chỉ *n byte* cần ghi.

Như vậy, tập các kiểm chứng hợp lệ phải bao gồm tất cả các kết hợp sau:

```
                offset = 10, 11, …, 20
                c = 0, 1, 0x7F, 0x80, 0xFF, 0x11223344
                n = 0, 1, 2, 3, 4, 5, 7, 8, 9, 15, 16, 17,
                    31, 32, 33, …, 65535, 65536, 65537
```

Giá trị của `n` bao gồm ít nhất $2^i - 1$, 2^i, $2^i + 1$ với `i` từ 0 đến 16.

Những giá trị này không cần phải kết với phần chính của mô hình kiểm tra, nhưng nó cần xuất hiện trong mảng được tạo một cách thủ công hay bằng chương trình. Phát sinh ra chúng một cách tự động sẽ tốt hơn: điều này làm cho dễ định rõ các lũy thừa của hai hoặc có được nhiều địa chỉ

offset và nhiều ký tự hơn.

Những kiểm chứng này làm cho hàm `memset` làm việc một cách đúng đắn, tốn ít thời gian để tạo ra, thực hiện độc lập, vì có khoảng 3500 trường hợp cho các giá trị ở trên. Những kiểm chứng này có thể mang chuyển được, vì vậy chúng ta có thể chuyển nó sang một môi trường khác nếu cần thiết.

Những hàm như `memset` có thể có được các kiểm chứng một cách toàn diện vì chúng đơn giản đến nỗi một người có thể chứng minh rằng tất cả trường hợp kiểm chứng có thể được thực hiện thông qua mã nguồn, vì thế nó có thể được kiểm chứng một cách hoàn chỉnh. Chẳng hạn như, có thể kiểm chứng hàm `memmove` trong mọi cách kết hợp chồng chéo, theo hướng, và theo hàng. Ta không có được sự kiểm chứng toàn diện trong trường hợp phải kiểm chứng tất cả những thao tác sao chép có thể, nhưng đó là sự kiểm chứng toàn diện đối với mỗi loại khác nhau của dữ liệu nhập.

Bài tập 6-6. Hãy tạo ra mô hình kiểm chứng cho hàm `memset` theo những đường thẳng mà ta chỉ định.

Bài tập 6-7. Hãy tạo ra những kiểm chứng cho phần còn lại của họ `mem...`.

Bài tập 6-8. Hãy định rõ các cơ chế kiểm chứng cho các hàm số học như `sqrt`, `sin`... trong thư viện `math.h`. Giá trị dữ liệu nhập có ý nghĩa gì? Những việc kiểm tra độc lập nào có thể được thực thi.

Bài tập 6-9. Hãy xác định các cơ chế kiểm chứng các hàm của họ `str...` trong C, chẳng hạn như `strcmp`. Một số hàm này, đặc biệt các hàm như `strtok` và `strcspn` thì phức tạp đáng kể so với họ `mem...` vì thế các kiểm chứng phức tạp hơn sẽ được gọi thực hiện.

6.5. Kiểm chứng tự động với tập dữ liệu có giá trị lớn

Một kỹ thuật kiểm chứng hiệu quả khác là phát sinh tập dữ liệu nhập có giá trị lớn. Tập dữ liệu nhập này gây khó khăn cho chương trình hơn là dữ liệu do người dùng đưa vào. Giá trị lớn có khuynh hướng tạo ra lỗi vì

nhiều dữ liệu nhập có thể làm tràn các vùng đệm bộ nhớ nhập liệu, mảng, biến đếm, và hiệu quả trong việc phát hiện ra những bộ lưu trữ có kích thước cố định chưa được kiểm tra trong chương trình. Hầu hết, mọi người có khuynh hướng tránh những trường hợp *không bình thường* chẳng hạn như dữ liệu nhập vào rỗng, hoặc nó không theo trật tự nào cả hoặc vượt quá giới hạn, và cũng không muốn tạo ra tên quá dài hoặc những giá trị dữ liệu khổng lồ. Ngược lại, máy tính, tạo ra những dữ liệu xuất đúng theo chương trình và không tránh bất kỳ trường hợp nào hết.

Để minh họa, sau đây là một dòng của dữ liệu xuất ra bởi trình biên dịch Microsoft Visual C++ 5.0 khi biên dịch bản cài đặt trên C++ STL của *Markov*, nó đã được hiệu chỉnh có dạng sau:

```
        xtree(114) : warning C4786:
'std::_Tree<std::deque<std::
        basic_string<char,
std::char_traits<char>,std::allocator
        <char>>,std::allocator<std::basic_string<char,std
::
        ...1420 characters omitted
        allocator <char>>>>>>::iterator' : indentifier
was
        truncated to '255' characters in the debug
information
```

Trình biên dịch đã cảnh báo rằng chương trình đã phát sinh một tên biến có chiều dài 1594 ký tự, trong khi chỉ có 255 ký tự được dành cho thông tin này. Không phải tất cả các chương trình đều tự bảo vệ đối với những chuỗi có chiều dài bất thường như vậy.

Dữ liệu nhập ngẫu nhiên (không cần thiết phải hợp lý) là một cách khác để tấn công vào một chương trình với hy vọng sẽ làm hỏng một cái gì đó. Chẳng hạn như, một số trình biên dịch thương mại C được kiểm chứng với các giá trị phát sinh ngẫu nhiên chứ không là những chương trình có cú

pháp hợp lệ. Trong trường hợp này (trong C chuẩn), mẹo là dùng các đặc tả của vấn đề để xử lý một chương trình đưa ra những dữ liệu kiểm chứng hợp lý nhưng kỳ lạ.

Những kiểm chứng này dựa trên việc phát hiện các kiểm tra gắn liền với chương trình, bởi vì không thể chứng minh rằng chương trình tạo ra các dữ liệu xuất đúng; mục đích là tạo ra một sự hư hỏng hoặc một trường hợp "không thể xảy ra" hơn là tìm ra dễ dàng các lỗi. Nó cũng là một cách tốt để kiểm chứng mã nguồn xử lý lỗi có làm việc hay không. Với dữ liệu nhập thích hợp, hầu hết các lỗi không xảy ra và đoạn mã nguồn xử lý chúng không làm việc; bản chất tự nhiên, lỗi có xu hướng ẩn ở những nơi kín đáo. Tuy nhiên, trong một số trường hợp loại kiểm chứng này cũng làm giảm bớt được lỗi: nó tìm ra được những vấn đề không giống như những gì xảy ra trong thực tế nên ta có thể không cần phải tốn công để sửa chữa.

Một số việc kiểm chứng dựa vào dữ liệu nhập có chủ định. Việc phá vỡ sự an toàn thường dùng các giá trị nhập lớn hoặc không hợp lệ gây nên việc ghi đè lên những dữ liệu quan trọng, do đó việc tìm ra những điểm yếu như vậy là vô cùng đáng giá. Một số hàm thư viện chuẩn có thể bị hư hỏng vì những loại tấn công này. Chẳng hạn như, hàm thư viện chuẩn `gets` không đưa ra kích thước giới hạn của dòng dữ liệu nhập vào, vì vậy không nên sử dụng nó; thay vào đó nên dùng hàm `fgets(buf, sizeof(buf), stdin)`. Hàm `scanf("%s", buf)` nguyên mẫu cũng không giới hạn chiều dài của dữ liệu nhập, vì vậy nó chỉ được dùng với kiểu dữ liệu có kích thước rõ ràng, chẳng hạn `scanf("%20s", buf)`. Trong phần 3.3, ta đã trình bày cách đánh địa chỉ cho vấn đề này đối với *buffer* có kích thước tổng quát.

Bất kỳ hàm nào có thể nhận giá trị bên ngoài chương trình một cách trực tiếp hoặc gián tiếp, thì cần phải kiểm tra tính hợp lệ của dữ liệu nhập trước khi sử dụng chúng. Chương trình sau đây đọc vào một số nguyên do người dùng nhập vào, và đưa ra thông báo nếu số nguyên quá lớn. Nhằm minh họa cách khắc phục những vấn đề gây ra của hàm `gets`, nhưng giải pháp này không phải lúc nào cũng thực hiện được.

```
?       #define MAXNUM 10
?
?       int main(void)
?       {
?               char num[MAXNUM];
?
?               memset(num, 0, sizeof(num));
?               printf("Type a number");
?               gets(num);
?               if (num[MAXNUM-1] != 0)
?                       printf("Number too big.\n");
?               /* ... */
?       }
```

Nếu số nhập vào dài hơn 10 ký số thì nó sẽ ghi đè một giá trị khác không lên số không cuối cùng trong mảng num, và theo giả thiết điều này được ngăn chặn sau khi kiểm tra giá trị trả về của hàm gets. Nhưng thật không may là điều này không đủ. Những người cố tình có thể nhập vào một chuỗi dài hơn sẽ ghi đè lên một số dữ liệu quan trọng, có thể là giá trị trả về của hàm gets, vì vậy chương trình không trở về thực hiện câu lệnh if và thay vào đó là thực hiện một điều gì đó bất hợp lệ. Như vậy, dữ liệu nhập không được kiểm chứng loại này là vấn đề tiềm ẩn ảnh hưởng đến sự an toàn.

Chương trình phân tích các dạng của HTML cũng có thể bị ảnh hưởng khi lưu trữ chuỗi có kích thước lớn trong mảng nhỏ:

```
?       static char query[1024];
?
?       char *read_form(void)
```

```
?       {
?               int qsize;
?
?               qsize = atoi(getenv("CONTENT_LENGTH"));
?               fread(query, qsize, 1, stdin);
?               return query;
?       }
```

Đoạn mã nguồn giả sử rằng dữ liệu đưa vào không bao giờ vượt quá 1024 *byte*, tương tự như hàm `gets`, nó dẫn đến tình trạng tràn vùng đệm.

Những sự tràn tương tự cũng gây ra nhiều rắc rối. Nếu một số nguyên bị tràn một cách không biết được thì kết quả có thể rất tai hại. Xét sự khai báo sau:

```
?       char *p;
?       p = (char *) malloc(x * y * z);
```

Nếu tích số của `x`, `y`, `z` bị tràn, lời gọi hàm `malloc` có thể tạo ra một mảng có kích thước hợp lý, nhưng `p[x]` có thể tham chiếu tới một vùng nhớ bên ngoài vùng đã khai báo. Giả sử mỗi số nguyên là 16 bit và `x`, `y`, `z` đều có giá trị 41. Như vậy `x*y*z` là 68921, bằng 3385 modulo 2^{16} . Do đó việc gọi hàm `malloc` chỉ định vị 3385 *byte;* bất kỳ sự tham chiếu nào với một chỉ số ngoài giá trị đó sẽ vượt qua ngoài giới hạn.

Các giá trị dữ liệu nhập ở dạng nhị phân đôi khi làm hỏng các chương trình chỉ nhận các giá trị dữ liệu nhập ở dạng văn bản, đặc biệt nếu ta giả sử rằng dữ liệu nhập thuộc tập các ký tự ASCII 7-bit. Đó là một bài học bổ ích khi bỏ qua các giá trị dữ liệu nhập ở dạng nhị phân đối với các chương trình chỉ nhận giá trị dữ liệu ở dạng văn bản.

Những trường hợp kiểm chứng tốt có thể áp dụng trong nhiều chương trình. Chẳng hạn như, bất kỳ chương trình nào đọc tập tin đều phải kiểm chứng được tập tin có rỗng hay không. Các chương trình đọc tập tin

văn bản phải thực hiện kiểm chứng với các tập tin nhị phân. Các chương trình đọc một dòng văn bản phải thực hiện kiểm chứng với một dòng có kích thước rất lớn, dòng trống hoặc dòng không có ký tự kết thúc chuỗi. Thật là một ý tưởng hay để hình thành một bộ sưu tầm các tập tin để thực hiện kiểm chứng một cách thuận tiện, khi đó bạn có thể kiểm chứng bất cứ chương trình nào mà không phải tạo lại những kiểm chứng. Hoặc viết một chương trình để tạo ra những kiểm chứng theo yêu cầu.

Bài tập 6-10. Hãy tạo ra một tập tin mà nó làm hỏng chương trình xử lý văn bản, trình biên dịch hoặc những chương trình khác mà bạn thích nhất.

6.6. Những mẹo trong việc kiểm chứng

Những người kiểm chứng nhiều kinh nghiệm sử dụng nhiều mẹo và kỹ thuật để làm cho công việc có hiệu quả hơn: mục này bao gồm nhiều mẹo được ưa chuộng.

Chương trình nên kiểm tra các biên của mảng (nếu ngôn ngữ lập trình không thực hiện việc này) nhưng mã nguồn có thể không thực hiện kiểm chứng nếu kích thước mảng rất lớn so với số phần tử nhập vào. Để thực hiện việc kiểm chứng, tạm thời làm cho kích thước của mảng thật nhỏ để dễ dàng tạo ra một lượng lớn các trường hợp kiểm chứng. Ta sử dụng một mẹo có liên quan đến đoạn mã nguồn mở rộng mảng ở Chương 2 và trong thư viện *CSV* ở Chương 4. Thật ra, ta đặt các giá trị khởi tạo nhỏ đúng vị trí, và chi phí cho việc khởi tạo là không đáng kể.

Hàm băm trả về một hằng số, do đó mỗi phần tử được đặt vào vị trí trong một bảng băm. Điều này minh họa cho cơ chế móc xích, nó cũng cung cấp sự chỉ định trong những trường hợp xấu nhất.

Hãy viết một phiên bản của hàm cấp phát bộ nhớ và phát hiện ra lỗi sớm, để kiểm chứng đoạn mã nguồn của bạn có lỗi thiếu bộ nhớ. Phiên bản này trả về giá trị `NULL` sau 10 lần gọi:

```
        /* Hàm testmalloc: trả về giá trị NULL sau 10 lần gọi
*/
```

```
void *testmalloc(size_t n)
{
        static int count = 0;

        if (++count > 10)
                return NULL;
        else
                return malloc(n);
}
```

Trước khi chạy đoạn mã nguồn của bạn, nên vô hiệu hóa các giới hạn kiểm chứng gây ảnh hưởng đến việc thực hiện. Ta đã bắt theo dấu lỗi thực thi của trình biên dịch đối với một hàm băm mà luôn trả về giá trị không vì mã nguồn kiểm chứng chưa được cài đặt.

Ta nên khởi tạo các biến và máng với những giá trị khác biệt nào đó, thay vì giá trị mặc nhiên thường được chọn là không; nếu bạn truy cập ra ngoài giới hạn hoặc tham chiếu đến một biến chưa được khởi tạo thì bạn phải chú ý đến nó. Hằng số `0xDEADBEEF` thì dễ được nhận ra trong một trình gỡ rối, trình cấp phát bộ nhớ đôi khi dùng những giá trị này để tìm ra những dữ liệu chưa được khởi tạo.

Thay đổi các cách kiểm chứng của bạn, đặc biệt khi thực hiện các kiểm chứng nhỏ một cách thủ công, nếu không ta dễ dàng cảm thấy nhàm chán khi luôn kiểm chứng cùng một điều, và bạn có thể không biết được những thứ khác đã hư hỏng.

Dừng tiếp tục cài đặt các tính năng mới hoặc kiểm chứng một tính năng sẵn có khi chương trình đang bị lỗi; nó có thể ảnh hưởng đến kết quả kiểm chứng.

Kết quả kiểm chứng phải bao gồm tất cả các thiết lập tham số nhập, nhờ đó các kết quả kiểm chứng có thể được tái tạo một cách chính xác. Nếu chương trình của bạn có tập dữ liệu nhập từ những số ngẫu nhiên thì ta có được một cách để thiết lập và in ra kết quả được chọn, nó không phụ thuộc vào các kiểm chứng ngẫu nhiên. Hãy đảm bảo rằng tập dữ liệu đầu vào và đầu ra tương ứng được xác định thích hợp, để chúng có thể tái tạo được.

Việc cung cấp những cách tạo ra một tập các dữ liệu xuất có thể điều khiển được khi chương trình đang chạy là rất hữu ích; các dữ liệu xuất thêm vào có thể giúp ích trong suốt quá trình kiểm chứng.

Nên thực hiện kiểm chứng trên nhiều máy, nhiều trình biên dịch và thậm chí trên nhiều hệ điều hành. Mỗi sự kết hợp sẽ phát hiện một số lỗi mà chúng không được tìm thấy trên những sự kết hợp khác, chẳng hạn như, sự phụ thuộc vào thứ tự *byte*, kích thước của số nguyên, việc xử lý con trỏ *null*, cách xử lý các giá trị trả về và ký tự kết thúc chuỗi, việc các tập tin *header* và thư viện. Kiểm chứng trên nhiều máy có thể phát hiện những vấn đề liên quan đến việc kết hợp các thành phần của chương trình, ta sẽ trình bày vấn đề này trong Chương 8 nhằm phát hiện sự phụ thuộc vào môi trường phát triển.

6.7. Ai sẽ thực hiện kiểm chứng?

Việc kiểm chứng được thực hiện bởi người cài đặt hay một người có thể truy cập tới mã nguồn, thường được gọi là phương pháp kiểm chứng hộp trắng *(white box testing)*. (Thuật ngữ gần tương tự với phương pháp kiểm chứng hộp đen *(black box testing)*, người kiểm chứng không biết bằng cách nào các thành phần được cài đặt.) Kiểm chứng mã nguồn của bạn là rất quan trọng; đừng cho rằng các bộ phận kiểm chứng hoặc người dùng sẽ tìm nó cho bạn. Nhưng cần phải kiểm tra rằng bạn đang kiểm chứng cẩn thận như thế nào, do đó hãy bỏ qua mã nguồn và nghĩ về những trường hợp phức tạp, những trường hợp không dễ một chút nào cả. Trích dẫn Don Knuth mô tả cách ông ta tạo ra những cho bộ định dạng TEX , *"Tôi nghĩ đến những trường hợp có ý nghĩa và khó khăn nhất mà tôi có thể xoay xở, và viết*

những mã nguồn phức tạp mà tôi nghĩ ra, sau đó tôi thay đổi hoàn toàn và ghi nhớ những chỉ thị hầu như là rất kinh khủng". Lý do của việc kiểm chứng là để tìm ra những lỗi, chứ không phải tuyên bố chương trình đã làm việc. Vì thế cần kiểm chứng ở những phần phức tạp, và khi tìm ra được lỗi, thì đó là một sự chứng minh cho phương pháp của bạn.

Phương pháp kiểm chứng hộp đen nghĩa là người thực hiện kiểm chứng không hiểu hoặc không có quyền truy cập vào từng chi tiết của mã nguồn. Nó tìm ra nhiều loại lỗi khác nhau, bởi vì người thực hiện kiểm chứng có những giả thuyết khác nhau khi bắt đầu thực hiện kiểm chứng. Các điều kiện biên là một vị trí tốt để bắt đầu sử dụng phương pháp hộp đen, các dữ liệu nhập với giá trị hoặc số lượng lớn và không hợp lệ là những bộ thử tốt. Dĩ nhiên bạn cũng cần phải kiểm chứng các "*trường hợp lưng chừng*" hoặc dùng các quy ước của chương trình để xác định những chức năng cơ bản.

Người sử dụng thật sự là bước kế tiếp. Người sử dụng mới tìm thấy nhiều lỗi mới, bởi vì họ thực hiện chương trình theo những cách chưa được đoán trước. Việc thực hiện những kiểm chứng này quan trọng trước khi chương trình được đưa ra thị trường, đáng buồn là nhiều chương trình được tung ra thị trường mà chưa được kiểm chứng kỹ lưỡng. Phiên bản Beta có nhiều người sử dụng thật sự để kiểm chứng chương trình trước khi nó được hoàn thành, nhưng phiên bản Beta không nên được sử dụng để thay thế cho việc kiểm chứng kỹ càng. Tuy nhiên, khi hệ thống phần mềm trở nên lớn và phức tạp hơn, và thời gian phát triển ngắn lại thì áp lực để đưa ra một chương trình chưa được kiểm chứng đầy đủ lại tăng lên.

6.8. Kiểm chứng chương trình Markov

Chương trình *Markov* ở Chương 3 khá rắc rối nên nó cần được kiểm chứng một cách cẩn thận. Nó tạo ra các giá trị vô nghĩa nên khó phân tích tính hợp lệ, và ta đã viết nhiều phiên bản trên một số ngôn ngữ. Do sự phức tạp ở bước cuối cùng, việc phát sinh dữ liệu của chúng là ngẫu nhiên và mỗi lần lại khác nhau. Bằng cách nào ta có thể áp dụng những bài học trong

chương này để kiểm chứng chương trình *Markov*?

Kiểm chứng đầu tiên xét tập dữ liệu bao gồm một số tập tin nhỏ được dùng kiểm tra các điều kiện biên, để đảm bảo chương trình sẽ phát sinh ra kết quả đúng đối với những dữ liệu nhập chỉ chứa vài từ. Đối với các tiếp vĩ ngữ có chiều dài là hai, ta sử dụng năm tập tin lưu trữ riêng biệt (với một từ trên một hàng)

```
(empty file)

a

a b

a b c

a b c d
```

Đối với mỗi tập tin, dữ liệu xuất phải đồng nhất với giá trị dữ liệu nhập. Những kiểm chứng này phát hiện ra nhiều lỗi trong quá trình khởi tạo bảng, khởi động và kết thúc bộ phát sinh.

Kiểm chứng thứ hai nhằm xác minh các thuộc tính lưu trữ. Đối với các tiếp vĩ ngữ hai từ, mỗi từ, mỗi cặp từ và mỗi bộ ba từ xuất hiện trong kết quả được phát sinh ra phải xuất hiện trong dữ liệu nhập. Ta đã viết chương trình Awk đọc các giá trị dữ liệu nhập chuẩn vào một mảng lớn, tạo ra các mảng của những bộ đôi, bộ ba, sau đó đọc các kết quả được phát sinh của chương trình *Markov* vào một mảng khác và so sánh hai mảng này:

```
    # Kiểm chứng chương trình Markov
    # Thực hiện kiểm tra trên tất cả các từ, cặp, và bộ ba
từ trong
    # kết quả phát sinh ARGV[2] từ dữ liệu nhập ARGV[1]
    BEGIN {
        while (getline <ARGV[1] > 0)
            for (i = 1; i <= NF; i++) {
            wd[++nw] = $i      # nhập từ vào
```

```
                single[$i]++
            }
        for (i = 1; i < nw; i++)
                pair[wd[i],wd[i+1]]++
        for (i = 1; i < nw-1; i++)
                triple[wd[i],wd[i+1],wd[i+2]]++
        while (getline < ARGV[2]>0) {
                outwd[++ow] = $0        #phát sinh kết quả
                if (!($0 in single))
                    print "unexpected word", $0
        }
        for (i = 1; i < ow; i++)
                if (!((outwd[i],outwd[i+1]) in pair))
                    print "unexpected pair", outwd[i],
outwd[i+1]
        for (i = 1; i < ow-1; i++)
                if(!((outwd[i],outwd[i+1], outwd[i+2]) in
triple))
                    print"unexpected triple",
                        outwd[i], outwd[i+1],
outwd[i+2]
    }
```

Chúng ta không cần phải tạo ra một kiểm chứng hiệu quả mà chỉ cần tạo ra một chương trình kiểm chứng càng đơn giản càng tốt. Nó mất khoảng 6 hoặc 7 giây để kiểm tra tập tin dữ liệu được phát sinh khoảng 10000 từ tương ứng tập tin dữ liệu nhập 42685 từ, không lâu hơn thời gian của một số phiên bản của *Markov* dùng để phát sinh ra kết quả. Việc kiểm tra sự duy trì gặp một lỗi chính của bản cài đặt bằng Java: thỉnh thoảng chương trình ghi đè lên một số mục của bảng băm bởi vì nó sử dụng tham chiếu thay vì tạo ra

những bản sao các tiếp đầu ngữ.

Kiểm chứng này minh họa cho nguyên tắc xác định một tính chất của kết quả thì dễ hơn việc tạo ra nó. Chẳng hạn như, việc kiểm tra một tập tin có được sắp xếp hay không thì dễ hơn thực hiện sắp xếp tập tin.

Kiểm chứng thứ ba là việc thống kê trong tự nhiên. Dữ liệu nhập bao gồm chuỗi tuần tự

```
a b c a b c ... a b d ...
```

với 10 lần xuất hiện của `abc` rồi sau đó là `abd` và cứ tiếp tục như thế. Dữ liệu xuất có số lần xuất hiện của `c` gấp 10 lần của `d` nếu sự chọn lựa ngẫu nhiên được thực hiện tốt. Tất nhiên, ta khẳng định điều này nhờ hàm `freq`.

Kiểm chứng thống kê trình bày một phiên bản sơ lược của chương trình Java, gắn một bộ đếm vào mỗi tiếp vĩ ngữ, tạo ra 20 ký tự `c` cho mỗi ký tự `d`, gấp đôi số lượng cần có. Sau vài thảo luận sơ bộ, ta nhận thấy bộ phát sinh số ngẫu nhiên của Java tạo ra cả số nguyên âm lẫn số dương, thừa số của hai xuất hiện bởi vì phạm vi của các giá trị thì gấp đôi như những gì ta muốn, vì vậy sẽ có gấp đôi giá trị sẽ trả về là không khi thực hiện chia lấy dư của bộ phát sinh ngẫu nhiên; điều này đã xảy ra với phần tử đầu tiên trong danh sách, đó chính là `c`. Biện pháp sửa chữa là lấy giá trị tuyệt đối trước khi thực hiện phép chia lấy dư. Nếu không thực hiện kiểm chứng loại này thì sẽ không bao giờ phát hiện ra lỗi này, vì dữ liệu xuất có vẻ là tốt.

Cuối cùng, ta tìm được một văn bản tiếng Anh đơn giản để thấy rằng chương trình *Markov* đã phát sinh ra văn bản không có nghĩa. Dĩ nhiên, ta cũng thực hiện việc kiểm chứng này trong quá trình phát triển chương trình. Nhưng ta không chỉ dừng lại kiểm chứng khi chương trình xử lý những dữ liệu nhập thông thường, mà còn thực hiện nó với những trường hợp khó khăn sẽ xảy ra trong thực tế. Kiểm chứng các trường hợp đúng và dễ thì rất hấp dẫn, nhưng các trường hợp khó cũng cần phải được kiểm chứng. Việc kiểm chứng có hệ thống được tự động hóa là cách tốt nhất để tránh các lỗi này.

Tất cả các kiểm chứng đều được tự động hóa. Một giao diện *script*

phát sinh các dữ liệu nhập cần thiết, chạy và định giờ các kiểm chứng, và in ra những dữ liệu xuất bất thường. *Script* có thể được định dạng do đó các kiểm chứng giống nhau có thể được áp dụng trong bất kỳ phiên bản nào của *Markov*, và mỗi khi chương trình thay đổi thì ta chạy lại các kiểm chứng để chắc chắn rằng không có gì bị hư hỏng.

6.9. Tổng kết

Mã nguồn của bạn được viết càng tốt thì sẽ càng có ít lỗi và chắc chắn bạn càng thực hiện kiểm chứng được kỹ lưỡng hơn. Kiểm chứng điều kiện biên khi bạn viết chương trình là một cách hiệu quả để loại ra nhiều lỗi nhỏ ngớ ngẩn. Việc kiểm chứng có hệ thống thực hiện việc dò tìm tại những điểm rắc rối một cách tuần tự, hơn nữa, những lỗi thông thường được tìm thấy ở biên, có thể được dò tìm một cách thủ công hoặc bằng chương trình. Thật lý tưởng để tự động hóa quá trình kiểm chứng, bởi vì máy tính không gây ra nhiều lỗi, không mệt mỏi và cũng không gây ra ngộ nhận một chức năng nào đó đang thực hiện trong khi nó không thực hiện. Sự kiểm chứng lùi kiểm chứng phiên bản sau của chương trình vẫn cho ra những kết quả như được thực thi bởi phiên bản trước. Thực hiện kiểm chứng sau mỗi thay đổi nhỏ là một kỹ thuật tốt để cục bộ hóa mã nguồn của mọi vấn đề, bởi vì hầu hết những lỗi mới thường xuất hiện trong các mã nguồn mới.

Một quy tắc cơ bản của việc kiểm chứng là hãy thực hiện kiểm chứng.

Chương 7

TỐC ĐỘ THỰC THI

Từ khi máy tính mới ra đời, các nhà lập trình phải nỗ lực rất nhiều để viết chương trình sao cho thật sự hiệu quả vì máy tính thời đó vừa đắt tiền vừa có cấu hình yếu. Ngày nay, máy tính rẻ hơn và có cấu hình mạnh hơn nhiều, nên nhu cầu viết chương một cách hiệu quả tuyệt đối đã giảm xuống rõ rệt. Có đáng bận tâm về tốc độ thực thi chương trình hay không?

Vẫn còn, khi chương trình thực sự chạy quá chậm, ta vẫn có thể hy vọng làm cho chương trình chạy nhanh hơn trong khi vẫn giữ được tính đúng đắn, và tính sáng sủa của nó. Một chương trình chạy nhanh mà lại cho kết quả sai thì không có ý nghĩa gì cả..

Như vậy, nguyên tắc tối ưu hóa đầu tiên là đừng tối ưu hóa. Chương trình thực thi đã đủ tốt chưa? Trước hết, cần tìm hiểu cách thức sử dụng chương trình và môi trường mà nó thực thi để từ đó xác định xem việc tăng tốc độ của chương trình có lợi hay không. Các chương trình viết để ra bài tập trong trường học sẽ không bao giờ được dùng lại lần nữa: tốc độ rất ít khi là vấn đề. Đối với các chương trình ra đề bài tập cho học sinh thì tốc độ thực thi rất ít khi gặp khó khăn. Cũng tương tự đối với các chương trình cá nhân, các công cụ ít dùng, khung trắc nghiệm, thí nghiệm, và chương trình mẫu. Tuy nhiên, vì thời gian thực thi (run-time) của một sản phẩm thương mại hay của một thư viện đồ họa có thể vô cùng quan trọng nên cần phải quan tâm đến tốc độ thực thi của chúng, do vậy cần phải hiểu cách suy tính về vấn đề tốc độ thực thi.

Khi nào ta nên tăng tốc cho một chương trình? Chúng ta làm như thế bằng cách nào? Chúng ta mong muốn có được những gì? Chương này sẽ nói

về những cách làm cho chương trình chạy nhanh hơn hoặc dùng ít bộ nhớ hơn. Tốc độ thường là điều quan tâm lớn nhất nên chủ yếu sẽ nói về vấn đề này. Không gian (bộ nhớ, đĩa) ít khi là vấn đề gây trở ngại nhưng cũng có thể đóng vai trò quyết định nên ta cũng giành một phần để nói về nó.

Như đã thấy ở Chương 2, chiến lược tốt nhất là dùng những thuật toán đơn giản nhất, rõ ràng nhất cũng như những cấu trúc dữ liệu thích hợp. Sau đó đo lường tốt độ thực thi để xét xem liệu có cần thay đổi chỗ nào không; cho phép lựa chọn sinh mã nhanh nhất của trình biên dịch; đánh giá những thay đổi nào của chương trình hiệu quả nhất; mỗi lần chỉ đưa vào một thay đổi và đánh giá lại; giữ phiên bản đơn giản nhất để kiểm nghiệm.

Đo lường là một thành phần quan trọng trong việc cải thiện tốc độ thực thi chương trình vì suy luận và trực giác có thể là những định hướng sai lầm và cần phải được bổ sung bằng những công cụ đo thời gian thực thi và lập sơ đồ thời gian của chương trình. Việc cải thiện tốc độ thực thi chương trình có nhiều điểm chung với việc kiểm chứng, bao gồm những kỹ thuật như tự động hóa, lưu vết, và các kiểm chứng lùi để bảo đảm cho các thay đổi vẫn giữ được sự đúng đắn và không làm mất đi những cải tiến trước đó.

Nếu đã chọn tốt được thuật toán cũng như đã viết tốt ngay từ đầu thì có thể không cần tăng tốc thêm. Mã lệnh thiết kế tốt chỉ cần những thay đổi nhỏ thường gặp để sửa chữa các vấn đề về tốc độ thực thi. Trong khi đó, mã lệnh thiết kế kém sẽ cần phải viết lại nhiều hơn.

7.1. Hiện tượng nghẽn cổ chai

Chúng ta hãy bắt đầu bằng việc mô tả cách loại trừ hiện tượng nghẽn cổ chai ra khỏi một chương trình đang bị lỗi trong môi trường cục bộ.

Thư điện tử đến với chúng ta phải thông qua một máy tính gọi là gateway, có chức năng nối mạng nội bộ với Internet. Thư điện tử từ bên ngoài gửi đến gateway của một mạng nội bộ chỉ vài nghìn người có thể lên đến hàng chục nghìn bức một ngày, và sau đó được phát đến từng người trong mạng nội bộ; sự ngăn cách này cô lập mạng nội bộ của chúng ta với Internet và chỉ cho phép ta công bố một tên máy (là tên của gateway) cho tất

cả mọi người trong mạng nội bộ.

Một trong số các dịch vụ của gateway là lọc các *"spam"*, là các thư rác kèm theo các quảng cáo về những dịch vụ mà lợi ích chưa được kiểm chứng. Sau một số thử nghiệm thành công ban đầu bộ lọc các *"spam"* đã được cài đặt như một đặc tính không thể thiếu cho tất cả người dùng trong mạng nội bộ, và rồi rắc rối lập tức xuất hiện. Gateway, vốn cũ kỹ và thường rất bận, bị quá tải vì chương trình lọc chiếm quá nhiều thời gian – nhiều hơn hẳn so với thời gian cần thiết cho tất cả các thao tác xử lý khác đối với bức thư – đến nỗi hàng đợi thư bị đầy và việc truyền phát thư tín bị đình hoãn hàng giờ trong khi hệ thống cố xoay xở để kịp phân phối thư.

Sau đây là một ví dụ về vấn đề tốc độ thực thi chương trình: chương trình không đủ nhanh để hoàn thành nhiệm vụ, và sự chậm trễ này tạo ra trở ngại cho người dùng. Do đó chương trình cần phải chạy nhanh hơn nhiều so với trước.

Một cách đơn giản, bộ lọc *"spam"* hoạt động như sau. Mỗi thông điệp gửi tới được xem như một chuỗi đơn, và một bộ so mẫu ký tự kiểm tra chuỗi đó xem có chứa bất kỳ một cụm từ nào trong số các cụm từ thường gặp trong các "spam" hay không, chẳng hạn như: "Make millions in your spare time" (kiếm hàng triệu đồng trong thời gian rảnh của bạn) hay "XXX-rated" (các trang khiêu dâm). Các thông điệp có xu hướng lặp lại, vì vậy kỹ thuật này có hiệu quả đáng kể, và nếu như một thông điệp *"spam"* lọt lưới thì một cụm từ đặc trưng cho *"spam"* đó sẽ được thêm vào danh sách các *"spam"* để chặn nó vào lần sau.

Do không có một công cụ so sánh chuỗi nào, ví dụ như *grep*, vừa chạy nhanh vừa đáng tin cậy, nên người ta viết ra một bộ lọc đặc biệt dùng cho mục đích lọc các *"spam"*. Mã nguồn rất đơn giản; nó tìm xem mỗi bức thông điệp có chứa một trong số các cụm từ (các mẫu) nào đó hay không:

```
        /* Hàm isspam: kiểm tra chuỗi thông điệp
mesg có chứa spam nào không */
        int isspam(char *mesg)
```

```
        int i;
        for (i = 0; i < npat; i++)
            if (strstr(mesg, pat[i]) !=NULL){
                printf("spam: match for
'%s'\n", pat[i]);
                return 1;
            }
        return 0;
    }
```

Làm sao để thực hiện đoạn chương trình trên nhanh hơn? Cần phải tìm kiếm chuỗi ký tự mẫu trong bức thông điệp, dùng hàm `strstr` của thư viện C là cách tốt nhất để tìm kiếm vì nó vừa đúng chuẩn vừa hiệu quả.

Dùng kỹ thuật lập sơ đồ sử dụng thời gian, kỹ thuật này sẽ được nói kỹ hơn ở mục sau, ta thấy rõ rằng việc thực thi hàm `strstr` có những tính chất không thích hợp khi dùng cho một bộ lọc *spam*. Việc thay đổi cách thức làm việc của hàm `strstr` có thể tăng hiệu quả lên *trong trường hợp cụ thể này.* Hàm `strstr` được cài lại như sau:

```
    /* Hàm strstr: dùng hàm strchr để tìm kiếm kí tự
đầu tiên */
    char *strstr(const char *s1, const char *s2)
    {
        int n;
        n = strlen(s2);
        for (;;) {
            s1 = strchr(s1, s2[0]);
```

```
                if (s1 == NULL)
                    return NULL;
                if (strncmp(s1, s2, n) == 0)
                    return (char *) s1;
                s1++;
            }
        }
```

Đoạn chương trình này đã được viết hướng tới mục tiêu hiệu quả, và trên thực tế thì nó chạy nhanh nếu dùng cho một mục đích tiêu biểu nào đó, vì chỉ dùng các hàm được tối ưu hóa cao để thực hiện công việc. Đoạn chương trình này gọi hàm `strchr` để tìm vị trí tiếp theo của ký tự đầu tiên trong chuỗi, sau đó gọi để xem liệu phần còn lại của chuỗi có so khớp với phần còn lại của chuỗi mẫu. Như vậy, đoạn chương trình này sẽ bỏ qua một phần lớn của bức thông điệp để tìm chỉ ký tự đầu tiên của chuỗi mẫu, rồi quét nhanh để tìm kiếm phần còn lại. Nhưng tại sao tốc độ thực thi chương trình lại chậm?

Có vài lý do sau đây. Trước hết, hàm `strncmp` lấy chiều dài chuỗi mẫu bằng `strlen` làm tham số đầu vào. Nhưng vì chiều dài chuỗi mẫu là cố định nên không cần tính lại khi xử lý mỗi thông điệp.

Thứ hai, hàm `strncmp` có chứa một vòng lặp phức tạp. Thực tế không những phải so từng *byte* của hai chuỗi mà phải tìm *byte* cuối `'\0'` trên cả hai chuỗi trong khi vẫn đếm lùi tham số chiều dài. Vì chiều dài của tất cả các chuỗi đều đã biết trước (dù không cần dùng tới hàm `strncmp`) nên sự phức tạp này là không cần thiết; ta biết rằng phép đếm là đúng nên việc kiểm tra *byte* `'\0'` là phí phạm thời gian.

Thứ ba, hàm `strnchr` cũng phức tạp, bởi vì đồng thời tìm ký tự và kiểm tra *byte* `'\0'` kết thúc bức thông điệp. Trong một lần gọi hàm `isspam`

thì bức thông điệp là cố định, vì vậy thời gian dùng để kiểm tra *byte* '\0' là uổng phí vì ta đã biết bức thông điệp kết thúc ở đâu.

Cuối cùng, dù cho hàm strncmp, strchr, và strlen đều rất hiệu quả khi làm việc độc lập, tổng thời gian cần để gọi các hàm này cũng xấp xỉ với thời gian tính toán của chúng. Sẽ hiệu quả hơn nếu làm mọi việc bằng chỉ một hàm strstr được viết lại thật cẩn thận và tránh gọi tất cả những hàm khác.

Những loại vấn đề này là một nguồn gốc chung làm chậm tốc độ thực thi chương trình – một thủ tục hoặc một phương thức giao tiếp làm việc tốt trong một tình huống điển hình nhưng lại thực hiện kém trong tình huống bất thường xảy ra trở thành trung tâm của chương trình. Hàm strstr có sẵn thực hiện tốt khi chuỗi mẫu và chuỗi cần tìm ngắn và thay đổi ở mỗi lần hàm được gọi. Nhưng khi chuỗi cần tìm dài và cố định thì thời gian tiêu tốn quá nhiều.

Đã nắm vững điều nói trên, strstr được viết lại để xử lý trên chuỗi mẫu cùng với chuỗi thông điệp nhằm tìm ra chỗ so khớp mà không cần gọi thủ tục con. Khả năng thực thi thu được hoàn toàn có thể biết trước: có thể hơi chậm trong một số trường hợp, song rất nhanh trong việc lọc các "*spam*" và quan trọng nhất là không bao giờ chạy quá lâu. Để kiểm tra lại tính đúng đắn của bản cài đặt mới cũng như tốc độ thực thi của chương trình mới, ta xây dựng một bộ kiểm chứng tốc độ thực thi. Bộ thử này bao gồm không chỉ các ví dụ đơn giản như tìm một từ trong câu, mà còn gồm cả một số trường hợp đặc biệt như tìm chuỗi mẫu chỉ có một ký tự 'x' trong chuỗi có một ngàn ký tự 'e', và chuỗi mẫu có một ngàn ký tự 'x' trong một chuỗi chỉ có một ký tự 'e'. Những trường hợp đặc biệt như vậy là phần chính yếu trong việc đánh giá khả năng thực thi.

Thư viện được cập nhật hàm strstr mới và bộ lọc "*spam*" chạy nhanh hơn 30%, đây là kết quả đáng giá cho việc viết lại.

Tuy nhiên, tốc độ thực thi vẫn còn quá chậm. Khi giải quyết các khó khăn, điều quan trọng là xác định đúng vấn đề cần giải quyết. Cho đến nay,

ta đang đặt vấn đề tìm cách nhanh nhất để truy soát một chuỗi ký tự mẫu trong một chuỗi ký tự. Nhưng thực ra thì vấn đề là tìm kiếm một tập hợp lớn. cố định các chuỗi ký tự mẫu trong một chuỗi ký tự dài thay đổi. Trong trường hợp đó, hàm `strstr` không thực sự là giải pháp đúng.

Cách hiệu quả nhất để làm cho một chương trình chạy nhanh hơn là dùng một giải thuật tốt hơn. Bây giờ khi đã nhìn rõ hơn vấn đề cần giải quyết, ta cần suy nghĩ xem thuật toán nào sẽ làm việc tốt nhất.

Vòng lặp cơ bản:

```
for (i = 0; i < npat; i++)
if (strstr(mesg, pat[i]) != NULL)
     return 1;
```

duyệt qua bức thông điệp `npat` lần độc lập; giả thiết rằng nó không tìm ra bất kỳ sự so khớp nào, nó phân tích từng *byte* của bức thông điệp `npat` lần trong `strlen(mesg)*npat` phép so sánh.

Một cách tiếp cận tốt hơn là đảo ngược vòng lặp, quét qua bức thông điệp một lần ở vòng lặp ngoài trong khi tìm tất cả các chuỗi mẫu song song ở vòng lặp trong:

```
for (j = 0; mesg[j] != '\0'; j++)
   if(có chuỗi mẫu nào có phần đầu trùng với ký
tự đầu của mesg[j])
        return 1;
```

Sự cải thiện tốc độ bắt nguồn từ một cách nhìn đơn giản. Để nhận ra liệu có chuỗi mẫu nào so khớp với bức thông điệp ở vị trí `j` không, ta không cần phải xét tất cả các chuỗi mẫu mà chỉ cần xét các chuỗi có cùng ký tự đầu tiên với `mesg[j]`. Đại khái, với 52 ký tự chữ hoa và chữ thường ta chỉ cần thực hiện `strlen(mesg)*npat/52` phép so sánh. Vì các chữ cái không phân bố đều, có nhiều từ bắt đầu bằng ký tự `'s'` hơn là bằng ký tự `'x'`, ta sẽ không có được một sự cải thiện 52 lần tốt hơn, nhưng ắt là phải có ở một

mức độ nào đó. Để thực hiện, ta xây dựng một bảng băm dùng ký tự đầu tiên của chuỗi mẫu làm khóa.

Dựa vào một vài tính toán trước để xây dựng một bảng những chuỗi mẫu bắt đầu với từng ký tự, hàm `isspam` vẫn ngắn như sau:

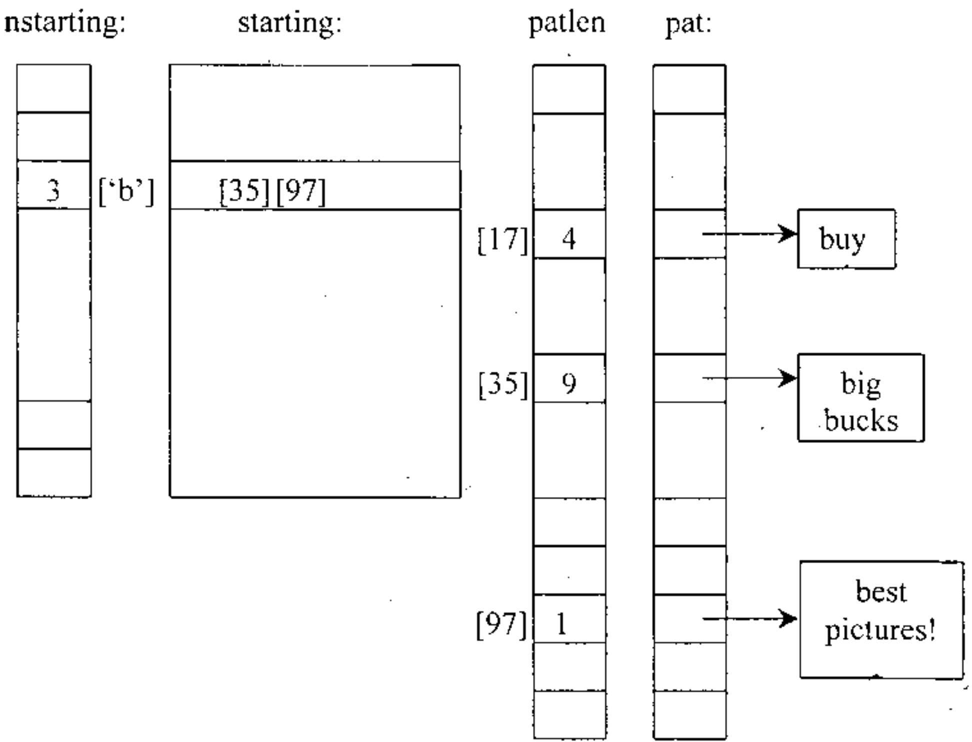
```
int patlen[NPAT];    /* mảng chiều dài của các
chuỗi mẫu */
int starting[UCHAR_MAX+1][NSTART];    /* danh mục
các chuỗi mẫu bắt bằng ký tự tương ứng */
int nstarting[UCHAR_MAX+1];     /* số chuỗi mẫu
bắt đầu bằng ký tự tương ứng */
      ...
/* Hàm isspam: kiểm tra chuỗi thông điệp mesg có
chứa spam nào không */
int isspam(char *mesg)
{
      int i, j, k;
      unsigned char c;

            for (j = 0; (c = mesg[j]) != '\0'; j++)
 {
                  for (i = 0; i < nstarting[c]; i++)
 {
                        k = staring[c][i];
                        if (memcmp(mesg+j, pat[k],
patlen[k]) == 0) {
                              printf("spam: match for
'%s'\n", pat[k]);
                              return 1;
```

```
                }
            }
            return 0;
    }
```

Với mỗi ký tự `'c'`, mảng hai chiều `starting[c][]` chứa danh mục các chuỗi mẫu bắt đầu bằng ký tự `'c'` đó. Mảng `nstarting[c]` ghi nhận bao nhiêu chuỗi mẫu bắt đầu bằng ký tự `'c'`. Nếu không có các bảng đó, vòng lặp trong sẽ chạy từ 0 đến `npat`, khoảng 1000, thay vì chỉ chạy từ 0 đến khoảng 20. Còn phần tử mảng `patlen[k]` chứa kết quả tính toán trước của `strlen(pat[k])`.

Hình dưới đây phác họa những cấu trúc dữ liệu này dùng một tập hợp ba chuỗi mẫu bắt đầu bằng ký tự `'b'`:



Đoạn chương trình để xây dựng các bảng trên rất dễ:

```
int i;
unsigned char c;

for (i = 0; i < npat; i++){
     c = pat[i][0];
     if (nstarting[c] >= NSTART)
          eprintf("too many patterns (>=%d)
begin '%c'",
                         NSTART, c);
     starting[c][nstarting[c]++] = i;
     patlen[i] = strlen(pat[i]);
}
```

Phụ thuộc vào dữ liệu nhập, bộ lọc *"spam"* bây giờ cải tiến chạy nhanh hơn từ 5 đến 10 lần khi dùng hàm `strstr`, và nhanh hơn từ 7 đến 15 lần khi dùng phương pháp đầu tiên. Ta không có được sự cải thiện đến 52 lần, một phần là do sự phân bố không đều của bảng chữ cái, một phần do vòng lặp trong chương trình mới phức tạp hơn, và một phần khác do vẫn còn phải thực hiện quá nhiều phép so chuỗi không đạt, nhưng bộ lọc *"spam"* không còn là nút cổ chai gây nghẽn việc phân phối thư nữa. Vấn đề tốc độ thực thi chương trình đã được giải quyết.

Phần còn lại của chương này sẽ đi sâu vào các kỹ thuật được dùng để phát hiện các vấn đề về tốc độ thực thi chương trình, tách các đoạn mã chậm và tăng tốc cho chúng. Trước khi tiếp tục, ta cần xem lại ví dụ về bộ lọc *"spam"* nhằm rút ra những bài học cần thiết. Nếu bộ lọc *"spam"* không phải là một nút cổ chai thì tất cả các cố gắng của ta đều không có ý nghĩa. Một khi ta biết nó gây ra vấn đề, ta đã dùng kỹ thuật *profiling* và những kỹ thuật khác để nghiên cứu cách thực thi của chương trình nhằm tìm ra vấn đề

thực sự nằm ở đâu. Sau đó ta phải chắc chắn rằng ta đã giải quyết đúng vấn đề, thử nghiệm toàn bộ chương trình chứ không chỉ tập trung vào hàm `strstr`, là mối nghi ngờ ban đầu, thoạt đầu tưởng là đã rõ nhưng thực ra lại là không đúng. Cuối cùng, ta giải quyết đúng vấn đề bằng một giải thuật tốt hơn, và kiểm nghiệm lại xem chương trình đã chạy nhanh hơn chưa. Một khi chương trình đã chạy đủ nhanh, ta ngưng lại. Tại sao phải suy nghĩ quá mức cần thiết?

Bài tập 7.1. Cần phải cải tiến tầm mức của một bảng sơ đồ các chuỗi mẫu bắt đầu bằng một ký tự. Hãy cài đặt một phiên bản `isspam` dùng hai ký tự làm chỉ mục. Có thể cải tiến được đến mức nào? Đây là những trường hợp đơn giản của một kiểu cấu trúc dữ liệu được gọi là *trie*. Đa số những kiểu cấu trúc dữ liệu như vậy, đều chấp nhận tốn không gian lưu trữ để thời gian chạy chương trình nhanh hơn.

7.2. Kỹ thuật lập sơ đồ sử dụng thời gian

Các cách đo thời gian tự động

Đa số hệ thống đều có lệnh để đo thời gian chạy một chương trình. Trên Unix, lệnh này là `time`:

```
%time slowprogram

real       7.0

user       6.2

sys        0.1

%
```

Đoạn chương trình trên chạy lệnh và trả lại 3 giá trị đều đo bằng giây: thời gian "thực", là thời gian đã trôi qua cho đến khi chương trình kết thúc; thời gian CPU của người dùng, là thời gian dùng để thực thi chương trình; và thời gian CPU của hệ thống, là thời gian chương trình giữ quyền điều khiển trong hệ thống. Nếu hệ thống bạn đang dùng có một lệnh tương tự, hãy dùng lệnh đó; giá trị đo được chứa đựng nhiều thông tin hơn, tin cậy hơn, và dễ theo dõi hơn là thời gian đo bằng một đồng hồ bấm giờ. Và hãy

ghi chú cẩn thận. Trong khi bạn làm việc trên chương trình, sửa đổi, đo đạc, bạn sẽ tích lũy được nhiều dữ liệu đến nỗi chỉ một hai ngày sau là sẽ gây nhầm lẫn. (Chẳng hạn như phiên bản nào của chương trình chạy nhanh hơn 20%?). Nhiều kỹ thuật đã nghiên cứu trong Chương 6 có thể được đưa vào để đo đạc và cải tiến tốc độ thực thi chương trình. Dùng máy chạy và đo bộ chương trình kiểm tra, và điều quan trọng nhất là dùng kỹ thuật kiểm tra hồi quy để bảo đảm các sửa đổi không làm gãy chương trình.

Nếu hệ thống của bạn không có lệnh `time`, hay nếu bạn đang đo thời gian của riêng một hàm, xây dựng một cấu trúc đo thời gian tương tự với một cấu trúc kiểm tra là không khó. Trong C và C++ đều có cung cấp một hàm chuẩn, hàm `clock`, trả ra giá trị thời gian CPU mà chương trình đã dùng cho đến thời điểm hiện tại. Hàm này có thể được gọi trước và sau một hàm để đo thời gian sử dụng CPU:

```
#include <time.h>
#include <stdio.h>
     ...
     clock_t  before;
     double  elapsed;

     before = clock();
     long_running_function();
     elapsed = clock() - before;
     printf("function used %.3f seconds\n",
            elapsed/CLOCKS_PER_SEC);
```

Số hạng đếm gộp `CLOCKS_PER_SEC` ghi lại kết quả của chương trình bấm giờ do hàm `clock` báo ra. Nếu hàm chỉ thực hiện trong một phần nhỏ của một giây, cho hàm chạy trong một vòng lặp, nhưng cần nhớ phải tính phần trước vòng lặp nếu phần này tốn thời gian đáng kể:

```
before = clock();
for (i = 0; i < 1000; i++)
    short_running_function();
elapsed = (clock()-before)/(double)i;
```

Trong Java, các hàm trong lớp `Date` cho sẵn thời gian đồng hồ biên, là một xấp xỉ của thời gian CPU:

```
Date before = new Date();
long_running_function();
Date after = new Date();
long elapsed = after.getTime() -
before.getTime();
```

Giá trị trả về của `getTime` là miligiây.

Dùng công cụ lập sơ đồ sử dụng thời gian

Bên cạnh một phương pháp đo thời gian đáng tin cậy, công cụ quan trọng nhất trong phép phân tích tốc độ thực thi chương trình là một hệ thống giúp thiết lập sơ đồ sử dụng thời gian. Nó xác định cách chương trình sử dụng thời gian vào từng thao tác trong chương trình. Một số sơ đồ sử dụng thời gian liệt kê ra số các hàm, số lần chúng được gọi, và phần thời gian đã dùng để thực thi từng hàm. Số khác lại đếm số lần mỗi phát biểu được thực thi. Những phát biểu nào được thực thi nhiều lần sẽ chiếm thời gian thi hành nhiều hơn, trong khi đó những phát biểu không bao giờ được thực thi chính là những đoạn mã hoặc vô dụng hoặc chưa được kiểm tra đúng đắn.

Kỹ thuật lập sơ đồ sử dụng thời gian là công cụ hiệu quả để tìm các đoạn mã chính trong chương trình, tức là những hàm hoặc những đoạn mã chiếm phần lớn thời gian tính toán. Tuy nhiên, sơ đồ sử dụng thời gian cần phải được diễn giải một cách thận trọng. Do sự tinh tế của trình biên dịch và sự phức tạp trong các hiệu ứng của bộ nhớ đệm và bộ nhớ chính cũng như sự kiện là việc lập sơ đồ sử dụng thời gian của chương trình sẽ làm ảnh hưởng đến tốc độ thực thi, các thống kê trong sơ đồ sử dụng thời gian chỉ là

xấp xỉ.

Kỹ thuật lập sơ đồ sử dụng thời gian thường được kích hoạt bằng một cờ của trình biên dịch hoặc chức năng đặc biệt. Chương trình được cho chạy, và sau đó một công cụ phân tích biểu diễn kết quả. Trên Unix, cờ thường là cờ `-p` và công cụ được gọi là `prof` như sau:

```
% cc -p spamtest.c -o spamtest
% spamtest
% prof spamtest
```

Bảng sau đây mô tả một sơ đồ sử dụng thời gian được tạo ra bởi một phiên bản đặc biệt của bộ lọc *spam* chúng tôi đã tạo ra để tìm hiểu cách thức hoạt động của nó. Chương trình dùng một bức thông điệp cố định và một tập hợp cố định gồm 217 cụm từ, tập hợp này so khớp với bức thông điệp 10000 lần. Chương trình chạy trên máy 250 MHz MIPS R1000 dùng bản gốc của hàm `strstr` trong đó gọi các hàm chuẩn khác. Kết quả xuất ra đã được biên tập và định dạng lại cho vừa với khổ giấy. Hãy chú ý cách thể hiện kích cỡ của dữ liệu đầu vào (217 cụm từ) và số lần chạy (10000) dưới dạng những phép kiểm tra không đổi trong cột "*số lần gọi*", cột này đếm số lần gọi mỗi hàm.

12234768552: Tổng số lệnh thực thi

13961810001: Tổng số chu kỳ tính toán

55.847: Tổng thời gian tính toán

1.141: Số chu kỳ trung bình/lệnh

Rõ ràng là hàm `strchr` và hàm `strncmp`, cả hai do hàm `strstr` gọi, đã chi phối hoàn toàn tốc độ thực thi. Chỉ dẫn của Knuth tỏ ra hoàn toàn đúng: chỉ một phần nhỏ của chương trình đã chiếm gần hết thời gian chạy chương trình. Khi một chương trình lần đầu tiên được xây dựng *profile*, thường có thể thấy hàm được gọi nhiều nhất chiếm đến 50% hoặc hơn, như ở trường hợp này, và giúp ta dễ dàng nhận ra nơi cần tập trung chú ý.

secs	%	cum%	chu kỳ	lệnh	số lần gọi	hàm
45.260	81.0%	81.0%	11314990000	9440110000	48350000	strchr
6.081	10.9%	91.9%	1520280000	1566460000	46180000	strncmp
2.592	4.6%	96.6%	648080000	85450000	2170000	strstr
1.825	3.3%	99.8%	456225559	344882213	2170435	strlen
0.088	0.2%	100.0%	21950000	28510000	10000	isspam
0.000	0.0%	100.0%	100025	100028	1	main
0.000	0.0%	100.0%	53677	70268	219	_memccpy
0.000	0.0%	100.0%	48888	46403	217	strcpy
0.000	0.0%	100.0%	17989	19894	219	fgets
0.000	0.0%	100.0%	16798	17547	230	_malloc
0.000	0.0%	100.0%	10305	10900	204	realfree
0.000	0.0%	100.0%	6293	7161	217	estrdup
0.000	0.0%	100.0%	6032	8575	231	cleanfree
0.000	0.0%	100.0%	5932	5729	1	readpat
0.000	0.0%	100.0%	5899	6339	219	getline
0.000	0.0%	100.0%	5500	5720	220	_malloc

Tập trung quan tâm đến các đoạn mã nguồn chính

Sau khi viết lại hàm `strstr`, ta lập sơ đồ sử dụng thời gian của chương trình `spamtest` lần nữa và thấy rằng 99.8% thời gian bây giờ chiếm chỉ bởi `strstr`, mặc dù cả chương trình đã nhanh hơn rõ rệt. Khi có chỉ một hàm gây ra hiện tượng nghẽn cổ chai áp đảo, thì chỉ có hai con đường để lựa chọn: cải tiến hàm bằng một thuật toán tốt hơn, hoặc bỏ toàn bộ hàm đó bằng cách viết lại cả chương trình.

Trong trường hợp này, ta viết lại toàn bộ chương trình. Ở đây là một vài dòng đầu tiên trong sơ đồ sử dụng thời gian của chương trình `spamtest` dùng bản cài đặt có tốc độ cao của hàm `isspam`. Hãy chú ý rằng thời gian toàn bộ ít hơn nhiều, trong đó hàm `memcmp` là *hot spot*, và rằng hàm `isspam` bây giờ dùng một phần đáng kể trong thời gian tính toán. Bản này phức tạp hơn so với bản có gọi hàm `strstr`, nhưng ít tốn chi phí hơn nhờ vào sự loại bỏ hàm `strlen` và hàm `strchr` trong `isspam` và việc thay thế hàm `strncmp` bằng hàm `memcmp`, ít tốn *byte* hơn.

secs	%	cum%	chu kỳ	lệnh	số lần gọi	Hàm
3.524	56.9%	56.9%	880890000	1027590000	46180000	memcpy
2.662	43.0%	100.0%	665550000	902920000	10000	Isspam
0.001	0.0%	100.0%	140304	106043	652	Strlen
0.000	0.0%	100.0%	100025	100028	1	Main

Hãy so sánh số chu kỳ đếm được và số lần gọi trong hai sơ đồ sử dụng thời gian. Lưu ý rằng hàm `strlen` đi từ vài triệu lần gọi đến còn 652 và `strncmp` cùng với `memcmp` có cùng số lần gọi. Cũng lưu ý rằng hàm `isspam`, bây giờ hợp nhất với hàm `strchr`, vẫn dùng số chu kỳ ít hơn nhiều so với hàm `strchr` trước đó vì hàm này chỉ kiểm tra các chuỗi mẫu thích hợp ở mỗi bước. Các bước thực thi chi tiết có thể hiểu rõ thông qua phân tích định lượng.

Một đoạn mã nguồn chính thường có thể được loại trừ, hoặc ít nhất cũng làm yếu đi bằng một kỹ thuật đơn giản hơn so với cách ta đã dùng cho bộ lọc *spam*. Trước đây, một sơ đồ sử dụng thời gian của Awk cho thấy rằng một hàm được gọi khoảng một triệu lần qua diễn biến của một phép thử hồi quy trong vòng lặp sau:

```
?       for(j = i; j < MAXFLD; j++)
?           clear(j);
```

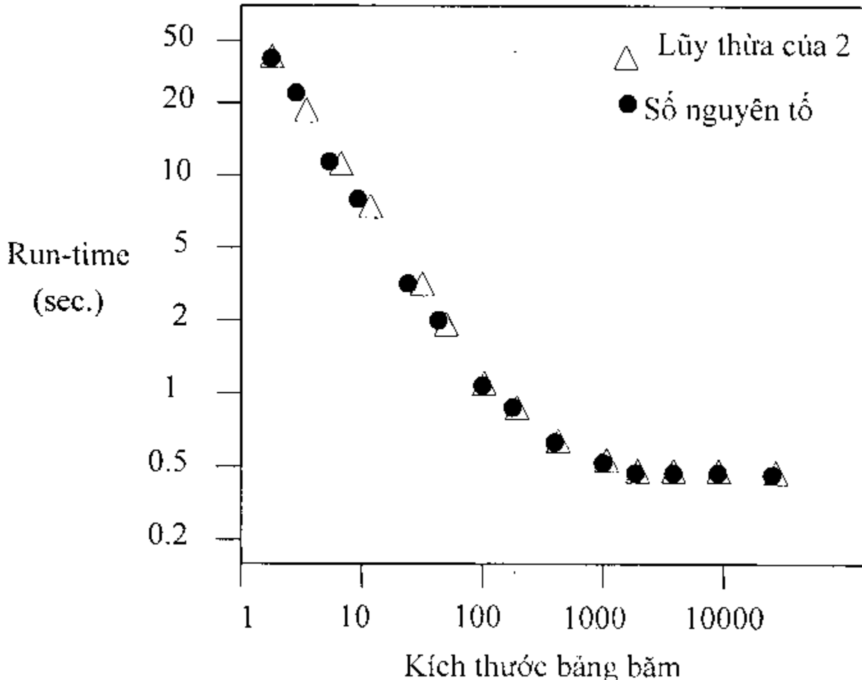
Vòng lặp này, thực hiện việc xóa các trường trước khi đọc vào một dòng mới, đã dùng hơn 50% thời gian thi hành. Hằng số `MAXFLD`, số trường tối đa cho phép trong một dòng, bằng 200. Nhưng trong hầu hết các ứng dụng của Awk, số trường thực sự chỉ là hai hay ba. Do vậy một khoảng thời gian rất dài đã bị sử dụng lãng phí vào việc xóa đi các trường không bao giờ được xác lập. Thay hằng số trên bằng giá trị trước đó của số trường tối đa cho phép tăng tốc toàn bộ lên chừng 25%. Sự sửa chữa được thực hiện bằng cách thay đổi cận trên của vòng lặp:

```
for(j = i; j < maxfld; j++)
    clear(j);
maxfld = i;
```

Vẽ hình

Hình vẽ thường rất hữu ích trong việc trình bày sự đo lường tốc độ thực thi chương trình. Các hình vẽ có khả năng chuyển tải thông tin về hiệu quả của việc thay đổi các tham số, so sánh các thuật toán và các cấu trúc dữ liệu, và đôi khi còn chỉ ra được những chỗ chương trình thực thi theo cách không mong muốn. Các đồ thị chiều dài dây xích rất có tác dụng trong một hàm băm số nhân ở Chương 5 đã cho thấy rất rõ rằng một số nhân tốt hơn số còn lại.

Đồ thị sau đây cho thấy ảnh hưởng do kích cỡ của mảng các bảng băm lên thời gian thi hành chương trình của giải thuật *Markov* viết bằng C dùng quyển sách 42685 từ, 22482 tiếp đầu ngữ làm dữ liệu nhập. Ta hãy làm hai thử nghiệm. Một thử nghiệm gồm một số các lần chạy chương trình dùng các cỡ mảng là lũy thừa của 2 từ 2 đến 16384; thử nghiệm kia dùng các cỡ mảng là số nguyên tố lớn nhất nhỏ hơn mỗi lũy thừa của 2. Ta xem liệu kích thước của mảng là số nguyên tố có làm nên sự khác biệt nào trong tốc độ thực thi chương trình hay không.



Đồ thị cho thấy thời gian chạy ứng với đầu vào trên đây không thay đổi nhiều so với kích thước bảng khi bảng có trên 1000 phần tử, cũng không có sự khác biệt rõ rệt nào giữa hai loại kích thước bảng theo số nguyên tố và theo lũy thừa của 2.

Bài tập 7.2. Cho dù hệ thống bạn đang dùng có lệnh `time` hay không, hãy dùng hàm `clock` hay hàm `getTime` viết một tiện ích đo thời gian cho mục đích dùng riêng của bạn. Hãy so sánh sự đo đạc thời gian ở tiện ích của bạn với một đồng hồ treo tường. Những hoạt động khác trong máy ảnh hưởng đến việc đo thời gian như thế nào?

Bài tập 7.3. Ở sơ đồ sử dụng thời gian đầu tiên, hàm `strchr` được gọi 48350000 lần và hàm `strncmp` chỉ gọi 46180000 lần. Hãy giải thích sự khác biệt.

7.3. Chiến lược tăng tốc độ

Trước khi thay đổi một chương trình để nó chạy nhanh hơn, hãy chắc rằng chương trình chạy thực sự quá chậm, và hãy dùng các công cụ đo thời gian và công cụ lập sơ đồ sử dụng thời gian để tìm xem thời gian đã dùng vào những việc gì. Một khi đã biết sự thể ra sao, ta có nhiều chiến lược để thực hiện mục đích. Ta liệt kê ra dưới đây một vài chiến lược theo thứ tự hiệu quả giảm dần.

Dùng thuật toán hoặc cấu trúc dữ liệu tốt hơn

Yếu tố quan trọng nhất làm cho chương trình chạy nhanh hơn là sự lựa chọn thuật toán và cấu trúc dữ liệu; có thể có một khác biệt rất lớn giữa một thuật toán hiệu quả và một thuật toán không hiệu quả. Bộ lọc *spam* của ta như đã thấy, một sự thay đổi cấu trúc dữ liệu đã đem lại một hệ số tốc độ 10 lần, và còn có thể có một sự cải thiện lớn hơn nếu như thuật toán mới giảm bớt số lệnh tính toán từ $O(n^2)$ còn $O(n\log n)$. Ta đã xem xét điều này ở Chương 2 nên không nhắc lại ở đây.

Hãy hiểu rằng điều mà ta chắc chắn sẽ gặp là sự phức tạp; nếu không, có thể có một lỗi ẩn đâu đó. Thuật toán tựa tuyến tính dùng để duyệt

chuỗi sau đây:

```
?      for (i = 0; i < strlen(s); i++)
?          if (s[i] == c)
?              ...
```

thực tế là bậc hai, nếu s có *n* ký tự, mỗi lệnh gọi hàm strlen sẽ đi qua *n* ký tự của chuỗi và vòng lặp thì được thực hiện *n* lần.

Làm hữu hiệu mọi khả năng tối ưu hóa của trình biên dịch

Một sự thay đổi không có phí tổn nào mà thường đem lại một sự cải thiện đáng kể là cho phép mọi khả năng tối ưu hóa mà trình biên dịch có. Các trình biên dịch hiện đại đều làm việc rất hiệu quả và cho phép loại bỏ nhiều những yêu cầu sửa đổi không lớn đối với lập trình viên.

Theo ngầm định thì hầu hết các trình biên dịch C và C++ không đưa vào nhiều khả năng tối ưu hóa cho chương trình. Một tùy chọn của trình biên dịch cho phép làm hữu hiệu bộ tối ưu hóa. Điều này nên là tùy chọn ngầm định trừ khi khả năng tối ưu hóa có xu hướng gây xáo trộn trình bắt lỗi ở mức độ nguồn, do vậy lập trình viên dứt khoát phải kích hoạt bộ tối ưu hóa một khi tin là chương trình đã được bắt lỗi.

Sự tối ưu hóa trình biên dịch thường cải thiện thời gian chạy chương trình từ vài phần trăm cho đến hai trăm phần trăm. Nhưng đôi khi lại có thể làm giảm tốc độ của chương trình, do vậy cần đo lường sự cải thiện trước khi đóng gói sản phẩm. Đối với bộ lọc *spam*, trong dãy các thử nghiệm dùng phiên bản thuật toán so trùng cuối cùng, thời gian thi hành ban đầu là 8.1 giây, giảm xuống còn 5.9 giây khi làm hữu hiệu khả năng tối ưu hóa, mức độ cải thiện như vậy là 25%. Ngược lại, ở phiên bản dùng hàm strstr đã sửa chữa, sự tối ưu hóa không đem lại một cải thiện nào bởi lẽ hàm strstr đã được tối ưu hóa khi đưa vào thư viện: bộ tối ưu hóa chỉ áp dụng vào mã nguồn đang được biên dịch trong chương trình chứ không phải vào các thư viện của hệ thống. Tuy nhiên, một số trình biên dịch có các bộ tối ưu hóa

toàn cục có khả năng phân tích toàn bộ chương trình tìm các chỗ còn có thể cải thiện được. Nếu một trình biên dịch như thế có trong hệ thống của bạn, hãy thử dùng; nó có thể giúp giảm được một vài chu kỳ.

Một điều cần phải hiểu rõ là càng cho phép trình biên dịch tối ưu hóa nhiều bao nhiêu thì khả năng đưa vào lỗi trong chương trình càng nhiều bấy nhiêu. Sau khi làm hữu hiệu khả năng tối ưu hóa, chạy lại dãy thử nghiệm hồi quy để có thể kịp thời thêm những sửa đổi cần thiết.

Tinh chỉnh mã

Chọn lựa thuật toán đúng đắn là điều rất quan trọng khi kích thước dữ liệu lớn. Hơn nữa, sự cải thiện do thuật toán có tác dụng ở mọi máy, mọi trình biên dịch, mọi ngôn ngữ lập trình. Nhưng trong trường hợp đã có thuật toán đúng mà tốc độ vẫn còn là vấn đề thì tiếp theo nên thử tinh chỉnh mã: điều chỉnh chi tiết các vòng lặp và phát biểu sao cho tốc độ thực thi nhanh hơn.

Phiên bản `isspam` ta đã xét ở phần cuối của mục 7.1 chưa được tinh chỉnh. Ở đây ta sẽ làm rõ những cải thiện nào có thể thu được bằng cách ngắt vòng lặp. Để nhắc lại, dưới đây là đoạn mã ban đầu:

```
for (j = 0; (c = mesg[j]) != '\0'; j++) {
     for (i = 0; i < nstarting[c]; i++) {
          k = staring[c][i];
          if (memcmp(mesg+j, pat[k], patlen[k])
== 0) {
                printf("spam: match for
'%s'\n", pat[k]);
                return 1;
          }
     }
}
```

Phiên bản ban đầu này trong chuỗi thử nghiệm chạy trong 6.6 giây khi được biên dịch có dùng bộ tối ưu hóa. Vòng lặp trong có một chỉ mục mảng (nstarting[c]) trong điều kiện lặp, mà chỉ mục này có giá trị không đổi ở mỗi lần lặp của vòng lặp ngoài. Ta có thể tránh tính lại giá trị đó bằng cách lưu vào một biến cục bộ:

```
for (j = 0; (c = mesg[j]) != '\0'; j++) {
    n = nstarting[c];
    for (i = 0; i < n; i++) {
        k = staring[c][i];
        ...
```

Cách làm này giảm thời gian thực thi chương trình xuống còn 5.9 giây, nhanh hơn được 10%, một sự tăng tốc điển hình cho khả năng cải thiện tốc độ mà của việc tinh chỉnh mã. Còn có thể đưa ra một biến khác: starting[c] cũng cố định. Có vẻ như loại bỏ phép tính khỏi vòng lặp như đã làm ở trên sẽ có ích, nhưng thực tế khi thử nghiệm ta không thấy một khác biệt rõ rệt nào. Điều này lại cũng là một kết quả điển hình của việc tinh chỉnh mã: đôi khi có ích, đôi khi lại không, và người ta phải tự đo đạc để tìm xem sự tinh chỉnh nào là có ích. Kết quả thu được có thay đổi theo từng máy cũng như từng trình biên dịch khác nhau.

Còn có một thay đổi khác ta có thể đưa vào cho bộ lọc *spam*. Vòng lặp trong so toàn bộ chuỗi mẫu với chuỗi cần xét, nhưng thuật toán bảo đảm là ký tự đầu tiên đã so trùng. Do vậy ta có thể tinh chỉnh mã để bắt đầu hàm memcmp ở *byte* tiếp theo. Thử nghiệm cho thấy sự tinh chỉnh này giúp tăng tốc độ 3%, tuy ít nhưng chỉ phải điều chỉnh 3 dòng của chương trình, mà một đã ở trước đoạn mã tính toán.

Không tối ưu hóa những gì không gây vấn đề

Đôi khi sự tinh chỉnh mã không đem lại kết quả gì bởi vì nó được sử dụng vào nơi không thích hợp. Hãy chắc chắn lại rằng đoạn mã đang được tối ưu hóa thực sự là chỗ ngốn nhiều thời gian chạy trong chương trình. Câu chuyện sau đây có thể là ngụy tạo, nhưng cũng nên kể ra làm ví dụ. Một

công ty nay không còn tồn tại, dùng một công cụ theo dõi tốc độ thực thi của phần cứng ở một chiếc máy đời đầu và thấy rằng máy dùng đến 50% thời gian thực thi cùng một dãy vài chỉ thị. Các kỹ sư đã tạo một chỉ thị để đóng gói lại thành một hàm, chạy lại, và thấy rằng họ không thu được kết quả gì; họ đã tối ưu hóa vòng lặp nghỉ của hệ thống.

Cần nỗ lực đến mức nào trong việc tăng tốc chương trình? Điều kiện chủ yếu là liệu các thay đổi có sinh lợi đến mức đáng giá hay không? Có một chỉ dẫn, đó là tổng thời gian đã dùng trong việc tăng tốc cho chương trình không được vượt quá thời gian mà sự tăng tốc có thể đem lại được suốt thời gian tồn tại của chương trình. Theo quy tắc này, sự cải tiến thuật toán đối với hàm `isspam` là đáng giá: mất một ngày làm việc nhưng lại làm lợi được hàng giờ mỗi ngày. Bỏ chỉ số mảng khỏi vòng lặp trong có vẻ kém nghệ thuật hơn, nhưng vẫn có giá trị, vì chương trình cung cấp dịch vụ cho một cộng đồng lớn. Tối ưu hóa các dịch vụ công cộng như bộ lọc *spam* hay một thư viện hầu như lúc nào cũng có giá trị, còn tăng tốc cho các chương trình trắc nghiệm hầu như không bao giờ có giá trị. Đối với một chương trình chạy suốt năm, chắt lọc tất cả những gì có thể được. Có thể khởi động lại sau một tháng chạy chương trình nếu tìm được cách tăng tốc cho chương trình đến 10%.

Các chương trình có tính cạnh tranh như trò chơi, trình biên dịch, xử lý văn bản, bảng tính, hệ thống cơ sở dữ liệu cũng đều thuộc nhóm này, vì sự thành công thương mại thường đến với phần mềm chạy nhanh nhất, ít nhất trong các kết quả chấm điểm chính thức.

Việc đo thời gian của chương trình khi thực hiện các thay đổi là rất quan trọng để chắc chắn rằng chương trình vẫn đang được cải thiện. Đôi khi hai cải tiến khác nhau có thể có ích một cách riêng rẽ nhưng tác động cùng lúc của cả hai lại loại trừ lẫn nhau. Cũng có trường hợp cơ chế đo thời gian hoạt động thất thường đến mức khó mà rút ra được kết luận chính xác về tác dụng của các thay đổi. Ngay cả trên một hệ thống chỉ có một người dùng, thời gian cũng có thể dao động ngoài dự tính. Nếu sự biến thiên của đồng hồ thời gian nội bộ (hay ít nhất là những gì được báo lại) đến 10%, những thay

đổi đem lại sự cải thiện 10% rất khó phân biệt với độ nhiễu đó.

7.4. Tinh chỉnh mã

Có nhiều kỹ thuật làm giảm thời gian chạy khi tìm thấy một đoạn mã chiếm phần lớn thời gian của chương trình. Sau đây là một số đề xuất, nên được áp dụng rất cẩn thận, và cần có các thử nghiệm hồi quy để bảo đảm chương trình vẫn chạy tốt. Cần nhớ rằng một số trình biên dịch tốt có thể làm giúp bạn một số phần, và thực tế là bạn có thể vô tình ngăn cản điều đó bằng cách làm phức tạp chương trình. Bất cứ những gì bạn thử, hãy đo lường hiệu quả để bảo đảm rằng điều đó có ích.

Tập hợp những biểu thức chung

Nếu một phép tính tốn nhiều thời gian xuất hiện nhiều lần, hãy thực hiện chỉ một lần rồi lưu lại kết quả. Ví dụ như ở Chương 1, ta đã gặp một macro dùng để tính khoảng cách bằng cách gọi hàm `sqrt` hai lần trong một dòng lệnh với cùng các giá trị, phép tính là như sau:

```
?     sqrt(dx*dx + dy*dy) + ((sqrt(dx*dx + dy*dy) > 0)?
...)
```

Hãy tính căn chỉ một lần và dùng giá trị đó ở hai nơi.

Nếu một phép tính thực hiện bên trong vòng lặp nhưng không phụ thuộc vào những thay đổi bên trong vòng lặp, hãy đưa ra ngoài, như thay đoạn chương trình sau:

```
for (i = 0; i < nstarting[c]; i++){
```

bằng đoạn chương trình:

```
n = nstarting[c];
for (i = 0; i < n; i++){
```

Thay các phép tính chạy chậm bằng các phép tính chạy nhanh hơn

Dùng các phép tính tối ưu hóa ít tốn thời gian thay thế các phép tính tốn thời gian. Ngày trước, kỹ thuật này được dùng để chỉ sự thay phép nhân

bằng phép cộng hoặc phép dịch bit, nhưng nay thì không cần nữa. Phép chia và phép lấy phần dư còn chậm hơn nhiều so với phép nhân, tuy nhiên, có thể cải thiện được khi thay phép chia bằng phép nhân với số nghịch đảo, hoặc phép lấy phần dư bằng một toán tử mặt nạ nếu số chia là lũy thừa của 2. Thay chỉ số mảng bằng con trỏ trong C hay C++ có thể tăng tốc được, dù hầu hết các trình biên dịch tự động làm việc này. Thay một phép gọi hàm bằng một phép tính đơn giản hơn cũng vẫn có thể có giá trị. Khoảng cách trong mặt phẳng được xác định theo công thức `sqrt(dx*dx+dy*dy)`, như vậy để xét xem điểm nào ở xa hơn trong hai điểm thường cần hai phép lấy căn bậc hai. Tuy nhiên ta có thể chỉ cần xét bình phương của khoảng cách:

```
if (dx1*dx1+dy1*dy1 < dx2*dx2+dy2*dy2)
    ...
```

cũng cho cùng kết quả như khi so sánh các căn bậc hai.

Ví dụ tương tự xảy ra trong các bộ so mẫu văn bản như ở bộ lọc *spam*. Nếu chuỗi mẫu bắt đầu với một chữ cái trong bảng chữ cái, một phép tìm kiếm được thực hiện đến cuối đoạn văn bản đầu vào đối với chữ cái đó; nếu không có sự trùng lặp nào, một cơ chế tìm kiếm tốn nhiều thời gian hơn sẽ không được gọi.

Khử hay loại bỏ các vòng lặp

Có những phí tổn thời gian nhất định trong xác lập và chạy một vòng lặp. Nếu thân vòng lặp không quá dài và không có nhiều bước lặp, có thể sẽ có hiệu quả hơn khi viết rõ từng bước lặp thành một dãy lệnh. Ví dụ như chuyển:

```
for (i = 0; i < 3; i++)
    a[i] = b[i] + c[i];
```

thành

```
a[0] = b[0] + c[0];
a[1] = b[1] + c[1];
a[2] = b[2] + c[2];
```

Điều này giúp loại bỏ phí tổn thời gian cho vòng lặp, đặc biệt trong trường hợp rẽ nhánh mà thường làm chậm các bộ xử lý hiện đại bằng cách ngắt dòng thực thi chương trình.

Nếu vòng lặp dài hơn, có thể dùng cách hoán chuyển tương tự để giảm bớt số lần lặp, chuyển đoạn chương trình sau:

```
for (i = 0; i < 3*n; i++)
    a[i] = b[i] + c[i];
```

thành

```
for (i = 0; i < 3*n; i += 3) {
    a[i+0] = b[i+0] + c[i+0];
    a[i+1] = b[i+1] + c[i+1];
    a[i+2] = b[i+2] + c[i+2];
}
```

Chú ý rằng điều này chỉ có kết quả khi chiều dài của mảng là bội số của bước lặp; nếu không thì lại cần thêm một đoạn mã để sửa đoạn cuối vòng lặp, đó là chỗ để các sai sót lọt vào và các hiệu quả đã thu được trôi mất lần nữa.

Lưu trữ lại các giá trị thường dùng

Các giá trị lưu trữ không cần phải tính lại. Việc lưu trữ có lợi thế ở tính nội bộ, là xu hướng chương trình dùng lại các mục mới truy cập (hoặc ở gần) có liên quan tới các dữ liệu cũ hơn (hoặc ở xa hơn).

Điều tốt nhất nên làm là làm sao để hoạt động lưu trữ không thể thấy được từ bên ngoài, như vậy sẽ không gây ảnh hưởng đến phần chương trình còn lại trừ tác động tăng tốc cho chương trình. Nghĩa là trong trường hợp chương trình xem trước trang in trên, phần chung của hàm vẽ ký tự

không đổi, luôn luôn là

```
drawchar(c);
```

Phiên bản gốc của `drawchar` gọi hàm `show(lookup(c))`. Việc thực thi phép lưu trữ dùng các giá trị biến nội bộ tĩnh để lưu ký tự trước, mã như sau:

```
if (c != lastc) { /*cập nhật cache */
     lastc = c;
     lastcode = lookup(c);
}
show(lastcode);
```

Viết một cấp phát đặc biệt

Một đoạn mã đơn lẻ chiếm phần lớn thời gian trong một chương trình thường là một cấp phát bộ nhớ, trong đó gặp rất nhiều phép gọi các hàm `malloc` hay dùng toán tử `new`. Khi hầu hết yêu cầu đều đòi hỏi các ô nhớ có cùng kích thước, có thể thu được một sự tăng tốc đáng kể nhờ thay các lệnh gọi đến bộ cấp phát tổng quát bằng lệnh gọi đến một cấp phát đặc biệt. Bộ cấp phát đặc biệt này thực hiện một lệnh gọi hàm `malloc` để xin cấp phát một mảng lớn các phần tử, rồi cấp phát cho mỗi phần tử khi cần, làm theo cách này ít tốn thời gian hơn. Sau khi các phần tử dùng xong vùng nhớ, vùng nhớ sẽ được giải phóng và được đặt trong *danh sách vùng nhớ tự do* để có thể được dùng lại nhanh chóng.

Nếu các phần tử khác nhau có cùng cỡ, ta có thể đánh đổi không gian lưu trữ lấy thời gian bằng cách luôn luôn cấp phát đủ cho cỡ lớn nhất được yêu cầu. Điều này khá hiệu quả trong việc quản lý các chuỗi ký tự ngắn nếu dùng cùng độ dài cho mọi chuỗi ở một giá trị xác định.

Một số thuật toán có thể dùng cách cấp phát dựa trên *stack*, trong đó toàn bộ trình tự cấp phát được thực hiện cùng một lúc và sau khi dùng xong thì tất cả chúng sẽ được giải phóng cùng một lần. Bộ cấp phát chiếm giữ một vùng nhớ cho mình và quản lý ở dạng *stack*, đưa vào (*pushing*) ra các

phần tử đã được cấp phát khi cần và cuối cùng lấy ra (*poping*) toàn bộ chỉ bằng một thao tác. Một số thư viện C có hàm `alloca` cho loại cấp phát này, cho dù đó không phải là cách làm đúng tiêu chuẩn. Hàm này gọi *stack* cục bộ và xem *stack* như là tài nguyên của bộ nhớ, và giải phóng toàn bộ các phần tử khi nó gọi trả về hàm `alloca`.

Dùng bộ nhớ trung gian cho việc nhập và xuất dữ liệu

Dùng bộ nhớ trung gian (hay bộ nhớ đệm) cho các khối lệnh để các hoạt động thường xuyên được thực thi với phí tổn ít nhất có thể, và các thao tác chiếm nhiều thời gian chỉ được thực hiện khi cần thiết. Chi phí cho các thao tác do vậy sẽ trải ra trên nhiều giá trị dữ liệu. Ví dụ như khi một chương trình C gọi hàm `printf`, các ký tự sẽ được lưu trong một bộ nhớ đệm chứ không được chuyển cho hệ điều hành cho đến khi bộ nhớ đệm đầy. Hệ điều hành đến lượt nó sẽ lại tiếp tục hoãn việc ghi dữ liệu vào đĩa. Điều trở ngại là việc làm đầy bộ nhớ đệm xuất để giúp thấy được dữ liệu; trong trường hợp xấu nhất, thông tin vẫn nằm trong bộ nhớ đệm xuất sẽ bị mất khi chương trình bị treo.

Quản lý một cách riêng các tình huống đặc biệt

Bằng cách quản lý các đối tượng có cùng kích thước bằng các đoạn mã riêng, các bộ cấp phát đặc biệt làm giảm phí tổn thời gian và không gian trong bộ cấp phát tổng quát và tình cờ làm giảm sự phân mảnh.

Tính trước kết quả

Một đôi khi có thể làm cho chương trình chạy nhanh hơn bằng cách tính trước một số kết quả để sẵn dùng khi cần. Ta đã thấy điều này trong bộ lọc *spam*, ở đó đã tính trước giá trị `strlen(pat[i])` và lưu lại trong mảng `patlen[i]`. Nếu một hệ thống đồ họa cần tính nhiều lần một hàm toán học như hàm `sin` chẳng hạn nhưng chỉ với một số tập hợp giá trị rời rạc, ví dụ như với số đo độ bằng số nguyên, chương trình sẽ chạy nhanh hơn nếu tính trước một bảng gồm 360 phần tử (hoặc cung cấp nó như nguồn dữ liệu) và tham chiếu vào bảng đó khi cần. Đây là ví dụ cho sự đánh đổi không gian lấy thời gian. Có nhiều dịp để thay mã lệnh bằng dữ liệu hay bằng cách thực

hiện tính toán trong quá trình biên dịch để tiết kiệm thời gian hay cả không gian nữa. Ví dụ, các hàm `ctype` cũng như hàm `isdigit` hầu như luôn luôn được cài đặt bằng cách tham chiếu vào bảng cờ `bit` hơn là việc thực hiện một chuỗi kiểm tra.

Dùng các giá trị xấp xỉ

Nếu sự chính xác không phải là vấn đề lớn, hãy dùng các kiểu dữ liệu có độ chính xác ít hơn. Trên các máy đời cũ hay cỡ nhỏ, hay các cơ cấu mô phỏng dấu chấm động trong phần mềm, số học dấu chấm động độ chính xác đơn thường nhanh hơn độ chính xác kép, do vậy thường dùng kiểu `float` thay cho kiểu `double` để tiết kiệm thời gian.

Viết lại bằng một ngôn ngữ cấp thấp hơn

Các ngôn ngữ cấp thấp hơn có xu hướng hiệu quả hơn dù sẽ tốn thời gian của lập trình viên hơn. Như vậy viết lại các phần trọng yếu của một chương trình C++ hay Java trong C hay thay một *script* dạng thông dịch bằng một chương trình viết bằng một ngôn ngữ ở dạng biên dịch có thể làm cho chương trình chạy nhanh hơn nhiều.

Đôi khi, ta có thể tăng tốc rất nhiều nếu dùng các mã lệnh có tính phụ thuộc phần cứng. Đây là phương cách cuối cùng, không phải là một bước nên làm hời hợt, vì nó sẽ phá hủy sự linh động của chương trình và làm cho việc bảo trì và sửa chữa trong tương lai trở nên khó khăn hơn nhiều. Hầu như các thao tác diễn đạt trong ngôn ngữ assembly là các hàm tương đối nhỏ nên được nhúng trong một thư viện; hàm `memset` và `memmove` hay các thao tác đồ họa là các thí dụ điển hình. Cách tiếp cận là việc viết mã lệnh rõ ràng đến mức tối đa có thể bằng một ngôn ngữ cấp cao và bảo đảm rằng chương trình đã viết đúng bằng cách kiểm nghiệm như đã mô tả trong cho hàm `memset` trong Chương 6. Đây là bản linh động của chương trình, có thể làm việc trên bất kỳ hệ thống nào, dù chậm. Khi chuyển sang một môi trường mới, ta có thể bắt đầu với một phiên bản đã biết chắc là làm việc được. Bây giờ khi viết một phiên bản bằng ngôn ngữ assembly, hãy kiểm tra hết mọi mặt của chương trình so với phiên bản linh động ở trên. Khi có lỗi,

hãy nghĩ ngay đến các mã lệnh không linh động; đó là cách thích hợp nhất để so sách các bản cài đặt với nhau.

Bài tập 7-4. Một cách để làm cho một hàm như hàm `memset` chạy nhanh hơn là viết hàm ở dạng các đoạn có kích cỡ từng từ thay vì từng *byte*, điều này có thể phù hợp tốt hơn với phần cứng và giảm phí tổn thời gian cho các vòng lặp đến một hệ số từ 4 tới 8. Điểm yếu là ở chỗ bây giờ có nhiều tác động đầu cuối cần phải giải quyết nếu mục tiêu không được sắp hàng trên biên của một từ và nếu chiều dài không phải là bội số của độ lớn của từ. Hãy viết một phiên bản của hàm `memset` thực hiện tối ưu hóa chỗ này. So sánh tốc độ thực thi với phiên bản thư viện hiện có và với một vòng lặp từng-*byte*-tại-một-thời-điểm trực tiếp.

Bài tập 7-5. Hãy viết một bộ cấp phát bộ nhớ `smalloc` cho các chuỗi ký tự C dùng bộ cấp phát đặc biệt cho các chuỗi ngắn nhưng gọi hàm `malloc` cho các chuỗi dài. Sẽ cần định nghĩa một `struct` để biểu diễn chuỗi ký tự trong cả hai trường hợp. Làm thế nào để quyết định khi nào chuyển hàm `smalloc` sang dùng hàm `malloc`?

7.5. Dùng hiệu quả không gian

Bộ nhớ đã luôn là tài nguyên tính toán quý giá nhất, luôn luôn thiếu hụt, và rất nhiều chương trình kém đã được viết ra với cố gắng chiếm phần lớn nhất của nguồn tài nguyên ít ỏi. Sự cố năm 2000 khét tiếng thường được trích dẫn làm ví dụ cho điều này: khi bộ nhớ thực sự khan hiếm, chỉ hai *byte* cần thiết để lưu trữ cặp số 19 đã được cho là quá đắt giá. Cho dù không gian có đúng là nguyên nhân thực sự của vấn đề hay không thì những mã lệnh như vậy đơn giản chỉ phản ánh cách con người ghi ngày tháng trong cuộc sống thường nhật, trong đó phần chỉ thế kỷ thường bị lược bỏ, điều này minh họa mối nguy hiểm cố hữu trong một sự tối ưu hóa thiển cận.

Trong bất kỳ tình huống nào, thời thế cũng đã thay đổi, và cả bộ nhớ chính cũng như các phương tiện nhớ thứ cấp nay đã rẻ một cách đáng ngạc nhiên. Do vậy hướng tiếp cận đầu tiên để tối ưu hóa không gian nên giống như để cải thiện tốc độ: đừng lo lắng.

Vẫn có những hoàn cảnh trong đó sự dùng hiệu quả không gian là một vấn đề. Nếu một chương trình không đủ với lượng bộ nhớ hiện có, vài phần sẽ được đưa vào phân trang, và điều này sẽ làm cho tốc độ thực thi trở nên không thể chấp nhận được. Ta gặp điều này ở các phiên bản mới của phần mềm lãng phí bộ nhớ; đó là một thực tế đáng buồn khi việc nâng cấp phần mềm thường kéo theo việc mua thêm bộ nhớ.

Tiết kiệm không gian bằng cách dùng kiểu dữ kiệu có kích thước nhỏ nhất

Một bước tiến tới việc dùng hiệu quả không gian là thực hiện những thay đổi nhỏ để dùng lượng bộ nhớ hiện hữu cách tốt hơn, ví dụ như bằng cách dùng kiểu dữ liệu nhỏ nhất có thể làm việc được. Điều này có thể có nghĩa là thay kiểu `int` bằng kiểu `short` nếu khớp được với dữ liệu; đây là kỹ thuật chung cho các tọa độ trong các hệ thống đồ họa hai chiều, vì 16 bit có thể quản lý bất kỳ tọa độ nào trên màn hình. Hay cũng có thể có nghĩa là thay kiểu `double` bằng kiểu `float`: vấn đề tiềm ẩn là mất đi độ chính xác, vì kiểu `float` thường chỉ giữ 6 hay 7 số thập phân.

Trong những trường hợp này cũng như những trường hợp tương tự, những thay đổi khác cũng có thể cần đến, đáng chú ý nhất là các định dạng chi tiết trong hàm `printf` và đặc biệt là các phát biểu `scanf`.

Sự mở rộng hợp lôgic của hướng tiếp cận này là mã hóa thông tin trong một *byte* hay với một số bit ít hơn, ngay chỉ với một bit nếu có thể được. Không nên dùng trường bit của C hay C++: chúng rất không linh động và có xu hướng sinh mã khối lượng lớn và không hiệu quả. Thay vào đó, gói gọn các phép toán muốn có trong các hàm tác động đến từng bit đơn trong từ hay một mảng các từ với các toán tử dời và toán tử mặt nạ. Hàm này trả về một nhóm các bit liên tục từ giữa một từ:

```
/* Hàm getbits: lấy n bit từ vị trí của p */
unsigned int getbits(unsigned int x, int p, int
n)
{
```

```
        return (x >> (p+1-n)) & ~(~0 << n);
    }
```

Nếu những hàm như vậy trở nên quá chậm, có thể cải tiến bằng kỹ thuật đã được mô tả trước đây trong chương này. Trong C++, sự định nghĩa chồng toán tử có thể được dùng để tạo ra sự truy cập vào bit giống như chỉ số dưới thông thường.

Không lưu trữ những gì có thể tính lại dễ dàng

Những thay đổi loại này là thiểu số, tuy nhiên, chúng tương tự như việc tinh chỉnh mã. Các cải tiến lớn thường do cấu trúc dữ liệu tốt hơn, có khi gắn liền với thay đổi thuật toán.

7.6. Ước tính

Khó mà ước tính được chương trình sẽ chạy nhanh như thế nào, và còn khó khăn gấp đôi để ước tính phí tổn thời gian của các phát biểu trong một ngôn ngữ hay chỉ thị trong một máy nhất định. Dù vậy, rất dễ tạo ra một mô hình phí tổn cho một ngôn ngữ hay cho một hệ thống ít nhất có thể cho một ý tưởng sơ bộ về những thao tác quan trọng kéo dài bao lâu.

Một cách tiếp cận thường được dùng cho các ngôn ngữ lập trình cổ điển là một chương trình đo thời gian các dãy mã lệnh đại diện. Chẳng hạn như, ta có một mô hình ước tính phí tổn, thời gian đo bằng nano-giây cho mỗi phép tính như sau:

Các thao tác trên số nguyên

```
i1++                 8
i1 = i2 + i3        12
i1 = i2 - i3        12
i1 = i2 * i3        12
i1 = i2 / i3       114
i1 = i2 % i3       114
```

Các thao tác trên số thực `float`

```
f1 = f2                 8
f1 = f2 + f3           12
f1 = f2 - f3           12
f1 = f2 * f3           11
f1 = f2 / f3           28
```

Các thao tác trên số thực `double`

```
d1 = d2                 8
d1 = d2 + d3           12
d1 = d2 - d3           12
d1 = d2 * d3           11
d1 = d2 / d3           58
```

Chuyển đổi số

```
i1 = f1                8
f1 = i1                8
```

Các thao tác trên số nguyên rất nhanh, ngoại trừ phép chia và phép chia lấy dư. Các thao tác trên dấu chấm động nhanh hoặc còn nhanh hơn, điều ngạc nhiên cho những ai đã quen với thời kỳ các phép tính dấu chấm động chậm hơn nhiều so với phép tính số nguyên.

Các thao tác cơ bản khác cũng rất nhanh, gồm có thao tác gọi hàm, ba hàng cuối cùng trong nhóm dưới đây:

Thao tác trên mảng số nguyên

```
v[i] = i               49
v[v[i]] = i            81
v[v[v[i]]] = i         100
```

Cấu trúc điều khiển

```
if (i == 5)  i1++            4
if (i != 5)  i1++            12
while (i < 0)  i1++          3
i1 = sum1(i2)                57
i1 = sum1(i2, i3)            58
i1 = sum1(i2, i3, i4)        54
```

Nhưng nhập và xuất không nhanh lắm, cũng như hầu hết các hàm thư viện khác:

Các thao tác nhập xuất

```
fputs(s, fp)             270
fget(s, 9, fp)           222
fprintf(fp, "%d\n", i)   1820
fscanf(fp, "%d", &i1)    2070
```

Các thao tác gọi hàm `malloc`

```
free(malloc(8))          342
```

Các hàm về chuỗi

```
strcpy(s, "0123456789")          157
i1 = strcmp(s, s)                176
i1 = strcmp(s, "a123456789")     64
```

Chuyển đổi kiểu chuỗi/số

```
i1 = atoi("12345")               402
sscanf("12345", "%d", &i1)       2376
sprintf(s, "%d", i)              1492
f1 = atof("123.45")              4098
sscanf("123.45", "%f", &f1)      6438
sprintf(s, "%6.2f", 123.45)      3902
```

Thời gian cho các hàm `malloc` và `free` có lẽ không thể hiện đúng tốc độ chạy của chúng, vì sự giải phóng ngay lập tức sau khi cấp phát không phải là một kiểu mẫu điển hình.

Cuối cùng là các hàm toán học:

```
i1 = rand()            135
f1 = log(f2)           418
f1 = exp(f2)           462
f1 = sin(f2)           514
f1 = sqrt(f2)          112
```

Những giá trị này tất nhiên sẽ khác ở những phần cứng khác nhau, nhưng chiều hướng chung có thể được dùng cho các ước tính nháp về thời gian chạy, hoặc cho việc so sánh phí tổn tương đối giữa xuất nhập và các phép tính cơ bản, hoặc cho việc quyết định liệu có nên viết lại một biểu thức hay nên dùng một hàm nội tuyến.

Có nhiều nguyên nhân gây ra các biến thể. Một trong số đó là mức tối ưu hóa trình biên dịch. Các trình biên dịch hiện đại có thể tìm được những sự tối ưu hóa vượt tầm đa số lập trình viên. Hơn nữa, các CPU hiện nay phức tạp đến nỗi chỉ có một trình biên dịch tốt là có thể tận dụng các khả năng phát ra nhiều chỉ thị đồng thời, phân luồng sự thi hành, tìm kiếm chỉ thị và dữ liệu trước khi cần đến, và những khả năng tương tự như vậy.

Kiến trúc của bản thân máy tính là một nguyên nhân chủ yếu khác làm cho giá trị bằng số của tốc độ thực thi khó dự đoán. Bộ nhớ truy cập nhanh (cache memory) làm nên một sự khác biệt lớn trong tốc độ, và phần lớn sự khôn ngoan trong thiết kế phần cứng đưa đến chỗ che giấu sự thực là bộ nhớ chính thực sự chậm hơn bộ nhớ truy cập nhanh một ít. Tốc độ đồng hồ thô của bộ xử lý như "400 MHz" có thể coi là một gợi ý nhưng không phải là tất cả. Một trong số các máy Pentium 200 MHz cũ của chúng tôi chậm hơn rõ rệt so với một máy Pentium 100 MHz cũ hơn vì chiếc máy sau có một lượng bộ nhớ truy cập nhanh lớn trong khi chiếc máy tính đầu không có. Và các thế hệ khác nhau của bộ xử lý, ngay cả có cùng cấu trúc, dùng

một số chu kỳ đồng hồ khác nhau cho một phép tính nhất định.

Bài tập 7-6. Tạo một tập hợp các trắc nghiệm để ước tính phí tổn thời gian của các thao tác cơ bản cho các máy tính và trình biên dịch bạn có, và nghiên cứu những sự tương tự và sự khác biệt trong tốc độ thực thi.

Bài tập 7-7. Tạo một mô hình phí tổn cho các hoạt động bậc cao hơn trong C++. Một số chức năng có thể bao gồm xây dựng, sao chép, xóa một lớp đối tượng; lệnh gọi các hàm thành viên; các hàm thực; các hàm nội tuyến; thư viện `iostream`; và STL. Bài tập này là bài tập mở, do đó chỉ nên xoay quanh một tập hợp nhỏ các hoạt động đại diện.

Bài tập 7-8. Làm lại bài tập trên trong Java.

7.7. Tổng kết

Một khi đã chọn được thuật toán đúng, tối ưu hóa tốc độ thực thi thường là điều lo ngại cuối cùng khi viết chương trình. Tuy nhiên, nếu buộc phải thực hiện thì chu trình cơ bản là đo đạc, tập trung vào một số chỗ mà sự thay đổi sẽ đem lại nhiều khác biệt nhất, kiểm tra lại các sửa chữa đã thêm vào, và đo lại. Nên dừng lại sớm nhất có thể, và dùng phiên bản đơn giản nhất làm cơ sở cho việc đo thời gian và sửa chữa.

Khi đang nỗ lực cải tiến tốc độ hoặc không gian tiêu thụ của chương trình, một ý tưởng hay là làm các trắc nghiệm chấm điểm hay ghi nhận các vấn đề trục trặc để có thể ước tính và nắm được tốc độ thực thi của chương trình.

Việc chấm điểm có thể được quản lý theo cùng một kiểu khung ta đã gặp trong Chương 6 về kiểm tra. Các trắc nghiệm thời gian được chạy tự động; ngõ xuất chứa đủ các nét nhận dạng để có thể hiểu và tái tạo được; các bản ghi được giữ lại để có thể quan sát các xu hướng và các thay đổi có ý nghĩa.

.Chương 8

TÍNH KHẢ CHUYỂN

Rất khó khăn để viết được một phần mềm chạy chính xác và hiệu quả. Khi một chương trình đã chạy tốt trong một môi trường nào đó, bạn không muốn phải thay đổi nhiều khi chuyển nó sang một trình biên dịch, bộ xử lý hay hệ điều hành khác. Lý tưởng nhất là chẳng cần thay đổi gì cả.

Ý tưởng này được gọi là tính khả chuyển. Thực tế "tính khả chuyển" được hiểu đơn giản hơn, đó là việc dễ dàng điều chỉnh chương trình khi có nhu cầu chuyển chúng tới môi trường khác hơn là phải viết lại từ đầu. Sự thay đổi càng ít thì tính khả chuyển càng cao.

Có lẽ bạn sẽ tự hỏi tại sao phải lo lắng về tính khả chuyển. Nếu phần mềm chỉ chạy trong một môi trường, dưới những điều kiện nhất định, tại sao phải mất thời gian làm cho nó có thể sử dụng rộng hơn? Thứ nhất, bất kỳ một chương trình tốt nào, theo định nghĩa có thể được dùng bất kỳ ở đâu dưới bất kỳ hình thức nào. Xây dựng phần mềm tổng quát hơn đặc tả ban đầu của nó thì sự bảo trì sẽ đơn giản và tiện lợi hơn khi sử dụng. Thứ nhì, môi trường luôn thay đổi. Khi trình biên dịch, hệ điều hành hay phần cứng thay đổi hoặc nâng cấp, đặc tính của chúng thay đổi. Chương trình càng ít phụ thuộc vào các đặc tính cụ thể nó càng ít bị trục trặc và thích ứng với các tình huống thay đổi. Cuối cùng, quan trọng nhất, một chương trình có tính khả chuyển thì vẫn tốt hơn. Sự nỗ lực làm cho một chương trình có tính khả chuyển cũng làm cho việc thiết kế, cấu trúc chương trình được dễ dàng hơn, cũng như việc kiểm tra được thông suốt hơn. Nói chung những kỹ thuật làm cho một chương trình có tính khả chuyển cũng gần nghĩa với những kỹ thuật làm cho một chương trình chạy tốt hơn.

Dĩ nhiên mức độ khả chuyển cũng tùy theo tình hình thực tế. Không có chương trình có tính khả chuyển tuyệt đối, chỉ là một chương trình chưa được thử trong những môi trường khác nhau. Nhưng tính khả chuyển là mục đích của chúng ta khi viết phần mềm để chúng có thể chạy khắp nơi mà không cần phải thay đổi nhiều. Ngay cả khi mục đích này không được thỏa mãn hoàn toàn, thời gian dành cho việc giải quyết tính khả chuyển sẽ được đền đáp khi nâng cấp phần mềm.

Hãy cố gắng viết phần mềm có thể chạy trên phần chung của nhiều tiêu chuẩn, giao diện và môi trường khác nhau. Đừng sửa các lỗi liên quan đến tính khả chuyển bằng cách thêm vào đoạn mã nguồn riêng, thay vì vậy hãy sửa lại phần mềm cho nó hoạt động với các ràng buộc mới. Dùng tính trừu tượng và tính đóng gói dữ liệu để giới hạn và điều khiển phần mã nguồn không khả chuyển. Bằng cách đứng bên trong phần chung của các ràng buộc và cục bộ hóa sự phụ thuộc hệ thống, mã nguồn của bạn sẽ trở nên rõ ràng và tổng quát hơn khi sử dụng chúng.

8.1. Ngôn ngữ

Tuân theo chuẩn

Dĩ nhiên bước đầu tiên để viết mã nguồn có tính khả chuyển là lập trình với ngôn ngữ cấp cao, ngôn ngữ chuẩn càng tốt. Tập tin nhị phân không có tính khả chuyển tốt, nhưng mã nguồn thì ngược lại. Ngay cả khi trình biên dịch chuyển một chương trình sang mã máy không hoàn toàn theo định nghĩa, thậm chí đó là ngôn ngữ chuẩn. Một vài ngôn ngữ sử dụng rộng rãi chỉ có một các cài đặt duy nhất: do đó có nhiều nhà cung cấp, hoặc có nhiều phiên bản cho các hệ điều hành khác nhau. Cách thức thông dịch mã nguồn cũng khác nhau.

Tại sao không có chuẩn hay một định nghĩa chặt chẽ? Đôi lúc chuẩn chưa hoàn thiện và không thể định nghĩa các hành vi khi các tính năng tương tác lẫn nhau. Đôi khi sự ràng buộc lại không chặt chẽ: ví dụ kiểu dữ liệu char trong C và C++ lúc có dấu, lúc không có dấu, và không cần chính xác là 8 bit. Tùy theo người viết trình biên dịch bỏ qua những vấn đề như

vậy để cho việc cài đặt được hiệu quả hơn và tránh bị giới hạn về phần cứng khi dùng ngôn ngữ này, hậu quả là rủi ro sẽ tăng lên khi viết chương trình. Các nguyên tắc và các vấn đề kỹ thuật tương thích nảy sinh có thể dẫn đến sự thỏa hiệp không rõ ràng. Cuối cùng, vì ngôn ngữ khó hiểu và trình biên dịch thường rắc rối nên dễ dàng phát sinh lỗi trong quá trình thông dịch và có lỗi trong khi quá trình cài đặt.

Vì vậy, mặc dù các tiêu chuẩn và các sổ tay tham khảo đã nêu ra các đặc tả chặt chẽ, chúng chưa bao giờ định nghĩa một ngôn ngữ đầy đủ, và các cách thức cài đặt khác nhau có thể thông dịch hợp lệ nhưng không tương thích. Xét khai báo không hợp lệ sau trong C và C++:

```
?       *x[] = {"abc"};
```

qua kiểm tra hàng chục trình biên dịch cho thấy một số chẩn đoán đúng lỗi thiếu khai báo kiểu `char` cho `x`, một số báo lỗi *"mismatched types"*, và một số đã biên dịch thành công đoạn mã nguồn không hợp lệ này.

Lập trình theo xu hướng chính

Cần phải tránh những điểm mơ hồ của ngôn ngữ lập trình – ví dụ như các trường bit trong C và C++. Chỉ nên sử dụng những tính năng mà ngôn ngữ đã định nghĩa rõ ràng và dễ hiểu. Vì những tính năng này thông dụng và xử lý như nhau trong nhiều ngôn ngữ. Chúng ta gọi đó là *xu hướng chính* của ngôn ngữ.

Thật khó biết được đâu là xu hướng chính, nhưng có thể dễ dàng nhận ra thông qua các cấu trúc tốt bên ngoài của ngôn ngữ. Những đặc trưng mới như dấu chú thích (`//`) và kiểu `complex` trong C, hoặc những đặc trưng cụ thể cho một kiến trúc như từ khóa `near` và `far` chắc chắn là nguyên nhân của vấn đề này. Nếu như có một đặc tính mà bạn không rõ thì đừng dùng nó và nên tham khảo ý kiến chuyên gia về ngôn ngữ mà bạn đang quan tâm.

Để thảo luận vấn đề này, chúng ta sẽ tập trung vào C và C++, những ngôn ngữ đa dụng và đã được sử dụng rộng rãi để viết phần mềm có tính khả chuyển.

Xu hướng chính của ngôn ngữ C là gì? Cách nói này thường đề cập đến phong cách của ngôn ngữ. Xét phiên bản C gốc không cần phải khai báo *prototype*, hàm `sqrt` được khai báo như sau:

```
?       double sqrt();
```

hàm này định nghĩa kiểu trả về nhưng không có các tham số. ANSI C đã thêm vào cách khai báo *prototype* như sau:

```
double sqrt(double);
```

các trình biên dịch ANSI C chấp nhận kiểu khai báo không có *prototype*, nhưng tốt nhất là nên khai báo *prototype*, như thế sẽ an toàn hơn vì các lời gọi hàm sẽ được kiểm tra đầy đủ, và nếu có bất kỳ thay đổi nào trong quá trình giao tiếp giữa các hàm thì trình biên dịch sẽ nhận ra. Nếu bạn gọi hàm

```
func(7, PI);
```

nếu hàm `func` không khai báo *prototype* thì trình biên dịch sẽ không kiểm tra rằng hàm `func` này có được gọi chính xác hay không. Nếu sau đó hàm `func` bị sửa trở thành có ba tham số, thì việc sửa lại phần mềm có thể bị sai vì kiểu cú pháp cũ không có khả năng kiểm tra các thông số của hàm.

C++ là một ngôn ngữ rộng hơn với nhiều chuẩn, do đó khó xác định xu hướng chính của nó. Ví dụ, mặc dù chúng ta mong đợi STL sẽ là xu hướng chính, nhưng điều đó sẽ không xảy ra ngay, và nhiều phiên bản chưa hỗ trợ STL hoàn toàn.

Kích thước của các kiểu dữ liệu

Kích thước của các kiểu dữ liệu cơ bản trong C và C++ không được định nghĩa rõ ràng; chỉ có một quy luật đơn giản:

```
sizeof(char) ≤ sizeof(short) ≤ sizeof(int) ≤ sizeof(long)

sizeof(float) ≤ sizeof(double)
```

và char ít nhất là 8 bit, short và int ít nhất là 16 bit, và long ít nhất là 32 bit, tuy nhiên không có gì bảo đảm cho các quy định này. Thậm chí không yêu cầu giá trị của con trỏ phù hợp với kiểu dữ liệu int.

Chúng ta dễ dàng biết được kích thước của các kiểu dữ liệu trên một trình biên dịch cụ thể:

```
/* sizeof: hiển thị kích thước của các kiểu dữ liệu cơ bản */

int main(void)

{

    printf("char %d, short %d, int %d, long %d,",

        sizeof(char),sizeof(short),

        sizeof(int),sizeof(long));

    printf("float %d, double %d, void* %d, %d\n",

        sizeof(float),sizeof(double),
sizeof(void*));

        return 0;

}
```

Hầu hết các kết quả là như sau:

```
char 1, short 2, int 4, long 4, float 4, double 8, void* 4
```

tuy nhiên cũng có ngoại lệ. Một số hệ thống 64-bit cho kết quả:

```
char 1, short 2, int 4, long 8, float 4, double 8, void* 8
```

và các máy PC thời kỳ đầu lại cho kết quả:

```
char 1, short 2, int 2, long 4, float 4, double 8, void* 2
```

Trước đây, phần cứng của những máy PC hỗ trợ một vài kiểu dữ liệu con trỏ. Sự lộn xộn này nảy sinh ra hai từ khóa `far` và `near` cho con trỏ, tuy nhiên chúng không phải là chuẩn, vấn đề này đã gây không ít khó khăn cho các trình biên dịch hiện tại. Nếu như trình biên dịch của bạn có thể thay đổi kích thước của các kiểu dữ liệu cơ bản, hay nếu máy của bạn có các kiểu dữ liệu cơ bản với kích thước khác nhau thì cứ việc biên dịch và chạy thử chương trình của bạn trong các cấu hình khác nhau này.

Tập tin chuẩn `stddef.h` định nghĩa một số kiểu dữ liệu có thể giúp ta viết chương trình có tính khả chuyển. Thông dụng nhất là kiểu `size_t`, đó là kiểu dữ liệu số nguyên không dấu trả về bởi toán tử `sizeof`. Những giá trị của kiểu này được trả về bởi một số hàm như `strlen` và được dùng như là đối số của nhiều hàm khác, `malloc` chẳng hạn.

Trong Java kích thước của các kiểu dữ liệu cơ bản được định nghĩa rõ ràng: `byte` là 8 bit, `char` và `short` là 16 bit, `int` là 32 và `long` là 64 bit.

Chúng ta sẽ không đề cập đến những rắc rối tiềm tàng của việc tính toán dấu chấm động bởi vì đó là một vấn đề lớn. May mắn là hầu hết các máy tính hiện nay đều hỗ trợ chuẩn IEEE cho các phần cứng có khả năng tính toán dấu chấm động, như vậy các thuật toán cho việc tính toán dấu chấm động được thực hiện khá hợp lý.

Thứ tự thực hiện

Trong C và C++, thứ tự khi thực hiện của các thao tác trong một biểu thức, và các tham số hàm không được định nghĩa rõ ràng. Chẳng hạn như, trong phép gán sau:

```
?       n = (getchar() << 8) | getchar();
```

hàm `getchar()` thứ hai có thể được gọi trước; vì thế phép gán có thể được xử lý không như mong đợi. Xem đoạn mã nguồn sau:

```
?       ptr[count] = name[++count];
```

biến `count` có thể tăng trước hoặc sau khi nó làm chỉ mục cho `ptr` , và trong

```
?     printf("%c %c\n", getchar(), getchar());
```

ký tự đầu có thể được in ra sau thay vì được in ra trước. Tương tự thế, cách viết:

```
?     printf("%f %s\n", log(-1.23), strerror(errno));
```

giá trị của lỗi có thể được tính trước khi hàm `log` được gọi.

Có một số quy luật khi đánh giá các biểu thức. Theo quy định tất cả các hiệu ứng lề và lời gọi hàm phải được hoàn thành tại dấu chấm phẩy, hoặc khi một hàm được gọi. Toán tử `&&` và `||` được thi hành từ trái sang phải cho đến khi giá trị cuối cùng được xác định (gồm cả các hiệu ứng lề).

Thứ tự thực hiện trong Java được định nghĩa rất chặt chẽ. Nó yêu cầu các biểu thức, gồm cả các hiệu ứng lề, phải được thực hiện từ trái sang phải, tuy nhiên một số chuyên gia cho rằng cũng không nhất thiết phụ thuộc vào điều này khi viết chương trình. Lời khuyên này có giá trị trong trường hợp có nhu cầu chuyển mã nguồn từ Java sang C hoặc C++. Chuyển mã nguồn từ ngôn ngữ này sang ngôn ngữ khác là điều bất đắc dĩ, nhưng lại hữu ích khi muốn kiểm tra tính khả chuyển.

Dấu của kiểu dữ liệu `char`

Trong C và C++ không xác định rõ kiểu dữ liệu `char` có dấu hay không. Điều này gây ra rắc rối khi kết hợp kiểu dữ liệu `char` với `int`, chẳng hạn trong đoạn mã nguồn sau, trả về giá trị `int` từ hàm `getchar()`. Nếu bạn viết:

```
?     char c; /*nên là kiểu int*/

?     c = getchar();
```

theo thông thường kiểu dữ liệu ký tự là 8-bit, thì giá trị của `c` nằm giữa 0 và 255 nếu `char` không dấu, còn nếu có dấu thì `c` có giá trị từ -128 tới 127. Điều này có thể khi kiểu dữ liệu character được dùng như một mảng hoặc được dùng để kiểm tra `EOF`, có giá trị là `-1` trong thư viện `stdio`. Ví dụ như, chúng ta đã phát triển đoạn mã nguồn trong phần 6.1 sau khi sửa đổi một vài

điều kiện về giới hạn so với bản gốc. Việc so sánh `s[i] == EOF` sẽ luôn sai nếu kiểu `char` không dấu:

```
?      int i;
?      char s[MAX];
?
?      for (i = 0; i < MAX-1; i++)
?       if ((s[i] = getchar()) == '\n' || s[i] == EOF)
?                 break;
?      s[i] = '\0';
```

Khi hàm `getchar` trả về `EOF`, giá trị 255 (`0xFF`, kết quả của việc chuyển đổi từ `-1` sang kiểu dữ liệu `char` không dấu) sẽ được giữ trong `s[i]`. Nếu `s[i]` là không dấu, giữ giá trị là 255 khi so sánh với `EOF` luôn sai.

Tuy nhiên ngay cả khi kiểu `char` có dấu, đoạn mã nguồn trên vẫn không chính xác. Việc so sánh sẽ thành công tại `EOF`, nhưng một giá trị *byte* hợp lệ `0xFF` nhập vào giống như `EOF` sẽ ngắt vòng lặp sớm. Vì vậy để tránh vấn đề rắc rối này, bạn dùng kiểu dữ liệu `int` để lưu giá trị trả về của hàm `getchar` để so sánh với `EOF`. Dưới đây là cách viết vòng lặp có tính khả chuyển:

```
int c, i;
      char s[MAX];
for (i = 0; i < MAX-1; i++) {
      if ((c = getchar()) == '\n' || c == EOF)
            break;
      s[i] = c;
}
s[i] = '\0';
```

Java không có từ khóa `unsigned`; kiểu dữ liệu số nguyên là có dấu còn kiểu dữ liệu `char` (16-bit) thì không.

Phép dịch chuyển số học hay logic

Phép dịch phải của các đại lượng với toán tử `>>` có thể là phép dịch chuyển số học (một bản sao các bit dấu được sinh ra trong khi dịch chuyển) hay logic (số 0 điền vào các bit trống trong khi dịch chuyển). Một lần nữa, Java đã rút kinh nghiệm vấn đề này từ C và C++, Java dùng `>>` cho phép dịch phải số học và dùng `>>>` cho phép dịch phải logic.

Sự canh chỉnh cấu trúc và các thành phần lớp

Canh chỉnh lại các thành phần trong cấu trúc, và lớp không được xác định rõ ràng, ngoại trừ cách sắp đặt trật tự các thành viên của một khai báo. Ví dụ:

```
struct X {
        char c;
        int i;
};
```

địa chỉ trong bộ nhớ của `i` có thể là 2, 4, hay 8 byte từ vị trí bắt đầu của cấu trúc. Một vài máy cho phép `int` được lưu trữ tại các vùng lẻ, nhưng hầu hết yêu cầu các kiểu dữ liệu chính *n*-byte phải được lưu trữ tại một vùng biên *n*-byte, ví dụ kiểu dữ liệu `double` có chiều dài 8 byte được lưu trữ tại các địa chỉ chia hết cho 8. Tuy nhiên do nhu cầu về hiệu suất, trình biên dịch có thể được thay đổi sao cho phù hợp.

Bạn không nên cho rằng các thành phần của một cấu trúc nằm liên tiếp nhau trong bộ nhớ. Khi cấp phát bộ nhớ cho cấu trúc `X` thì ít nhất một byte được cấp phát dư. Những byte cấp phát dư này ý là kích thước thật của một cấu trúc lớn hơn tổng kích thước của các thành viên cộng lại, và thay đổi từ máy này sang máy khác. Kích thước thật của cấu trúc `x` là `sizeof(struct x)` chứ không phải là `sizeof(char) + sizeof(int)`.

Các trường bit

Các trường bit phụ thuộc rất nhiều vào phần cứng do đó không nên dùng chúng.

Tóm lại, những lỗi tiềm tàng này có thể khắc phục được nếu tuân theo một số quy tắc. Không sử dụng các hiệu ứng lề ngoại trừ một vài cách lập trình như:

```
a[i++] = 0;

c = *p++;

*s++ = *t++;
```

Không so sánh một ký tự với `EOF`. Luôn dùng `sizeof` để tính kích thước của một kiểu dữ liệu hay một đối tượng. Không bao giờ dịch phải một giá trị có dấu. Bảo đảm kiểu dữ liệu đủ lớn cho phạm vi của các giá trị mà bạn muốn lưu trữ trong nó.

Thử dùng một số trình biên dịch

Cũng đơn giản để nghĩ rằng bạn hiểu được tính khả chuyển, nhưng các trình biên dịch lại gặp một số vấn đề mà bạn không ngờ, và các trình biên dịch khác nhau có cách nhìn không giống nhau về chương trình của bạn. Vì vậy, bạn nên quan tâm đến sự hỗ trợ của các trình biên dịch. Đọc kỹ tất cả các cảnh báo của trình biên dịch. Cần thử nghiệm nhiều trình biên dịch trên một máy và trên các máy khác nhau. Thử dùng trình biên dịch C++ cho chương trình C.

Do các trình biên dịch chấp nhận các chương trình một cách khác nhau, việc chương trình của bạn biên dịch với một trình biên dịch là không bảo đảm ngay cả với sự chính xác về cú pháp. Tuy nhiên nếu một vài trình biên dịch chấp nhận mã nguồn của bạn thì khả năng thành công rất cao.

Tất nhiên, nhiều trình biên dịch cũng là vấn đề của tính khả chuyển, do có nhiều sự chọn lựa khác nhau cho các hành vi không rõ ràng. Nhưng chúng ta vẫn còn hy vọng vào cách tiếp cận của chúng ta. Thay vì viết mã

nguồn khác nhau cho các hệ thống, môi trường và trình biên dịch khác nhau, chúng ta cố gắng tạo ra các phần mềm không phụ thuộc vào sự khác biệt này.

8.2. Tập tin tiêu đề (header) và thư viện

Các tập tin tiêu đề và các thư viện cung cấp các dịch vụ bổ sung thêm cho các ngôn ngữ cơ bản. Ví dụ, việc nhập dữ liệu và xuất dữ liệu thông qua thư viện `stdio` của C, `iostream` của C++, và `java.io` của Java. Thật ra chúng không phải thành phần của ngôn ngữ, nhưng chúng đi kèm một cách khăng khít với ngôn ngữ và được chấp nhận như là một thành phần của môi trường có hỗ trợ ngôn ngữ này. Nhưng bởi vì các thư viện chứa đựng rất lớn các tiện ích viết sẵn, và thường giải quyết cụ thể cho một hệ điều hành nào đó, do đó chúng tiềm ẩn các vấn đề về tính khả chuyển.

Dùng các thư viện chuẩn

Đối với mỗi ngôn ngữ nên: bám sát chuẩn, và các thành phần đã được xây dựng ổn định. Ngôn ngữ C định nghĩa một thư viện chuẩn các hàm cho việc nhập và xuất dữ liệu; xử lý chuỗi, ký tự; cấp phát, lưu trữ vùng nhớ; và nhiều tác vụ khác. Nếu bạn hạn chế hệ điều hành của bạn tương tác với các hàm này, thì mã nguồn của bạn có khả năng xử lý và thi hành giống nhau khi di chuyển từ hệ thống này sang hệ thống khác. Nhưng bạn cũng nên cẩn thận, bởi vì có nhiều phiên bản khác nhau cho các thư viện và một số có các đặc tính không tuân theo chuẩn.

ANSI C không định nghĩa hàm sao chép chuỗi `strdup` (tuy thế hầu hết các môi trường hỗ trợ nó) ngay cả khi điều này được yêu cầu thành chuẩn. Một lý do nữa là các lập trình viên có thói quen dùng hàm `strdup` mà không biết rằng đó không phải là chuẩn. Do đó chương trình trình sẽ thất bại khi biên dịch ở môi trường khác không hỗ trợ hàm này. Loại vấn đề này là nguyên nhân chính gây rắc rối cho tính khả chuyển khi dùng các thư viện; giải pháp duy nhất là bám sát chuẩn và kiểm tra chương trình của bạn trên nhiều môi trường khác nhau.

Các tập tin tiêu đề và các thao tác khai báo phương thức giao tiếp cho các hàm chuẩn. Một vấn đề là các tiêu đề thường có khuynh hướng lộn xộn bởi vì chúng cố gắng đưa một số ngôn ngữ vào cùng một tập tin. Ví dụ, ta thường tìm thấy một tập tin tiêu đề như `stdio.h` dùng chung cho các trình biên dịch – ANSI C cũ, ANSI C, và ngay cả với C++. Trong những trường hợp này, tập tin đó sẽ chứa lộn xộn các cú pháp định hướng như `#if` và `#ifdef`. Bởi vì các ngôn ngữ lập trình trước đây không được uyển chuyển lắm, nên các tập tin thường phức tạp và khó đọc, đôi khi còn chứa lỗi.

Xét đoạn trích sau từ một tập tin tiêu đề:

```
?     #ifdef _OLD_C
?           extern int fread();
?           extern int fwrite();
?     #else
?     # if defined(__STDC__) || defined(__cplusplus)
?  extern size_t fread( void*, size_t, size_t, FILE*);
?  extern size_t fwrite( const void*, size_t, size_t,
FILE*);
?     #else /*not __STDC__ || __cplusplus */
?  extern size_t fread();
?  extern size_t fwrite();
?     # endif /*else not __STDC__ || __cplusplus */
?     #endif
```

mặc dù chúng được viết khá rõ ràng, nhưng nó cũng chỉ ra rằng các tập tin tiêu đề có dạng như trên rất khó hiểu và khó bảo trì. Dễ dàng hơn nếu ta dùng các tập tin tiêu đề khác nhau cho các trình biên dịch và môi trường khác nhau. Điều này đòi hỏi phải bảo trì các tập tin tiêu đề riêng biệt, nhưng

điều này thích hợp cho các hệ thống cụ thể, và nó cũng giảm thiểu các lỗi tương tự như lỗi `strdup` trong môi trường ANSI C.

8.3. Tổ chức chương trình

Có hai hướng chính cho tính khả chuyển: sự hợp nhất và sự giao nhau. Hướng tiếp cận sự hợp nhất, dùng các đặc tính tốt nhất của các hệ thống riêng biệt, dựa vào đó tạo ra các tiến trình cài đặt và biên dịch phù hợp cho các môi trường cục bộ. Kết quả mã nguồn giải quyết được thống nhất tất cả các tình huống, tận dụng được thế mạnh của các hệ thống. Tuy nhiên một điều trở ngại là khối lượng và độ phức tạp của tiến trình cài đặt, cũng như mã nguồn sẽ khó hiểu hơn do có các điều kiện biên dịch.

Chỉ sử dụng các đặc tính có sẵn ở nhiều nơi

Hướng tiếp cận sự giao nhau chỉ dùng các đặc trưng tồn tại ở tất cả các hệ thống; đừng dùng một đặc trưng nào mà nó không tồn tại ở mọi nơi. Một điều nguy hiểm là sự yêu cầu phải có sẵn các đặc trưng ở mọi nơi có thể làm giới hạn số lượng các hệ thống hoặc khả năng của chương trình biên dịch; và phải tổn thất về hiệu suất ở một số môi trường.

Để so sánh hai hướng tiếp cận này, ví dụ sau dùng mã nguồn theo hướng sự hợp nhất và hướng sự giao nhau. Bạn sẽ thấy, mã nguồn hợp nhất được viết không có tính khả chuyển, bất chấp mục tiêu đã định, trong khi mã nguồn giao nhau cũng không có tính khả chuyển nhưng thường đơn giản hơn.

Ví dụ nhỏ sau cố gắng phù hợp với các môi trường, không có tập tin chuẩn `stdlib.h`:

```
?     #if defined (STDC_HEADERS) || defined (_LIBC)
?     #include <stdlib.h>
?     #else
?     extern void *malloc(unsigned int);
?     extern void *realloc(void *, unsigned int);
```

```
?     #endif
```

Kiểu đối phó này chấp nhận được nếu tình huống trên ít khi xảy ra, nhưng thất bại nếu tình huống trên xảy ra thường xuyên. Nó cũng đặt ra câu hỏi hóc búa đó là cần khai báo bao nhiêu hàm theo cách này? Nếu chương trình dùng hàm `malloc` và `realloc` thì chắc chắn nó cũng dùng hàm `free`. Chuyện gì xảy ra nếu các tham số `unsigned int` của `malloc` và `realloc` không bằng `size_t`? Ngoài ra làm sao ta biết `STDC_HEADERS` hay `_LIBC` đã được định nghĩa đúng? Và làm sao ta chắc chắn không có một trong những cái tên vô tình gọi đến một số hàm đặc biệt nào đó trong một số môi trường? Bất cứ đoạn mã có điều kiện nào giống như trên đều không đầy đủ - không khả chuyển - bởi vì tồn tại những hệ thống không thỏa những điều kiện mà ta nghĩ là đã tổng quát, khi đó ta lại phải điều chỉnh `#ifdef`. Nếu chúng ta có thể giải quyết vấn đề mà không dùng các phát biểu điều kiện phức tạp, chúng ta có thể loại trừ được các vấn đề bảo trì phức tạp.

Vẫn còn những vấn đề tương tự cần giải quyết, và làm thế nào chúng ta giải quyết được chúng trọn vẹn? Một cách lý tưởng ta luôn cho rằng các tập tin tiêu đề chuẩn đã tồn tại; đây cũng chính là một vấn đề nếu ta nghĩ vậy. Sẽ dễ dàng hơn khi phát hành phần mềm ta kèm theo một tập tin tiêu đề định nghĩa chính xác các hàm `malloc`, `realloc` và `free` như ANSI C. Tập tin này có thể luôn luôn được dùng, thay vì phải dán thẳng váo mã nguồn. Do đó ta luôn chắc rằng các interface luôn có sẵn.

Tránh điều kiện biên dịch

Điều kiện biên dịch với `#ifdef` và các từ khóa tương tự rất khó quản lý.

```
?     #ifdef NATIVE
?           char *astring = "convert ASCII to native
character set";
?     #else
?     #ifdef MAC
```

```
?           char *astring = "convert to Mac textfile
format";
?       #else
?       #ifdef DOS
?           char *astring = "convert to DOS textfile
format";
?       #else
?           char *astring = "convert to Unix textfile
format";
?       #endif /* ?DOS */
?       #endif /* ?MAC */
?       #endif /* ?NATIVE */
```

Đoạn trích trên sẽ tốt hơn với `#elif` sau mỗi lời định nghĩa, hơn là một đống các `#endif` ở phía dưới. Nhưng vấn đề thực sự là đoạn mã nguồn trên tính khả chuyển rất kém (mặc dù ý định của nó là tính khả chuyển) bởi vì nó xử lý khác nhau trên mỗi hệ thống và cần phải cập nhật thêm `#ifdef` cho các môi trường mới. Một chuỗi diễn đạt tổng quát, đơn giản hơn, hoàn toàn có tính khả chuyển, và cung cấp nhiều thông tin hơn là:

```
        char *astring = "convert to local text format";
```

Không cần phải thêm các đoạn mã nguồn điều kiện do nó giống nhau trên tất cả các hệ thống.

Việc trộn lẫn các dòng điều khiển biên dịch (dùng câu phát biểu `#ifdef`) với dòng điều khiển trong lúc thực thi làm vấn đề tệ hơn, vì rất khó đọc.

```
?       #ifndef DISKSYS
?           for (i = 1; i <= msg->dbgmsg.msg_total; i++)
```

```
?      #endif
?      #ifdef DISKSYS
?            i = dbgmsgno;
?            if (i <= msg->dbgmsg.msg_total)
?      #endif
?            {
?                  …
?                  if ( msg->dbgmsg.msg_total == i)
?      #ifndef DISKSYS
?                  break; /*không còn message nào để đợi
*/
?      khoảng 30 dòng, với nhiều điều kiện biên dịch
?      #endif
?            }
```

Ngay cả khi có vẻ ổn, các điều kiện biên dịch có thể được thay thế bằng cách khác rõ ràng hơn. Ví dụ, `#ifdefs` thường dùng cho việc điều khiển debug mã nguồn:

```
?      #ifdef DEBUG
?            printf(…);
?      #endif
```

nhưng thông thường câu lệnh `if` với một hằng điều kiện sẽ tiện hơn như trong ví dụ sau:

```
enum { DEBUG = 0 };
…
if (DEBUG) {
```

```
        printf(...);
    }
```

Nếu DEBUG bằng không thì hầu hết các trình biên dịch không phát sinh mã gì, nhưng chúng sẽ kiểm tra cú pháp của đoạn mã nguồn thêm vào. Nhưng phát biểu #ifdef có thể che giấu lỗi cú pháp cho đến khi nào phát biểu #ifdef được thực hiện.

Đôi khi điều kiện biên dịch gộp vào các đoạn mã nguồn rất lớn:

```
#ifdef notdef /* undefined symbol */
    ...
#endif
```

hay

```
#if 0
    ...
#endif
```

nhưng đoạn mã nguồn điều kiện có thể được bỏ qua bằng cách cho chúng vào các tập tin riêng trong khi biên dịch. Chúng ta sẽ quay lại vấn đề này trong phần sau.

Khi bạn phải điều chỉnh chương trình cho phù hợp với một môi trường mới, đừng bắt đầu bằng cách tạo một bản sao của chương trình mà nên sửa trực tiếp trên mã nguồn luôn. Có thể bạn sẽ phải thay đổi phần chính của đoạn mã nguồn, và nếu như bạn thay đổi trên bản sao thì về sau sẽ có các phiên bản không đồng nhất. Tốt hơn là chỉ có một mã nguồn cho mỗi chương trình; nếu như bạn thay đổi mội vài thứ khi chuyển tới một môi trường cụ thể nào đó, thì nên tìm cách thay đổi sao cho chúng cũng chạy tốt ở nơi khác. Thay đổi các phương thức giao tiếp bên trong nếu cần, nhưng giữ cho mã nguồn được nhất quán. Điều này làm cho mã nguồn có tính khả chuyển cao, hơn là chỉ tập trung vào một vài môi trường cụ thể. Thu hẹp thành sự giao nhau hơn là mở rộng tính thống nhất.

Chúng ta đã nhấn mạnh vấn đề gây ra bởi các điều kiện biên dịch thông qua một số ví dụ. Nhưng vấn đề chính chúng ta chưa để ý: đó là rất khó kiểm tra các điều kiện biên dịch. Một `#ifdef` biến một chương trình đơn thành hai chương trình riêng biệt khi biên dịch. Rất khó biết khi nào các chương trình khác nhau được biên dịch và kiểm tra. Nếu chúng ta thay đổi trong khối `#ifdef` , có thể phải thay đổi ở nơi khác, nhưng sự thay đổi này chỉ có thể kiểm tra được trong môi trường nào đó thực sự thi hành đoạn `#ifdef`. Nếu một thay đổi tương tự cho các cấu hình khác, thì không thể kiểm tra được. Cũng vậy, khi ta thêm một khối `#ifdef` vào thì rất khó cách ly sự thay đổi, để xác định điều kiện nào cần phải thỏa cũng như nơi nào trong chương trình phải điều chỉnh cho phù hợp với sự thay đổi này. Cuối cùng, nếu điều gì trong đoạn mã nguồn bị bỏ qua do điều kiện biên dịch, thì trình biên dịch không thể nhìn thấy chúng được. Điều này gây ra những tình huống rất khó lường trước, nếu như một môi trường nào đó kích hoạt các điều kiện biên dịch này. Chương trình sau biên dịch thành công khi `_MAC` được định nghĩa và thất bại nếu ngược lại:

```
#ifdef _MAC
      printf("Là máy Macintosh\r");
#else
      printf("Không đúng cú pháp Macintosh");
#endif
```

Vì thế mục tiêu của chúng ta là chỉ dùng các đặc trưng thông dụng trên tất cả các môi trường đích. Chúng ta có thể biên dịch và kiểm chứng tất cả các đoạn mã nguồn. Nếu có thứ gì đó gây ra vấn đề về tính khả chuyển, ta có thể viết lại nó hay hơn cách thêm vào các điều kiện biên dịch; cách này làm tăng tính khả chuyển và cải tiến chương trình tốt hơn.

Một số hệ thống phân tán với một script tùy biến mã nguồn cho môi trường cục bộ. Vào thời điểm biên dịch, đoạn script kiểm tra các thuộc tính môi trường - định vị các thư viện, các tập tin tiêu đề, thứ tự `byte` trong các dữ liệu `word`, kích thước các kiểu dữ liệu... và phát sinh ra các tham số cấu

hình phù hợp với tình huống đang gặp. Các đoạn script này có thể lớn và rắc rối, một phần đáng kể của các phần mềm phân tán, và đòi hỏi phải thường xuyên bảo trì cho chúng làm được việc. Đôi khi những kỹ thuật này là cần thiết cho mã nguồn càng khả chuyển thì sự cấu hình và cài đặt càng đơn giản.

8.4. Sự cô lập

Mặc dù chúng ta mong muốn có một mã nguồn duy nhất có thể biên dịch trên tất cả các hệ thống mà không phải thay đổi gì, tiếc thay điều này là không thực tế. Nhưng đó sẽ là một sai phạm nếu ta để các đoạn mã nguồn không khả chuyển trong chương trình; một trong các vấn đề là do điều kiện biên dịch tạo ra.

Cục bộ hóa các hệ thống phụ thuộc vào các tập tin riêng biệt. Khi các hệ thống khác nhau cần mã nguồn khác nhau, thì sự khác nhau này nên để trong các tập tin riêng biệt, một tập tin cho mỗi hệ thống.

Che giấu các phụ thuộc hệ thống đằng sau các phương thức giao tiếp

Trừu tượng hóa là một kỹ thuật tốt để tạo ra các ranh giới giữa các phần mềm khả chuyển và không khả chuyển của một chương trình. Các thư viện nhập/xuất đi kèm các ngôn ngữ lập trình là một minh họa tốt; chúng thể hiện sự trừu tượng cho các thành phần lưu trữ thứ cấp trong các tập tin được mở, đóng, đọc và ghi, mà không cần phải tham chiếu tới cấu trúc và vị trí vật lý của chúng. Những chương trình tuân theo các phương thức giao tiếp sẽ chạy trên bất kỳ hệ thống nào hiện thực các phương thức giao tiếp này.

Hướng tiếp cận tính khả chuyển của Java là một ví dụ tốt. Một chương trình trình Java được dịch sang thi hành trong một "máy ảo", mô phỏng một máy tính và có thể thi hành để chạy trên bất kỳ máy thực nào. Các thư viện của Java cung cấp sự truy xuất thống nhất các đặc tính phía dưới của hệ thống như đồ họa, giao tiếp người dùng, mạng và nhiều thứ khác.

8.5. Chuyển đổi dữ liệu

Việc chuyển các dữ liệu văn bản từ hệ thống này sang hệ thống khác rất dễ và là cách đơn giản nhất để chuyển đổi các thông tin tùy ý giữa các hệ thống.

Việc chuyển đổi văn bản cũng còn có một rắc rối: các hệ thống PC dùng kí hiệu trở về đầu dòng `'\r'` và xuống dòng kế tiếp `'\n'` để ngắt dòng, trong khi các hệ thống Unix chỉ dùng newline. Xuống dòng mới là một khái niệm cho máy điện báo đánh chữ ngày xưa, nó có một thao tác trở về đầu dòng (CR) và một thao tác xuống dòng (LF) riêng cho việc xuống dòng kế tiếp.

Mặc dù các máy tính ngày nay đã khác chúng vẫn dùng khái niệm (CRLF) sự kết hợp của CR và LF cho mỗi dòng. Nếu trong tập tin không có các ký hiệu này thì nó được xem là một dòng dài. Dẫn đến việc đếm dòng và các kí tự có thể sai. Một số phần mềm tránh được vấn đề này nhưng đa số là không. Một điều may mắn là các giao thức mạng hiện đại tương thích với vấn đề này chẳng hạn HTTP cũng dùng CRLF để phân định dòng.

Nên dùng các phương thức giao tiếp chuẩn, chúng xử lý vấn đề CRLF trên bất kỳ hệ thống nào, trên các hệ thống PC chúng bỏ `\r` khi nhập dữ liệu và thêm vào khi xuất dữ liệu, còn trên các hệ thống Unix thì không tạo ra `\r` . Với các tập tin di chuyển giữa các hệ thống khác nhau thường xuyên thì cần phải có một chương trình đặc biệt để chuyển đổi.

Bài tập 8-1. Viết một chương trình bỏ các `\r` ra khỏi một tập tin. Viết một chương trình thứ hai thêm chúng vào bằng các thay thế các dòng mới với một CR và LF. (Làm sao bạn kiểm tra được các chương trình này?).

8.6. Nâng cấp và tính khả chuyển

Một trong các nguyên nhân gây ra vấn đề về tính khả chuyển là sự thay đổi của các phần mềm trong vòng đời của nó. Các thay đổi có thể xảy ra tại bất kỳ các phương thức giao tiếp trong hệ thống và gây ra sự không tương thích giữa phiên bản đã có của chương trình.

Thay đổi tên nếu bạn thay đổi đặc tả

Ví dụ ưu chuộng là việc thay đổi các thuộc tính của lệnh `echo` trong Unix, thông qua cách khởi động các thông số của nó:

```
% echo hello, world

hello, world

%
```

Tuy nhiên, lệnh `echo` là một phần chính của giao diện script trên Unix, và sự cần thiết phát sinh ra các định dạng kết quả trở nên quan trọng. Do đó lệnh `echo` được thay đổi để thông dịch các thông số của nó, giống như hàm `printf`:

```
% echo 'hello\nworld'

hello

world

%
```

Điểm mới này rất hữu ích, nhưng lại gây ra vấn đề về tính kha chuyển cho bất kì các giao diện script nào phụ thuộc vào lệnh `echo`, với ý định chỉ dùng để `echo` mà thôi. Cách xử sự của lệnh:

```
%echo $PATH
```

bây giờ phụ thuộc vào phiên bản nào của `echo`. Nếu trên dòng lệnh vô tình có các dấu sổ ngược (\), thường có trong DOS và Windows, nó có thể bị lệnh `echo` thông dịch nhưng là một lệnh đặc biệt. Sự khác biệt tương tự như kết quả từ hàm `printf(str)` và `printf("%s", str)` nếu chuỗi `str` có chứa kí hiệu phần trăm (`%`).

Chúng ta chỉ mới nói một phần nhỏ về lệnh `echo`, nhưng nó cũng chỉ ra vấn đề cơ bản: thay đổi các hệ thống có thể phát sinh ra các phiên bản khác nhau của phần mềm, vô tình gây ra vấn đề về tính khả chuyển. Và các vấn đề loại này rất khó giải quyết. Có thể giảm bớt rắc rối bằng cách đặt tên

khác nhau cho các phiên bản của echo. Ví dụ dễ thấy nhất là lệnh sum của Unix nó in ra kích thước và ký số của một tập tin. Nó dùng để kiểm tra các thông tin truyền đi có thành công hay không:

```
% sum file
52313 2 file
%
% copy file to other machine
%
% telnet othermachine
$
$ sum file
52313 2 file
$
```

Ta thấy kí số của tập tin giống nhau sau khi truyền đi, chứng tỏ bản chính và bản sao là một.

Khi hệ thống giãn nở, các phiên bản biến đổi, và một vài người đã quan sát thấy thuật toán dùng kí số là không hoàn hảo, do đó lệnh sum được thay đổi để dùng thuật toán tốt hơn. Một vài người khác cũng quan sát thấy điều tương tự và lại thay đổi để dùng thuật toán tốt hơn. Cứ thế mà hiện nay có rất nhiều phiên bản của lệnh sum, mỗi lệnh cho kết quả khác nhau. Chúng ta thử chép một tập tin sang nhiều máy khác nhau để xem kết quả của lệnh sum ra sao:

```
% sum file
52313 2 file
%
% copy file to machine 2
```

```
% copy file to machine 3

% telnet machine2

$

$ sum file

eaa0d468   713   file

$ telnet machine3

>

> sum file

62992  1  file

>
```

Phải chăng tập tin đã bị sai, hay là do các phiên bản của `sum` khác nhau? Có thể là cả hai.

Vậy lệnh `sum` thực sự là tai họa cho tính khả chuyển; một chương trình dự định giúp đỡ cho việc sao chép một phần mềm từ máy này sang máy khác đã có sự không tương thích về phiên bản, kết quả là dùng lệnh `sum` không còn có ý nghĩa gì.

Với nhiệm vụ đơn giản của nó, lệnh `sum` ban đầu là tốt. Việc thay đổi nó để tạo ra một chương trình tốt hơn, nhưng chẳng tốt hơn bao nhiêu, và vô tình đã gây ra không ít phiền toái. Vấn đề ở đây không phải do tính mở rộng mà là do sự không tương thích của chương trình khi dùng cùng tên.

Bảo trì tính tương thích với các chương trình và dữ liệu có sẵn

Khi một phiên bản mới của phần mềm chẳng hạn như xử lý văn bản được phát hành, thông thường có nhu cầu phải xử lý được các tập tin do phiên bản cũ tạo ra. Điều mong đợi là các đặc tính mới được thêm vào sẽ không gây ảnh hưởng xấu. Nhưng các phiên bản mới đôi khi lại gây lỗi khi xử lý các định dạng cũ. Người dùng các phiên bản mới, ngay cả khi họ không dùng đặc tính mới, không thể chia sẻ tập tin với người khác nếu họ

còn dùng phiên bản cũ, vậy là họ buộc phải nâng cấp. Dù là một sự thiếu sót hay chiến lược tiếp thị, thì đây cũng được coi là một kiểu thiết kế rất đáng tiếc.

Tương thích ngược là khả năng của một chương trình có thể chạy được với các phiên bản cũ của nó. Nếu bạn định thay đổi một chương trình thì phải chắc không gây tổn hại cho các phần mềm hay dữ liệu phụ thuộc vào nó. Lập tài liệu cho các thay đổi, và cung cấp các phương thức giao tiếp để nó có thể thích ứng với các đặc tả cũ. Quan trọng nhất là phải xem xét sự thay đổi đó có thực sự cải tiến chương trình so với cái giá phải trả cho tính tương thích không?

8.7. Tổng kết

Phấn đấu để viết mã nguồn có tính khả chuyển là một điều đáng làm, vì mất rất nhiều thời gian để thay đổi một chương trình để cho nó có thể chạy được trên một hệ thống khác, hay khi chính hệ thống mà chương trình đang chạy có sự thay đổi. Tất nhiên tính khả chuyển không tự nhiên mà có, mà đòi hỏi phải cẩn thận khi cài đặt cũng như phải có kiến thức về tính khả chuyển trên tất cả các hệ thống mà chương trình có khả năng được triển khai.

Có hai hướng tiếp cận tính khả chuyển mà chúng ta gọi là hướng hợp nhất và hướng giao nhau. Hướng tiếp cận hợp nhất yêu cầu viết các phiên bản có thể chạy được trên mỗi hệ thống, kết hợp các đoạn mã nguồn càng nhiều càng tốt theo cơ chế biên dịch có điều kiện. Nhược điểm là phải viết nhiều mã nguồn và mã nguồn thường phức tạp, khó cập nhật và kiểm tra.

Hướng tiếp cận giao nhau là viết nhiều mã nguồn có thể chạy mà không thay đổi gì trên các hệ thống càng nhiều càng tốt. Sự phụ thuộc bắt buộc của từng hệ thống có thể đưa vào một tập tin duy nhất, có chức năng như một phương thức giao tiếp giữa chương trình và hệ thống phía dưới. Hướng tiếp cận giao nhau cũng có nhược điểm là làm mất nhiều hiệu suất cũng như đặc trưng riêng của mỗi hệ thống, tuy nhiên xét về mặt lâu dài hướng tiếp cận này có nhiều lợi ích hơn.

Chương 9

KÝ HIỆU

Một ngôn ngữ lập trình theo đúng nghĩa của nó luôn nâng cao tính tiện dụng để giúp cho việc viết chương trình được dễ dàng. Điều này lý giải vì sao mà trong bộ công cụ lập trình của những người lập trình kinh nghiệm không chỉ có những ngôn ngữ lập trình đa dụng như C và các công cụ liên quan mà còn có các ngôn ngữ script, các bộ công cụ tạo giao diện có thể lập trình được và rất nhiều những ngôn ngữ lập trình ứng dụng đặc thù khác.

Sức mạnh của tập ký hiệu tốt thoát ra khỏi phương pháp lập trình truyền thống để đi vào các lĩnh vực giải quyết các bài toán đặc thù. Các biểu thức thông thường cho phép ta viết các định nghĩa cô đọng (đôi khi còn mang vẻ tối nghĩa) của các lớp chuỗi ký tự; HTML cho phép định nghĩa cách bố trí cho các tài liệu tương tác, thường sử dụng các chương trình được nhúng vào các ngôn ngữ khác chẳng hạn như JavaScript, PostScript để biểu diễn toàn bộ tài liệu, chẳng hạn như việc biểu diễn cuốn sách này. Các chương trình bảng tính và xử lý văn bản thường có các ngôn ngữ lập trình như Visual Basic để tính toán các biểu thức, truy xuất thông tin hoặc là điều khiển việc hiển thị.

Nếu bạn cảm thấy rằng mình phải viết quá nhiều mã nguồn để làm một công việc lặp đi lặp lại một cách tẻ nhạt hoặc gặp khó khăn trong việc biểu diễn một quy trình nào đó, thì có lẽ bạn đang sử dụng một ngôn ngữ không phù hợp. Nếu chưa có một ngôn ngữ phù hợp thì đó là cơ hội cho bạn tạo ra một ngôn ngữ riêng cho chính mình. Việc tạo ra một ngôn ngữ không nhất thiết là xây dựng một ngôn ngữ như Java; thông thường thì việc thay đổi các ký hiệu có thể giải quyết các vấn đề rắc rối. Chẳng hạn như, các hàm định dạng chuỗi trong họ hàm `printf` là các hàm điều khiển sự hiển thị các

giá trị một cách cô đọng và đầy đủ ý nghĩa.

Chương này sẽ trình bày cách thức giải quyết các vấn đề thông qua tập ký hiệu, và minh họa một số kỹ thuật có thể được sử dụng để cài đặt những ngôn ngữ mang tính đặc thù. Chúng ta sẽ được xem xét việc sử dụng một chương trình để viết một chương trình khác, cùng với cách sử dụng tập ký hiệu thường dùng.

9.1. Định dạng dữ liệu

Trong thực tế thường có một khoảng cách giữa những gì chúng ta muốn máy tính làm với những gì ta cần phải làm để hoàn tất công việc. Khoảng cách này càng hẹp càng tốt. Tập ký hiệu tốt sẽ giúp ta giải quyết các bài toán dễ dàng hơn. Đôi khi, tập ký hiệu tốt giúp ta có cách hiểu mới, cho phép ta giải quyết các bài toán khó, thậm chí còn đưa đến những khám phá mới.

Các ngôn ngữ nhỏ gọn thường có các tập ký hiệu chuyên biệt dành cho các lĩnh vực hẹp. Các ngôn ngữ này không chỉ cung cấp một giao diện tốt mà còn giúp tổ chức cách cài đặt chương trình. Xét câu lệnh sau:

```
printf("%d %6.2f %-10.10s\n", i, f, s);
```

Mỗi dấu `%` trong chuỗi ký hiệu định dạng đại diện cho một giá trị của một tham số của lệnh `printf`, tiếp theo là một số cờ tùy chọn và độ rộng các trường, ký tự kết thúc biểu diễn kiểu tham số cần xuất. Các ký hiệu này cô đọng, trực quan và dễ viết, việc cài đặt cũng rõ ràng. Các thay đổi trong C++ (`iostream`) và Java (`java.io`) dường như khó khăn hơn bởi vì chúng không cung cấp tập ký hiệu đặc biệt, mặc dù chúng cho phép mở rộng các kiểu dữ liệu do người dùng định nghĩa và hỗ trợ việc kiểm tra kiểu dữ liệu.

Một số cài đặt không chuẩn của hàm `printf` cho phép ta tự thêm vào các quy ước vào các tập chức năng có sẵn của nó. Ví dụ, một trình biên dịch có thể dùng `%L` để biểu diễn số hàng và tên tập tin; một hệ thống đồ họa có thể dùng `%P` để biểu diễn điểm và `%R` để biểu diễn hình chữ nhật. Chuỗi cô đọng các ký tự và số dùng để đọc chỉ số chứng khoán được trình bày

trong Chương 4 cũng dựa trên tinh thần đó, tập ký hiệu cô đọng để sắp xếp các kết hợp dữ liệu chứng khoán.

Ta có thể tổng hợp các ví dụ tương tự như thế trong C và C++. Giả sử ta cần chuyển các gói chứa những kết hợp của các kiểu dữ liệu khác nhau từ một hệ thống này sang hệ thống khác. Như đã thấy trong Chương 8, giải pháp rõ ràng nhất là chuyển thành một biểu diễn văn bản. Khi đó, với một giao thức mạng chuẩn, định dạng biểu diễn sẽ ở dạng nhị phân để tăng hiệu suất chuyển đổi. Làm sao chúng ta có thể viết các đoạn mã nguồn xử lý gói tin trở nên khả chuyển, hiệu quả và dễ dùng?

Hãy tưởng tượng rằng ta dự tính gửi các gói tin chứa 8-bit, 16-bit và 32-bit dữ liệu từ hệ thống này sang hệ thống khác. Trong ANSI C, ta có thể lưu trữ tối thiểu 8 bit dữ liệu kiểu `char` (kiểu ký tự), 16 bit dữ liệu `short` (số nguyên ngắn), 32 bit cho `long` (số nguyên dài), vì thế ta sẽ sử dụng những kiểu dữ liệu này để biểu diễn các giá trị. Sẽ có nhiều loại gói tin: gói tin loại 1 sẽ có 1-byte chỉ định, 2-byte đếm, 1-byte giá trị và 4-byte dữ liệu:

0×01	cnt_1	cnt_0	val	$data_3$	$data_2$	$data_1$	$data_0$

Gói tin loại 2 sẽ chứa 1 số nguyên ngắn và 2 từ dữ liệu số nguyên dài:

0×02	cnt_1	cnt_0	$dw1_3$	$dw1_2$	$dw1_1$	$dw1_0$	$dw2_3$	$dw2_2$	$dw2_1$	$dw2_0$

Một cách tiếp cận là viết các hàm nén và giải nén cho mỗi loại gói tin tương ứng:

```
int pack_type1(unsigned char *buf, unsigned short cout,
        unsigned char val, unsigned long data)
```

```
{
        unsigned char *bp;
        bp = buf;
        *bp++ = 0x01;
        *bp++ = count >> 8;
        *bp++ = count;
        *bp++ = val;
        *bp++ = data >> 24;
        *bp++ = data >>16;
        *bp++ = data >> 8;
*bp++ = data;
        return bp - buf;
}
```

Đối với giao thức thực tế, sẽ có hàng tá thủ tục, tất cả các biến thể liên quan đến cùng một chủ đề. Các thủ tục có thể được đơn giản hoá bằng cách sử dụng các macro hay các hàm để kiểm soát những kiểu dữ liệu cơ bản (số nguyên ngắn, số nguyên dài, v.v...), nhưng nếu như vậy thì các đoạn mã lặp đi lặp lại rất dễ bị sai, khó đọc và khó bảo trì.

Sự lặp đi lặp lại vốn có của các đoạn mã nguồn là một đầu mối mà các ký hiệu có thể có ích. Mượn ý tưởng từ hàm `printf`, chúng ta có thể định nghĩa một một ngôn ngữ đặc tả nhỏ cho phép mỗi gói tin được định nghĩa bằng một chuỗi ngắn gọn, nắm bắt định dạng (layout) của gói tin. Các thành phần kế tiếp của gói tin được mã hoá với `c` cho một ký tự 8-bit, `s` cho một số nguyên ngắn 16-bit và `l` cho một số nguyên dài 32-bit. Vì vậy, chẳng hạn như một gói tin loại 1 tạo ra từ ví dụ trên, bao gồm cả byte kiểu dữ liệu, sẽ được mô tả bởi chuỗi định dạng `cscl`. Sau đó, chúng ta có thể sử dụng

một hàm nén đơn để tạo ra các gói tin của bất kỳ loại nào; gói tin này sẽ được tạo ra với

```
pack(buf, "cscl", 0x01, count, val, data);
```

Do chuỗi định dạng dữ liệu chỉ bao gồm các định nghĩa dữ liệu cho nên không cần phải dùng các ký tự % như trong hàm `printf`.

Trong thực tế, các thông tin ở đầu gói tin sẽ thông báo cho phía nhận gói tin cách giải mã những phần còn lại, nhưng chúng ta giả sử rằng byte đầu tiên của gói tin có thể được dùng để xác định định dạng. Người gửi mã hoá dữ liệu theo định dạng này và gửi đi; người nhận đọc gói tin, lấy ra byte đầu tiên và dùng nó để giải mã những phần còn lại.

Sau đây là cài đặt của gói tin sẽ lấp đầy `buf` với các biểu diễn đã mã hoá của các tham số như đã xác định bằng định dạng trên. Chuyển tất cả các dữ liệu thành không dấu, bao gồm các byte trong bộ nhớ đệm gói tin, để tránh các vấn đề về dấu mở rộng (sign-extension). Ở đây cũng sử dụng một số kiểu định nghĩa dữ liệu quy ước để cho việc khai báo ngắn gọn hơn:

```
typedef unsigned char uchar;

typedef unsigned short ushort;

typedef unsigned long ulong;
```

Cũng giống như hàm `printf`, `strcpy` và các hàm tương tự khác, hàm nén (`pack`) giả sử rằng bộ nhớ đệm đủ lớn để lưu kết quả, và trách nhiệm của hàm thực hiện lời gọi là đảm bảo điều này. Không cần phải kiểm tra tính không hợp kiểu dữ liệu giữa định dạng và danh sách tham số.

```
#include <stdarg.h>

/* pack: nén các mục nhị phân vào buf, và trả về
chiều dài */

int pack(uchar *buf, char *fmt,…)

{
```

```
va_list args;
char *p;
uchar *bp;
ushort s;
ulong l;

bp = buf;
va_start(args, fmt);
for (p = fmt; *p != '\0'; p++) {
      switch (*p) {
      case 'c': /* char */
            *bp++ = va_arg(args, int);
            break;
      case 's': /* short */
            s = va_arg(args, int);
            *bp++ = s >> 8;
            *bp++ = s;
            break;
      case 'l': /* long */
            l = va_arg(args, ulong);
            *bp++ = l >> 24;
            *bp++ = l >> 16;
            *bp++ = l >> 8;
            *bp++ = l;
```

```
                break;
            default:  /* ký tự không hợp lệ */
                va_end(args);
                return -1;
            }
        }
        va_end(args);
        return bp - buf;
    }
```

Hàm `pack` dùng `stdarg.h` một cách mở rộng hơn hàm `eprintf` trong Chương 4. Các tham số trước đó được tách ra sử dùng macro `va_arg`, tham số đầu tiên có kiểu `va_list` thiết lập bằng cách gọi `va_start` và toán hạng tiếp theo là kiểu của tham số (đó là lý do tại sao `va_arg` phải là một macro, không phải là một hàm). Khi thực hiện xong, `va_end` phải được gọi. Mặc dù các tham số đối với `'c'` và `'s'` đại diện cho các giá trị kiểu `char` và `short` nhưng chúng cũng phải được tách ra thành các số nguyên `int` bởi vì C chuyển `char` và `short` thành `int` khi chúng được biểu diễn bằng một tham số có dạng tham số... (ba dấu chấm).

Mỗi thủ tục `pack_type` có chiều dài một dòng, sắp xếp các tham số thành một hàm gọi hàm `pack`:

```
    /* pack_type1: nén gói tin có kiểu định dạng 1 */
    int pack_type1 (uchar *buf, ushort count, uchar
val, ulong data)
    {
        return pack(buf, "cscl", 0x01, count, val ,
data);
    }
```

Để giải nén, chúng ta có thể làm giống như vậy: thay vì viết các đoạn mã nguồn riêng biệt để cắt mỗi định dạng gói tin, ta gọi một hàm `unpack` với một chuỗi định dạng. Điều này tập trung sự chuyển dổi vào một nơi:

```
/* unpack: giải nén các mục từ buf, trả về chiều dài */
    int unpack(uchar *buf, char *fmt, ...)
    {
        va_list args;
        char *p;
        uchar *bp, *pc;
        ushort  *ps;
        ulong  *pl;

        bp = buf;
        va_start(args, fmt);
        for  (p = fmt; *p != '\0'; p++) {
            swicht (*p) {
            case 'c': /* char  */
                pc = va_arg(args, uchar*);
                *pc = *bp++;
                break;
            case 's': /* short  */
                ps = va_arg(args, ushort*);
                *ps  = *bp++ << 8;
                *ps |= *bp++;
```

```
                break;
            case 'l':  /* long */
                pl = va_arg(args, ulong*);
                *pl  = *bp++ << 24;
                *pl |= *bp++ << 16;
                *pl |= *bp++ << 8;
                *pl |= *bp++;
                break;
            default:  /* ký tự không hợp lệ */
                va_end(args);
                return -1;
            }
        }
        va_end(args);
        return bp - buf;
    }
```

Giống như hàm `scanf`, hàm `unpack` phải trả về nhiều giá trị cho hàm gọi nó, do đó các tham số của nó là các con trỏ chỉ đến các biến lưu trữ các kết quả. Giá trị của hàm là số byte trong gói tin, có thể dùng giá trị này để kiểm tra lỗi.

Do các giá trị dữ liệu không mang dấu và do ANSI C định nghĩa kích thước kiểu dữ liệu cho nên đoạn mã nguồn này có thể chuyển đổi dữ liệu một cách khả chuyển và thậm chí giữa hai máy có kích thước dữ liệu `short` và `long` khác nhau. Nếu như chương trình sử dụng hàm `pack` không gửi một số `long` (chẳng hạn), một giá trị không thể biểu diễn bằng 32 bit, thì giá trị sẽ nhận được một cách chính xác. Kết quả là chúng ta chuyển được

32 bit thấp của giá trị đó. Nếu muốn những giá trị lớn hơn thì chúng ta phải định nghĩa một định dạng khác.

Các hàm giải nén theo từng kiểu dữ liệu gọi đến hàm `unpack` rất dễ viết:

```
/* unpack_type2: giải nén và xử lý gói tin loại 2
*/

int unpack_type2(int n, uchar *buf)

{

        uchar c;

        ushort count;

        ulong dw1, dw2;


        if (unpack(buf, "csll", &c, &count, &dw1,
&dw2) != n)

            return -1;

        assert(c == 0x02);

        return process_type2(count, dw1, dw2);

}
```

Để gọi hàm `unpack_type2`, trước tiên chúng ta phải biết rằng chúng ta có một gói tin loại 2, sử dụng một vòng lặp nhận như sau:

```
while ((n = readpacket(network, buf, BUFSIZ)) >
0) {

        switch (buf[0]) {

        default:
```

```
            eprintf("bad packet type 0x%x", buf[0]);
                break;
        case 1:
                unpack_type1(n, buf);
                break;
        case 2:
                unpack_type2(n, buf);
                break;
        ...
        }
    }
```

Phong cách lập trình này còn có thể phát triển hơn nữa. Một phương thức cô đọng hơn là định nghĩa một bảng các con trỏ hàm mà các đầu vào là các hàm giải nén được đánh chỉ mục theo loại:

```
int (*unpackfn[])(int, uchar *) = {
        unpack_type0,
        unpack_type1,
        unpack_type2,
};
```

Mỗi hàm trong bảng thực hiện phân tích một gói tin, kiểm tra kết quả trả về và khởi tạo các xử lý tiếp theo cho gói tin đó. Bảng này giúp cho công việc của người nhận tiến triển dễ dàng hơn:

```
/* receive: đọc các gói tin từ mạng, xử lý chúng */
void receive(int network)
{
```

```
        uchar type, buf[BUFSIZ];
        int n;
        while ((n = readpacket(network, buf, BUFSIZ)) >
0) {
        type = buf[0];
        if (type >= NELEMS(unpackfn))
              eprintf("bad packet type 0x%x", type);
                    if ((*unpackfn[type])(n, buf) <0)
                    eprintf ("protocol error, type %x
length %d", type, n);
              }
        }
```

Mỗi đoạn mã nguồn xử lý gói tin cô đọng, riêng biệt và dễ bảo trì. Người nhận hoàn toàn độc lập về gíao thức, đồng thời nó cũng rõ ràng và nhanh.

Ví dụ này dựa trên một số đoạn mã thực của một giao thức mạng thương mại. Khi tác giả nhận ra rằng hướng tiếp cận này có thể thực hiện được, vài ngàn hàng mã nguồn lặp đi lặp lại, dễ bị sai sót đã thu gọn thành vài trăm hàng dễ dàng bảo trì. Việc ký hiệu hoá (các ký hiệu) đã giảm rất nhiều những sự lộn xộn.

Bài tập 9-1. Hãy sửa hàm `pack` và `unpack` để chuyển các giá trị có dấu một cách đúng đắn, giữa các máy có kích thước `short` và `long` khác nhau. Bạn sẽ sửa các chuỗi định dạng như thế nào để đặc tả một mục dữ liệu có dấu? Làm thế nào để kiểm tra đoạn mã để xác định rằng nó chuyển đúng đắn `a-1` từ một máy tính kiểu long 32-bit sang một máy kiểu long 64-bit?

Bài tập 9-2. Hãy mở rộng hàm `pack` và `unpack` để kiểm soát dữ liệu kiểu chuỗi; một khả năng là bao gồm cả chiều dài của chuỗi trong chuỗi

định dạng. Hãy mở rộng chúng để kiểm soát các mục lặp đi lặp lại với một biến đếm. Làm thế nào để hàm này tương tác với mã hoá của các chuỗi?

Bài tập 9-3. Bảng các con trỏ hàm trong chương trình C nêu trên là lõi của các hàm ảo trong C++. Hãy viết lại hàm `pack` và `unpack` và `receive` trong C++ để ứng dụng ưu điểm của phương pháp ký hiệu vượt trội này.

Bài tập 9-4. Hãy viết một phiên bản dòng lệnh (command-line) của hàm `printf` cho phép in các tham số thứ 2 và các tham số tiếp theo của nó theo định dạng đưa ra bởi tham số thứ nhất. Một số shells đã cung cấp điều này như là một phần dựng sẵn.

Bài tập 9-5. Hãy viết một hàm cài đặt các đặc tả định dạng thường thấy trong các chương trình bảng tính hay trong lớp `DecimalFormat` của Java, cho phép hiển thị các con số theo những mẫu chỉ định số nào là bắt buộc hoặc tuỳ chọn, vị trí của dấu chấm thập phân và dấu phẩy, v.v... Ví dụ, chuỗi định dạng:

```
##,##0.00
```

chỉ ra rằng một số có 2 số lẻ, ít nhất một con số bên trái của dấu chấm thập phân, một dấu phẩy sau con số hàng ngàn, và một điền vào khoảng trắng cho đến hàng chục ngàn. Nó sẽ biểu diễn số 12345.67 thành `12,345.67` và .4 thành _ _ _ _ _`0.40` (sử dụng dấu gạch dưới để đại diện cho khoảng trắng). Xem thêm định nghĩa của `DecimalFormat` hoặc các chương trình bảng tính để biết thêm chi tiết về đặc tả.

9.2. Các biểu thức có quy tắc

Các chỉ dẫn định dạng của hàm `pack` và `unpack` là những ký hiệu rất đơn giản dùng để định nghĩa cách thức trình bày của các gói tin. Chủ đề tiếp theo của chúng ta phức tạp hơn một chút nhưng là những ký hiệu dễ hiểu, *các biểu thức có quy tắc*, cho phép xác định mẫu của đoạn văn bản. Chúng tôi thường sử dụng các biểu thức có quy tắc trong suốt cuốn sách mà không cần phải giải thích nhiều. Mặc dù các biểu thức có quy tắc phổ biến trong môi trường lập trình Unix nhưng chúng không được sử dụng rộng rãi trong

các hệ thống khác, vì vậy phần này chúng tôi chỉ minh họa một số điểm mạnh của chúng. Trong trường hợp bạn không có sẵn thư viện biểu thức có quy tắc trong tay thì chúng tôi cũng sẽ đưa ra một cài đặt thô sơ.

Có một số đặc trưng của các biểu thức có quy tắc, nhưng nhìn chung thì chúng giống nhau, một cách mô tả các mẫu gồm các ký tự cơ bản, cùng với việc lặp lại, thay thế và viết tắt cho các lớp của các ký tự như số và chữ. Một ví dụ tương tự được gọi là "ký tự đại diện" (wildcard) sử dụng trong các trình xử lý dòng lệnh để tìm các mẫu của các tập tin. Thường thì `*` có nghĩa là "bất kỳ chuỗi ký tự nào", do đó ví dụ như một dòng lệnh

```
C:\> del *.exe
```

dùng một mẫu khớp với tất cả các tập tin có tên là bất kỳ chuỗi ký tự nào kết thúc bằng "`.exe`". Thông thường thì các chi tiết khác nhau trong các hệ thống khác nhau, và thậm chí trong các chương trình khác nhau cũng khác nhau.

Mặc dù những tính thất thường của các chương trình khác nhau có thể gợi ý rằng các biểu thức có quy tắc là một kỹ thuật phức tạp, nhưng thực tế chúng là một ngôn ngữ có ngữ pháp hình thức và ngữ nghĩa chính xác nhờ cách phát âm trong ngôn ngữ này. Hơn nữa, những cài đặt đúng sẽ chạy rất nhanh; một sự kết hợp giữa lý thuyết và kiến trúc thực tế sẽ tạo ra nhiều sự khác biệt, một ví dụ của ưu điểm của các thuật toán chuyên biệt mà chúng tôi đã đề cập đến trong Chương 2.

Một biểu thức có quy tắc là chuỗi các ký tự định nghĩa một tập hợp các chuỗi dùng cho việc tìm kiếm. Hầu hết các ký tự chỉ tương đương với chúng mà thôi, vì vậy biểu thức có quy tắc `abc` sẽ tương đương với chuỗi các ký tự xuất hiện bất cứ ở đâu. Thêm vào đó, một số siêu ký tự (*metacharacter*) chỉ định sự lặp lại, nhóm hay vị trí. Trong quy ước biểu thức có quy tắc trên Unix, `^` đại diện cho việc bắt đầu một chuỗi và `$` kết thúc chuỗi, do đó `^x` tương đương với một chuỗi chỉ có `x` ở đầu, `x$` tương đương với một chuỗi chỉ có 1 `x` ở cuối, `^x$` tương đương với chuỗi chỉ có `x` nếu nó là ký tự duy nhất trong chuỗi và `^$` tương đương với 1 chuỗi rỗng.

Ký tự "." tương đương với bất kỳ ký tự nào, do đó `x.y` tương đương `xay`, `x2y`, v.v..., nhưng không phải là `xy` hay `xaby`, và `^.$` tương đương với một chuỗi chỉ có một ký tự bất kỳ.

Một tập hợp các ký tự trong dấu `[]` tương đương với bất kỳ một ký tự nào trong tập hợp đó, do đó `[0123456789]` tương đương với một số; nó có thể được viết tắt là `[0-9]`.

Những khối này được phối hợp với dấu ngoặc đơn để biểu diễn nhóm, | để biểu diễn thay thế, * để biểu diễn không hoặc nhiều thể hiện, + để biểu diễn một hoặc nhiều thể hiện, và ? để biểu diễn không hoặc một thể hiện. Cuối cùng, \ được dùng như một tiếp đầu ngữ để khai báo một siêu ký tự và tắt ý nghĩa đặc biệt của nó; * là một từ khóa * và \\ là một từ khóa backslash.

Công cụ biểu thức có quy tắc nổi tiếng nhất là chương trình `grep` mà đã được đề cập đến một vài lần. Chương trình này là một ví dụ tuyệt vời về giá trị của ký hiệu. Nó áp dụng một biểu thức có quy tắc cho mỗi dòng trong tập tin đầu vào và in những hàng này ra chứa các chuỗi tương ứng. Đặc tả đơn giản này, cùng với những điểm mạnh của các biểu thức có quy tắc, cho phép ta giải quyết nhiều công việc phải làm từ ngày này sang ngày khác. Trong ví dụ sau, hãy lưu ý là cú pháp của biểu thức có quy tắc được dùng trong tham số của `grep` khác với các ký tự đại diện dùng để chỉ định một tập hợp các tập tin; sự khác biệt này phản ánh những cách dùng khác nhau.

Tập tin nguồn nào sử dụng lớp `Regexp`?

```
% grep Regexp *.java
```

Cái gì cài đặt nó?

```
% grep 'class.*Regexp' *.java
```

Tôi đã lưu thư của Bob ở đâu?

```
% grep '^From:.*bob@' mail/*
```

Có bao nhiêu dòng mã nguồn khác rỗng trong chương trình này?

```
% grep '.' *.c++ | wc
```

Với các cờ để in thứ tự dòng của các dòng tìm thấy, số lượng kết quả tìm thấy, thực hiện tìm kiếm có phân biệt chữ hoa chữ thường, đảo ngược kết quả tìm kiếm được (chọn các dòng không khớp chuỗi cần tìm) và thực hiện các phép biến đổi khác dựa trên ý tưởng cơ bản, `grep` được dùng phổ biến đến nỗi nó đã trở thành một ví dụ cổ điển của việc lập trình dựa trên các công cụ.

Thật không may, không phải hệ thống nào cũng có `grep` hay chương trình tương tự như vậy đi kèm. Một số hệ thống có một thư viện các biểu thức có quy tắc, thường được gọi là `regex` hay `regexp`, cho phép bạn dùng để viết một phiên bản của `grep`. Nếu cả hai thứ đó đều không có thì cũng không khó khăn gì để cài đặt một tập con vừa phải của ngôn ngữ biểu thức có quy tắc hoàn chỉnh. Ở đây, chúng tôi giới thiệu một cài đặt của các biểu thức có quy tắc và `grep` đi cùng; để đơn giản chỉ có các siêu ký tự là `^$.` và `*`, trong đó `*` chỉ sự lặp đi lặp lại của một dấu chấm trước đó hay ký tự cơ bản. Tập con này cung cấp rất nhiều điểm mạnh với chỉ một ít độ phức tạp của lập trình nhờ vào các biểu thức tổng quát.

Hãy bắt đầu với chính hàm tìm kiếm chuỗi ký tự. Đó là xác định xem một chuỗi ký tự có chứa một biểu thức cần tìm hay không:

```
/* match:  tìm kiếm regexp ở bất kỳ vị trí nào
trong văn bản */

int match(char *regexp, char *text)

{

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

                return matchhere(regexp+1, text);

        do {      /* phải xét cả trường hợp chuỗi
rỗng */

                if (matchhere(regexp, text))
```

```
                    return 1;
            } while (*text++ != '\0');
            return 0;
    }
```

Nếu biểu thức cần tìm bắt đầu với ^ thì chuỗi ký tự phải bắt đầu với một tương ứng của những phần còn lại trong biểu thức. Ngược lại, chúng ta sẽ duyệt tiếp chuỗi ký tự, sử dụng hàm `matchhere` để kiểm tra xem chuỗi ký tự có được tìm thấy ở bất kỳ vị trí nào hay không. Khi chúng ta tìm thấy một kết quả, chúng ta đã hoàn tất. Hãy lưu ý cách dùng vòng lặp `do-while`: các biểu thức có thể phù hợp với chuỗi rỗng (ví dụ như `$` sánh với với chuỗi rỗng ở cuối dòng và `.*` tương ứng với bất kỳ số ký tự nào, kể cả không có ký tự nào), do đó chúng ta phải gọi hàm `matchhere` ngay cả khi chuỗi rỗng.

Hàm đệ quy `matchhere` thực hiện hầu hết các công việc:

```
    /* matchhere:  tìm kiếm regexp từ vị trí đầu tiên
của văn bản */
    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;
    }
```

Nếu biểu thức `regexp` có giá trị rỗng thì việc tìm kiếm sẽ kết thúc và do đó chúng ta đã tìm được kết quả. Nếu biểu thức kết thúc với ký tự `$` thì kết quả sẽ chỉ được tìm thấy ở cuối văn bản. Nếu biểu thức bắt đầu bằng dấu chấm thì sẽ tìm kiếm bất kỳ ký tự nào. Ngược lại, biểu thức bắt đầu với ký tự thông thường sẽ thực hiện tìm kiếm chính nó trong phần văn bản đó. Ký tự `^` hoặc `$` nếu xuất hiện ở giữa biểu thức tìm kiếm sẽ được xem như một ký tự thông thường khác, chứ không được xem như ký tự đặc biệt.

Nên chú ý rằng các lời gọi `matchhere` sẽ gọi lại chính bản thân chúng sau khi tìm được một ký tự của mẫu và chuỗi, vì vậy số lần thực hiện của phép đệ quy sẽ tương ứng với chiều dài của mẫu.

Việc tìm kiếm sẽ gặp khó khăn nếu biểu thức bắt đầu với ký tự thay thế với dấu sao, chẳng hạn `x*`. Ta gọi hàm `matchstar` với đối số đầu tiên là toán hạng của ký tự thay thế dấu sao `(x)`, và những đối số kế tiếp là mẫu đứng sau dấu sao và phần văn bản.

```
    /* matchstar: tìm kiếm c*regexp từ vị trí đầu
tiên của văn bản */
    int matchstar(int c, char *regexp, char *text)
    {
        do { /* dấu * không hoặc thay thế nhiều ký
tự */
            if (matchhere(regexp, text))
                return 1;
        } while (*text != '\0' && (*text++ == c ||
c == '.'));
```

```
                return 0;
        }
```

Một vòng lặp `do-while` khác một lần nữa được kích hoạt với yêu cầu là biểu thức tìm kiếm `x*` có thể tương ứng với ký tự kết thúc chuỗi. Vòng lặp sẽ thực hiện kiểm tra xem văn bản có tương ứng với phần biểu thức còn lại hay không, thực hiện kiểm tra tại mỗi vị trí của phần văn bản miễn là ký tự đầu tiên tương ứng với toán hạng của dấu sao.

Phải thừa nhận đây là việc cài đặt không phức tạp nhưng vẫn hoạt động tốt, và với ít hơn 30 dòng mã nguồn, nó đã cho thấy các biểu thức tìm kiếm không cần phải sử dụng những kỹ thuật tiên tiến.

Chúng tôi sẽ bàn đến việc mở rộng chương trình này sau. Bây giờ, chúng ta sẽ viết một phiên bản của `grep` có sử dụng hàm `match`. Và đây là thủ tục chính:

```
/* grep main: tìm kiếm regexp trong các tập tin */
        int main(int argc, char *argv[])
        {
                int i, nmatch;
                FILE *f;

                setprogname("grep");
                if (argc < 2)
                    eprintf("usage: grep regexp [file …]");
                nmatch = 0;
                if (argc == 2) {
                      if (grep(argv[1], stdin, NULL))
                      nmatch++;
```

```
        } else {
        for (i = 2; i < argc; i++) {
           f = fopen(argv[i], "r");
           if (f  == NULL) {
              weprintf ("can't open %s:", argv[i]);
                 continue;
              }
               if  (grep(argv[1],  f,  argc>3  ?
argv[i] : NULL) >0)
                  nmatch ;
              fclose(f);
           }
        }
        return nmatch  == 0;
    }
```

Thông thường, các chương trình C trả về giá trị 0 nếu thành công và trả về giá trị khác không cho các loại lỗi khác nhau. Chương trình `grep` này sẽ cho biết quá trình tìm kiếm có thành công hay không trong quá trình tìm kiếm. Vì vậy, sẽ trả về giá trị 0 nếu không tìm thấy kết quả cần tìm, trả về 1 nếu không tìm thấy, và trả về giá trị 2 (thông qua hàm `eprintf`) nếu có lỗi xảy ra. Các chương trình khác có thể kiểm tra các giá trị trạng thái này.

Hàm `grep` duyệt qua từng dòng và thực hiện hàm `match` trên mỗi dòng:

```
/* grep: tìm kiếm regexp trong tập tin */
int grep(char *regexp, FILE *f , char *name)
```

```
        int n, nmatch;
        char buf[BUFSIZ];
        nmatch = 0;
        while (fgets(buf, sizeof buf, f) != NULL) {
            n = strlen(buf);
            if (n>0 && buf[n-1] == '\n')
                buf[n-1] = '\0';
            if (match(regexp, buf)){
                nmatch++;
                if (name != NULL)
                    printf ("%s:", name);
                printf("%s\n", buf);
            }
        }
        return nmatch;
    }
```

Thủ tục chính sẽ không kết thúc được nếu như không mở được tập tin. Cách thiết kế này được lựa chọn vì ta thường dùng lệnh như sau:

```
% grep herpolhode *.*
```

và nhận ra rằng một trong số các tập tin thuộc thư mục đó không đọc được. Tốt hơn hết là nên thiết kế hàm `grep` sao cho nó vẫn tiếp tục thực hiện công việc sau khi thông báo lỗi xảy ra. Hàm này không nên kết thúc giữa chừng và buộc người sử dụng gõ vào cả một danh sách tập tin một cách thủ công

chỉ vì muốn tránh tập tin gây ra lỗi. Tương tự, cần lưu ý rằng thủ tục `grep` in ra tên tập tin và dòng văn bản tìm kiếm nếu nó đang đọc từ dữ liệu nhập chuẩn hoặc một tập tin. Cách thiết kế này có vẻ hơi lạ nhưng nó phản ánh phong cách sử dụng thường thấy dựa trên kinh nghiệm. Khi chỉ cho biết một dữ liệu đầu vào duy nhất, nhiệm vụ của `grep` thường là phải chọn lựa và tên các tập tin sẽ làm lộn xộn dữ liệu xuất. Nhưng nếu như yêu cầu `grep` tìm kiếm trên nhiều tập tin, công việc thường gặp nhất là tìm kiếm tất cả những lần xuất hiện với thông tin đầy đủ về điều cần tìm.

Hãy so sánh:

```
% strings markov.exe | grep 'DOS mode'
```

với

```
% grep grammer chapter*.txt
```

Đây chính là lý do vì sao `grep` lại trở nên thông dụng, và cho thấy rằng ký hiệu phải đi kèm với tính tự nhiên để có thể xây dựng được một công cụ hiệu quả.

Hàm `match` sẽ kết thúc và trả về giá trị ngay khi tìm thấy kết quả. Nhưng để cài đặt chức năng thay thế (tìm kiếm và thay thế) trong một trình soạn thảo văn bản, kết quả tìm kiếm *dài nhất tận cùng bên trái* sẽ càng thích hợp hơn. Chẳng hạn như cho trước đoạn văn bản `"aaaaa"`, mẫu `a*` sẽ tương xứng với chuỗi rỗng ngay từ vị trí đầu tiên, nhưng sẽ tự nhiên hơn nếu xem như tương ứng với cả năm chữ `a`. Để hàm `match` tìm được chuỗi ký tự dài nhất tận cùng bên trái, ta phải viết lại hàm `matchstar` để thực hiện được nhiều hơn: thay vì chỉ tìm tại mỗi ký tự từ trái sang phải, nó cần bỏ qua chuỗi dài nhất tương ứng với toán hạng của dấu sao, sau đó duyệt ngược trở lại nếu phần còn lại của chuỗi không tương ứng với phần còn lại của mẫu. Nói cách khác là sẽ thực hiện duyệt từ phải sang trái. Sau đây là một phiên bản của hàm `matchstar` thực hiện việc tìm kiếm dài nhất tận cùng bên trái:

```
/* matchstar: tìm kiếm dài nhất tận cùng bên trái cho
c*regexp */
```

```
int matchstar(int c, char *regexp, char *text)
{
        char *t;

        for (t = text; *t != '\0' && (*t == c || c
== '.'); t++)                    ;
        do {  /* dấu * không hoặc thay thế nhiều ký
tự */
            if (matchhere(regexp, t))
                return 1;
        }while (t-- > text);
        return 0;
}
```

Không cần quan tâm đến kết quả tương xứng mà `grep` tìm được vì nó chỉ kiểm tra sự hiện diện của bất kỳ một kết quả tìm kiếm và in ra nguyên cả dòng đó. Do vậy, vì việc tìm kiếm chuỗi dài nhất tận cùng bên trái chỉ là một công việc phụ nên không cần thiết cho hàm `grep`, nhưng đối với việc thay thế chuỗi thì nó lại cần thiết.

Hàm `grep` cũng gặp khó khăn với những phiên bản được hệ thống hỗ trợ. Một số biểu thức rắc rối sẽ gây nên việc thực hiện có tính lũy thừa, chẳng hạn `a*a*a*a*a*b` khi cho dữ liệu đầu vào là `aaaaaaaaac`, nhưng việc thực hiện có tính lũy thừa này cũng xuất hiện trong một số chương trình thương mại. Một biến thể khác của `grep` hiện có sẵn trong Unix được gọi là `egrep` đã sử dụng thuật toán so sánh phức tạp hơn nhiều. Thuật toán này đảm bảo việc thực hiện mang tính chất tuyến tính bằng cách tránh việc quay lui khi việc tìm kiếm từng phần thất bại.

Vậy còn việc hàm `match` xử lý toàn bộ các biểu thức tìm kiếm thì

sao? Điều đó bao gồm các nhóm ký tự như [a-zA-Z] để tìm ký tự chữ cái, khả năng nhận biết ký tự đặc biệt (chẳng hạn như tìm kiếm một dấu chấm), các dấu ngoặc dùng cho việc gom nhóm, và các lựa chọn như (abc hoặc def). Bước đầu tiên là giúp hàm match biên dịch mẫu này thành cách biểu diễn sao cho dễ dàng phân tích hơn. Việc phân tích loại ký tự mỗi khi chúng ta so sánh chúng với một ký tự sẽ không tốt; trong khi đó một cách biểu diễn có tính toán trước dựa trên các vector theo bit sẽ làm cho các loại ký tự trở nên hiệu quả hơn. Với các biểu thức thông thường đầy đủ với các dấu ngoặc và các lựa chọn, việc cài đặt cần phải phức tạp hơn, nhưng vẫn có thể sử dụng một vài kỹ thuật mà chúng ta đã đề cập trong phần sau của chương này.

Bài tập 9-6. Hãy so sánh việc thực hiện của hàm match với hàm strstr khi thực hiện tìm kiếm trên văn bản thường.

Bài tập 9-7. Hãy viết một phiên bản hàm matchhere khác không đệ quy và so sánh việc thực hiện của nó với phiên bản có đệ quy.

Bài tập 9-8. Thêm một số tùy chọn cho hàm grep: -v dùng để thực hiện tìm kiếm đảo ngược, -i để thực hiện so sánh chữ cái không phân biệt chữ hoa và chữ thường, và -n để cho biết chỉ số dòng trong kết quả xuất. Cách thức các chỉ số dòng được in ra như thế nào? Có nên in chúng ra trên cùng dòng với văn bản cần tìm.

Bài tập 9-9. Thêm các toán tử + (một hoặc nhiều) và ? (không hoặc một) để tìm kiếm. Mẫu a+bb? sẽ tìm một hoặc nhiều chữ a và theo sau là một hoặc hai chữ b.

Bài tập 9-10. Việc cài đặt hiện thời của hàm match sẽ làm cho ý nghĩa đặc biệt của ^ và $ không còn nữa nếu chúng không bắt đầu hoặc kết thúc biểu thức, và của dấu * nếu nó không đi theo ngay sau một ký tự thông thường hoặc một dấu chấm. Một cách thiết kế thông thường là đánh dấu các ký tự đặc biệt bằng cách thêm vào trước nó dấu \. Sửa lại hàm match để xử lý các dấu \ theo cách này.

Bài tập 9-11. Hãy thêm các loại ký tự vào hàm match. Các loại ký

tự xác định một sự tương ứng cho bất kỳ ký tự nào xuất hiện trong ngoặc. Việc thêm vào phạm vi xem xét sẽ giúp chương trình trở nên tiện lợi hơn, chẳng hạn `[a-z]` dùng để xét các ký tự chữ thường, và thay đổi ý nghĩa, chẳng hạn `[^0-9]` dùng để thực hiện tìm bất kỳ ký tự nào *ngoại trừ* chữ số.

Bài tập 9-12. Thay đổi hàm `match` để sử dụng phiên bản `matchstar` để tìm chuỗi dài nhất tận cùng bên trái, và sửa đổi chúng để có thể trả về các vị trí ký tự đầu tiên và cuối cùng của chuỗi tìm được. Dùng chương trình này để xây dựng một chương trình `gres` tương tự với `grep` nhưng in tất cả các dòng dữ liệu đầu vào sau khi thay thế chuỗi cần tìm trong mẫu bằng chuỗi mới, như sau:

```
% gres 'homoiousian' 'homoousian' mission.stmt
```

Bài tập 9-13. Hãy chỉnh đổi `match` và `grep` để chúng có thể thực thi được với các chuỗi UTF-8 chứa các ký tự Unicode. Vì UTF-8 và Unicode là tập con của bảng mã ASCII, sự thay đổi này mang tính tương thích tiến. Các biểu thức cần tìm cũng như văn bản gốc đều cần thực thi tốt với UTF-8. Vậy thì các loại ký tự nên được cài đặt như thế nào?

Bài tập 9-14. Hãy viết chương trình kiểm chứng tự động cho các biểu thức cần tìm. Chương trình này sẽ phát sinh các biểu thức kiểm chứng và các chuỗi kiểm chứng để tìm kiếm. Nếu có thể, bạn hãy dùng các thư viện có sẵn để tham khảo cách thức cài đặt; cũng có thể bạn sẽ tìm được các lỗi trong những thư viện đó.

9.3. Một số công cụ lập trình

Nhiều công cụ được cấu trúc dựa vào một *ngôn ngữ dành cho mục đích đặc biệt*. Chương trình `grep` chỉ là một trong số các họ công cụ sử dụng các biểu thức thông thường hoặc sử dụng các ngôn ngữ khác để giải quyết các bài toán lập trình.

Một trong những ví dụ đầu tiên là trình thông dịch lệnh hay ngôn ngữ điều khiển công việc. Trình tự các câu lệnh có thể được ghi vào một tập tin đã được đưa vào áp dụng rất sớm, và một phiên bản của trình thông dịch

hay *shell* có thể thi hành các lệnh trong tập tin đó. Từ đó, chỉ cần thêm các tham số, điều kiện, vòng lặp, biến, và tất cả các trường hợp khác trong một ngôn ngữ lập trình truyền thống. Điểm khác nhau chủ yếu là ở đây chỉ có một kiểu dữ liệu - đó là `string` - và các phép toán trong các chương trình *shell* có khuynh hướng là các chương trình hoàn chỉnh thực hiện các tính toán thú vị. Mặc dù lập trình *shell* không còn được ưa chuộng, thường thì chúng được chuyển sang các lựa chọn khác như Perl trong các môi trường lập trình lệnh và các nút nhấn trong các giao diện người dùng dạng đồ họa, nhưng nó vẫn là một cách hữu hiệu để xây dựng các điều khiển phức tạp.

Awk là một công cụ khả lập trình khác, nó là một ngôn ngữ nhỏ, chuyên về các thao tác mẫu (specialized pattern-action); nó tập trung vào việc lọc và biến đổi một luồng dữ liệu nhập. Như chúng ta đã thấy trong Chương 3, Awk thực hiện tự động việc đọc các tập tin dữ liệu nhập và cắt mỗi dòng ra thành các trường được đặt tên là `$1` tới `#NF`, trong đó `NF` là số trường trên một dòng. Bằng cách hỗ trợ hành vi mặc định (default behavior) cho nhiều tác vụ thông thường, chúng ta có thể tạo ra được các chương trình chỉ có một dòng. Ví dụ, sau đây là một chương trình Awk hoàn chỉnh:

```
        # split.awk: chia dữ liệu nhập thành mỗi từ trên
một dòng

        { for (i = 1; i <= NF; i++) print $i }
```

chương trình trên in ra các từ của mỗi dòng dữ liệu nhập, một từ trên một dòng. Để chuyển sang vấn đề khác, chúng ta hãy xem xét một cài đặt của hàm `fmt` sau đây, lấp mỗi dòng dữ liệu xuất bằng các từ, tối đa 60 ký tự; dòng trắng là nơi ngắt đoạn:

```
        # fmt.awk: format into 60-character lines

        /./    { for (i = 1; i <=NF; i++) addword($i) }
#nonblank line

        /^$/  { printline(); print "" }                  #
blank line
```

```
END   { printline() }


function addword(w) {
        if (length(line) + 1 + length(w) > 60)
                printline()
        if (length(line) == 0)
                line = w
        else
                line = line " " w
}
function printline() {
        if (length(line) > 0) {
                print line
                line = ""
        }
}
```

Chúng ta thường dùng hàm `fmt` để chia đoạn lại các dòng tin trong email và các tài liệu ngắn khác, chúng ta cũng dùng nó để định dạng dữ liệu xuất của các chương trình *Markov* trong Chương 3.

Các công cụ lập trình thường có nguồn gốc từ các ngôn ngữ nhỏ được thiết kế cho việc diễn đạt tự nhiên các giải pháp cho các vấn đề tồn tại trong một phạm vi hẹp. Một ví dụ dễ thấy là công cụ `eqn` của Unix, nó thiết lập công thức toán học. Ngôn ngữ dữ liệu nhập của nó gần với cách đọc các phép toán trong các công thức, ví dụ `π/2` được viết là "`pi over 2`". TEX cũng theo cách tiếp cận này; ký hiệu của nó cho công thức này là `\pi\over 2`. Nếu có các ký hiệu tự nhiên hay quen thuộc cho vấn đề bạn đang giải

quyết thì hãy dùng nó và hiệu chỉnh nó cho thích hợp chứ đừng thực hiện từ đầu.

Awk bị thuyết phục bởi một chương trình sử dụng các biểu thức có quy tắc để chỉ ra các dữ liệu khác thường trong các hồ sơ theo dõi điện thoại, tuy nhiên Awk thêm vào các biến, diễn đạt, vòng lặp,... để làm cho nó trở thành một ngôn ngữ lập trình thực thụ. Perl và Tcl ngay từ đầu đã được xây dựng cho việc kết hợp sự tiện lợi và ấn tượng của các ngôn ngữ nhỏ với sức mạnh của các ngôn ngữ lớn. Chúng thật sự là các ngôn ngữ cho mục đích tổng quát, mặc dù chúng rất hay được dùng cho việc xử lý văn bản.

Thuật ngữ dùng chung cho các công cụ như thế là *ngôn ngữ kịch bản* (scripting language), bởi vì chúng được phát triển từ các trình thông dịch lệnh đầu tiên có khả năng hỗ trợ lập trình được giới hạn trong khuôn khổ chạy các "kịch bản" được gắn vào các chương trình. Các ngôn ngữ kịch bản cho phép cách dùng sáng tạo các biểu thức có quy tắc, không những cho việc xác định các mẫu hợp lệ - nhận biết sự xuất hiện của một mẫu nào đó – mà còn cho việc định vị các vùng văn bản sẽ được chuyển đổi dạng thức. Điều này xuất hiện trong hai lệnh `regsub` (REGular expression SUBstitution) trong chương trình Tcl dưới đây. Chương trình này là một sự tổng quát hoá không đáng kể của chương trình mà chúng ta đã đưa ra trong Chương 4 được dùng để lấy các thành phần trong các dấu nháy; chương trình này được dùng để lấy URL được lưu trong đối số đầu tiên của nó. Sự thay thế đầu tiên sẽ loại bỏ chuỗi `http://` nếu có; sự thay thế thứ hai sẽ thay thế dấu `/` đầu tiên bằng một khoảng trắng, kết quả là chia đối số ra thành hai trường. Lệnh `lindex` lấy các trường từ một chuỗi (bắt đầu với chỉ số 0). Văn bản bên trong `[]` được thi hành như một lệnh Tcl và được thay thế bởi văn bản kết quả; `$x` được thay thế bởi giá trị của biến `x`.

```
# geturl.tcl: retrieve document from URL

# input has form [http://]abc.def.com[/whatever...]

regsub "http://" $argv "" argv      ;#      remove
```

```
http:// if present
        regsub "/" $argv " " argv            ;#    replace
leading / with blank


        set  so  [socket  [lindex  $argv  0]  80]  ;#  make
network connection
        set q "/[lindex $argv 1]"


        puts $so "GET $q HTTP/1.0\n\n"       ;#send
request
        flush $so
        while {[gets $so line] >= 0 && $line != ""} {}
;#skip header
        puts [read $so]                       ;#read    and
print entire reply
```

Script này tạo kết quả đầu ra rất dài, phần lớn là các tag của HTML được đóng trong hai dấu < và >. Perl là một ngôn ngữ rất mạnh trong việc thay thế chuỗi, do đó công cụ tiếp theo của chúng ta là một script của Perl sử dụng các biểu thức cần tìm và các sự thay thế để loại bỏ các tag này:

```
        # unhtml.pl: delete HTML tags


        while (<>) {             # collect all input into
single string
                $str .= $_;      # by concatenating input
lines
        }
```

```
$str =~ s/<[^>]*>//g;   # delete <…>
$str =~ s/ / /g;   # replace   by blank
$str =~ s/\s+/\n/g;     # compress white space
print $str;
```

Ví dụ này sẽ khó hiểu đối với những người không đọc được chương trình Perl. Cấu trúc

```
$str = ~s/regexp/repl/g
```

có chức năng thay văn bản hợp với (có đầu bên trái dài nhất) các biểu thức có quy tắc `regexp` trong chuỗi `str` bằng chuỗi `repl`; chữ `g` ở cuối là viết tắt của "global", nghĩa là thực hiện thay thế đối với tất cả các văn bản phù hợp trong chuỗi chứ không phải chỉ cho văn bản đầu tiên. Ký tự đặc biệt `\s` là dạng viết tắt cho một loại ký tự trắng (khoảng trắng, tab, sang dòng,...); `\n` là ký tự sang dòng. Chuỗi "` `" là một ký tự HTML, tương tự như các ký tự trong Chương 2, nó định nghĩa một ký tự trắng không thể tách ra (non-breakable space character).

Sắp xếp tất cả chúng lại với nhau, ta được một trình duyệt web theo hướng chức năng được cài đặt như một script shell một dòng, mặc dù không mạnh:

```
# web: retrieve web page and format its text, ignoring HTML
geturl.tcl $1 | unhtml.pl | fmt.awk
```

Lệnh này lấy về trang web, loại bỏ tất cả thông tin định dạng và điều khiển, và định dạng lại văn bản theo quy tắc của nó. Đây là một cách nhanh chóng để lấy một trang văn bản từ web.

Hãy chú ý sự khác nhau của các ngôn ngữ đã được thảo luận, mỗi ngôn ngữ thích hợp cho một tác vụ nào đó: Tcl, Perl, Awk, và các ngôn ngữ biểu thức có quy tắc. Sức mạnh của chúng được thể hiện khác nhau trong

từng ngôn ngữ. Tcl đặc biệt tốt cho việc lấy về văn bản thông qua mạng; Perl và Awk tốt cho hiệu chỉnh và định dạng văn bản; và dĩ nhiên các biểu thức có quy tắc tốt cho xác định văn bản cho việc tìm kiếm và sửa chữa. Kết hợp các ngôn ngữ này với nhau sẽ mạnh hơn bất kỳ một ngôn ngữ riêng lẽ nào trong chúng. Một việc rất đáng được thực hiện là chia nhỏ công việc ra nếu điều này cho phép chúng ta tận dụng được thế mạnh từ hệ thống các ký hiệu của các ngôn ngữ này.

9.4. Các trình thông dịch, trình biên dịch và máy ảo

Một chương trình sẽ thực hiện như thế nào để chuyển từ dạng mã nguồn của nó sang dạng thực thi? Trong một ngôn ngữ đủ đơn giản, như trong `printf` hay các biểu thức có quy tắc đơn giản nhất của chúng ta chẳng hạn, chúng ta có thể chạy trực tiếp từ nguồn. Điều này dễ làm và khởi động rất nhanh.

Có một sự cân đối giữa thời gian cài đặt và tốc độ thi hành. Trong một ngôn ngữ phức tạp, người ta thích chuyển mã nguồn sang dạng thể hiện bên trong có hiệu quả và tiện lợi cho việc thi hành. Nó tốn thời gian để xử lý nguồn ban đầu nhưng bù lại là thi hành nhanh hơn. Các chương trình kết hợp việc đối thoại và thi hành vào trong một chương trình duy nhất có khả năng đọc mã nguồn, chuyển đổi nó, và chạy nó được gọi là các trình thông dịch. Awk và Perl thông dịch giống như nhiều *ngôn ngữ mục đích đặc biệt* và kịch bản khác.

Khả năng thứ ba là phát sinh các chỉ dẫn cho một loại máy tính đặc biệt mà chương trình sẽ chạy trên đó, giống như các trình biên dịch thực hiện. Điều này đòi hỏi có nhiều cố gắng và thời gian nhất trong giai đoạn đầu nhưng nó cho ra kết quả thi hành nhanh nhất.

Ngoài ra còn có các kết hợp khác. Một kết hợp mà chúng ta sẽ học trong phần này là biên dịch một chương trình thành các chỉ dẫn cho một máy tính giả lập (một máy ảo) có thể được giả lập trên bất kỳ một máy tính thực tế nào. Một máy ảo có khả năng kết hợp nhiều thế mạnh của các thông dịch và biên dịch truyền thống.

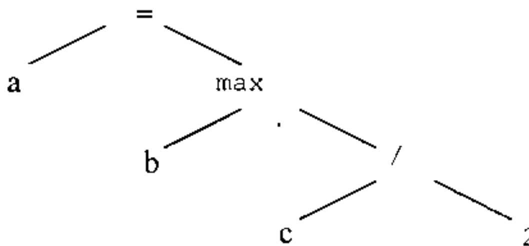
Trong một ngôn ngữ đơn giản thì không cần phải xử lý nhiều để hiểu được cấu trúc chương trình và chuyển đổi nó sang dạng thức bên trong. Tuy nhiên, trong một ngôn ngữ có một số phức tạp - các khai báo, các cấu trúc lồng nhau, các phát biểu hay các diễn đạt được định nghĩa đệ quy, các toán tử với thứ tự ưu tiên,... - thì sẽ phức tạp hơn trong việc phân tích dữ liệu nhập và xác định cấu trúc.

Các trình phân tích (parser) thường được viết với mục đích tạo ra một trình phát sinh phân tích tự động (automatic parser generator), còn được gọi là trình biên dịch của trình biên dịch (compiler – compiler), ví dụ như `yacc` hay `bison`. Các chương trình này dịch một mô tả của ngôn ngữ – được gọi là ngữ pháp của nó, sang (thường là) một chương trình C hay C++, và khi chương trình C/C++ này được biên dịch, nó sẽ dịch các phát biểu trong ngôn ngữ này sang một dạng thể hiện bên trong. Dĩ nhiên, việc phát sinh một trình phân tích trực tiếp từ một cấu trúc ngữ pháp là một minh họa khác về sức mạnh của hệ thống các ký hiệu tốt.

Dạng được tạo ra bởi một trình phân tích thường là một cây, với các nút trong chứa các toán tử (hay các chỉ dẫn điều khiển) và các nút lá chứa các toán hạng (hay các đối số của điều khiển). Phát biểu sau đây:

```
a = max(b, c/2);
```

có thể tạo ra cây phân tích (parse tree) hay còn gọi là cây cú pháp như sau:



Nhiều thuật toán cây được mô tả trong Chương 2 có thể được áp dụng để xây dựng và xử lý các cây phân tích.

Một khi một cây được xây dựng, có nhiều cách khác nhau để xử lý nó. Cách trực tiếp nhất, được dùng trong Awk, là duyệt cây một cách trực tiếp, tính số nút khi đi qua. Một phiên bản được đơn giản hoá của một tác vụ tính toán như thế trong một ngôn ngữ diễn đạt dựa trên số nguyên liên quan đến cách duyệt theo hậu thứ tự như sau:

```
typedef struct Symbol Symbol;
typedef struct Tree Tree;

struct Symbol {
      int   value;
      char  *name;
};

struct Tree {
      int         op;          /* mã số thao tác
tính toán */
      int         value;       /* chứa giá trị nếu
là số */
      Symbol      *symbol;     /* kiểu Symbol nếu
là biến */
      Tree        *left;
      Tree        *right;
};
```

```
/* eval: phiên bản 1: định giá biểu thức cây */
int eval(Tree *t)
{
	int left, right;

	switch (t->op) {
	case NUMBER:
		return t->value;
	case VARIABLE:
		return t->symbol->value;
	case ADD:
		return   eval(t->left)   +   eval(t-
>right);
	case DIVIDE:
		left = eval(t->left);
		right = eval(t->right);
		if (right == 0)
			eprintf("divide  %d  by  zero",
left);
		return left / right;
	case MAX:
		left = eval(t->left);
		right = eval(t->right);
```

```
                return left>right ? left : right;
        case ASSIGN:
                t->left->symbol->value    =    eval(t-
>right);
                return t->left->symbol->value;
        /* ... */
        }
}
```

Các trường hợp đầu tiên đánh giá các diễn đạt được đơn giản như các hằng số và giá trị; các trường hợp tiếp theo đánh giá các diễn đạt số học, và các trường hợp khác có thể thực hiện các xử lý đặc biệt, điều kiện, và vòng lặp. Để có thể cài đặt được các cấu trúc điều khiển, cây cần phải có thêm thông tin mô tả luồng điều khiển; vấn đề này không được trình bày ở đây.

Như trong hàm `pack` và `unpack`, chúng ta có thể thay thế một `switch` được mô tả rõ ràng bằng một bảng các con trỏ hàm. Các toán tử (hay chỉ dẫn điều khiển) riêng thì rất giống như trong phát biểu `switch` này:

```
/* addop: trả về tổng của hai biểu thức cây */
int addop(Tree *t)
{
        return eval(t->left) + eval(t->right);
}
```

Bảng các con trỏ hàm liên kết các chỉ dẫn điều khiển đến các hàm trình diễn các điều khiển:

```
enum { /* các mã số thao tác tính toán */
    NUMBER,
    VARIABLE,
    ADD,
    DIVIDE,
    /* ... */
};
/* optab: bảng chứa các chức năng thao tác tính
toán */
int (*optab[])(Tree *) = {
    pushop,          /* NUMBER */
    pushsymop,       /* VARIABLE */
    addop,           /* ADD */
    divop,           /* DIVIDE */
    /* ... */
};
```

Sau đây là đánh giá bằng cách sử dụng chỉ dẫn điều khiển để định vị bên trong bảng các con trỏ hàm để gọi hàm tương ứng: phiên bản này sẽ kích hoạt các hàm khác một cách đệ quy.

```
/* eval: phiên bản 2: định giá cây từ bảng thao
tác tính toán*/
int eval(Tree *t)
{
    return (*optab[t->op])(t);
}
```

Cả hai phiên bản của hàm `eval` đều đệ quy. Chúng ta cũng có các cách khử đệ quy, bao gồm một kỹ thuật thông minh được gọi là *đoạn mã nguồn theo thread*; nó hoàn toàn không sử dụng ngăn xếp. Một phương pháp đơn giản nhưng hiệu quả nhất là khử hoàn toàn đệ quy bằng cách lưu các hàm vào một mảng được dùng cho việc duyệt tuần tự để chạy chương trình. Mảng này trở thành một thứ tự liên tiếp các chỉ dẫn được một máy nhỏ phục vụ cho mục đích đặc biệt thi hành.

Chúng ta vẫn cần một ngăn xếp để lưu một phần các giá trị được đánh giá trong các tính toán, do đó hình thức của các hàm sẽ thay đổi, nhưng sự thay đổi này thì dễ thấy. Kết quả là, chúng ta tạo ra một *máy ngăn xếp* (stack machine) trong đó các chỉ dẫn là các hàm nhỏ và các toán hạng được lưu trong một ngăn xếp riêng biệt chứa các toán hạng. Đây không phải là một máy thật nhưng chúng ta có thể lập trình nó như thật, và chúng ta có thể cài đặt nó một cách dễ dàng như một trình thông dịch.

Chúng ta duyệt cây để phát sinh mảng các hàm để chạy chương trình thay vì duyệt để đánh giá nó. Mảng cũng chứa các giá trị dữ liệu mà các hướng dẫn dùng, như là các hằng số và các biến số (các biểu tượng), do đó các phần tử của mảng nên có kiểu là union:

```
typedef union Code Code;
union Code {
        void    (*op)(void);    /* hàm nếu là thao
tác tính toán */
        int     value;          /* giá trị nếu là
số */
        Symbol  *symbol;        /* kiểu Symbol nếu
là biến */
};
```

Sau đây là tác vụ phát sinh các con trỏ hàm và đưa chúng vào một mảng chứa các phần tử này, gọi là mảng `code`. Giá trị trả về của hàm `generate` thì không phải là giá trị của các diễn đạt sẽ được tính khi `code` phát sinh được thi hành, mà là chỉ số trong mảng `code` của điều khiển kế tiếp sẽ được phát sinh.

```
/* generate: phát sinh các chi thị bằng cách duyệt cây
*/

    int generate(int codep, Tree *t)

    {


        switch (t->op) {

        case NUMBER:

            code[codep++].op = pushop;

            code[codep++].value = t->value;

            return codep;

        case VARIABLE:

            code[codep++].op = pushsymop;

            code[codep++].symbol = t->symbol;

            return codep;

        case ADD:

            codep = generate(codep, t->left);

            codep = generate(codep, t->right);

            code[codep++].op = addop;

            return codep;
```

```
        case DIVIDE:
            codep = generate(codep, t->left);
            codep = generate(codep, t->right);
            code[codep++].op = divop;
            return codep;
        case MAX:
            /* ... */
        }
    }
```

Với phát biểu `a = max(b, c/2)`, mảng `code` được tổng hợp sẽ là:

```
pushsymop
b
pushsymop
c
pushop
2
divop
maxop
storesymop
a
```

Các hàm toán tử sẽ thao tác trên ngăn xếp, lấy các toán hạng ra khỏi ngăn xếp (pop) và đưa các kết quả vào ngăn xếp (push).

Trình thông dịch là một vòng lặp duyệt mảng các con trỏ hàm thông qua một biến đếm chương trình:

```
Code code[NCODE];
int stack[NSTACK];
int stackp;
int pc; /* biến đếm chương trình */

/* eval: phiên bản 3: định giá biểu thức từ mã nguồn đã phát sinh */
int eval(Tree *t)
{
    pc = generate(0, t);
    code[pc].op = NULL;

    stackp = 0;
    pc = 0;
    while (code[pc].op != NULL)
        (*code[pc++].op)();
    return stack[0];
}
```

Vòng lặp này chạy mô phỏng trong phần mềm trên máy ngăn xếp giả lập của chúng ta, giống như là những gì xảy ra trong phần cứng trên một máy thật sự. Sau đây là một số hàm toán tử tiêu biểu:

```
/* pushop: chèn số: giá trị là từ kế tiếp trong luồng mã số */
void pushop(void)
{
```

```
        stack[stackp++] = code[pc++].value;
}


/* divop: tính toán tỷ số của hai biểu thức */
void divop(void)
{
        int left, right;

        right = stack[--stackp];
        left = stack[--stackp];
        if (right == 0)
            eprintf("divide %d by zero\n", left);
        stack[stackp++] = left / right;
}
```

Chú ý rằng kiểm tra chia cho `0` xuất hiện trong hàm `divop`, chứ không phải trong hàm `generate`.

Các lệnh có điều kiện, các rẽ nhánh, và các vòng lặp vận hành bằng cách sửa biến đếm chương trình trong một hàm toán tử, đưa một nhánh tới một điểm khác trong mảng các hàm. Ví dụ, một điều khiển `goto` luôn gán giá trị của biến `pc`, trong khi một nhánh điều kiện chỉ gán `pc` khi điều kiện là đúng.

Mảng `code` dĩ nhiên là phần bên trong của trình thông dịch, nhưng hãy tưởng tượng xem trong trường hợp chúng ta muốn lưu chương trình được phát sinh vào một tập tin. Nếu chúng ta nêu ra các địa chỉ hàm thì kết quả sẽ không linh hoạt và khó sử dụng. Tuy nhiên, thay vào đó chúng ta sẽ

nêu ra các hằng số đại diện cho các hàm, ví dụ `1000` cho `addop`, `1001` cho `pushop`,..., và dịch ngược chúng sang các con trỏ hàm khi chúng ta đọc chương trình vào cho việc thông dịch.

Nếu chúng ta quan sát một tập tin do thủ tục này tạo ra, nó sẽ trông giống như một luồng chỉ dẫn cho một máy ảo mà ở đó các chỉ dẫn của nó tương ứng với các điều khiển cơ bản của ngôn ngữ nhỏ của chúng ta, và hàm `generate` thì thật sự là một trình biên dịch; nó dịch ngôn ngữ này sang dạng máy ảo. Các máy ảo là một ý tưởng cũ tuyệt vời, gần đây được phổ biến trở lại bởi Java và máy ảo Java (Java Virtual Machine – JVM); chúng hỗ trợ một cách dễ dàng để tạo ra các thể hiện hiệu quả, linh hoạt của các chương trình được viết trong một ngôn ngữ bậc cao.

9.5. Các chương trình để viết các chương trình khác

Điểm nổi bật nhất của hàm `generate` có lẽ là nó là một chương trình để viết một chương trình: kết quả của nó là một loạt chỉ dẫn có thể chạy được trên một máy (ảo) khác. Các trình thông dịch luôn làm việc này, dịch mã nguồn sang các chỉ dẫn máy, do đó ý tưởng này chắc chắn không có gì mới. Trong thực tế, các chương trình để viết các chương trình khác xuất hiện dưới nhiều dạng.

Một ví dụ thường gặp là việc phát sinh động HTML cho các trang web. HTML là một ngôn ngữ hạn chế, và nó cũng có thể chứa mã JavaScript. Các trang web thường được phát sinh trực tiếp trong khi chạy bởi các chương trình Perl hay C, với các nội dung đặc thù (ví dụ các kết quả tìm kiếm và các dịch vụ quảng cáo) được xác định bởi các yêu cầu đang được thực hiện. Chúng ta đã sử dụng các ngôn ngữ chuyên dụng cho các đồ thị, hình ảnh, bảng biểu, các biểu thức toán học, các chỉ mục trong quyển sách này. Lấy một ví dụ khác, PostScript là một ngôn ngữ lập trình được phát sinh từ các trình xử lý văn bản, các chương trình vẽ, và nhiều nguồn khác; ở giai đoạn cuối của quá trình xử lý, toàn bộ quyển sách được thể hiện như một chương trình PostScript với 57000 dòng.

Một tài liệu là một chương trình tĩnh, nhưng ý tưởng sử dụng một

ngôn ngữ lập trình như hệ thống ký hiệu để giải quyết bất kỳ một bài toán nào thì cực kỳ hay. Nhiều năm qua, các lập trình viên ao ước có được các máy tính viết tất cả các chương trình cho họ. Có lẽ điều này vẫn chỉ là một ước mơ, tuy nhiên các máy tính ngày nay vẫn đều đặn viết các chương trình cho chúng ta, thường để thể hiện những việc mà trước kia chúng ta đã không quan tâm đến trong các chương trình.

Chương trình viết ra chương trình thông dụng nhất là một trình biên dịch; nó dịch các ngôn ngữ cấp cao sang mã máy. Thật sự hữu ích nếu có thể dịch mã nguồn sang một ngôn ngữ lập trình. Trong phần trước, chúng ta đã đề cập đến việc các bộ phát sinh đã chuyển một định nghĩa cấu trúc ngữ pháp của một ngôn ngữ sang một chương trình C, trong đó ngôn ngữ C có khả năng phân tích ngôn ngữ. C thường được dùng theo cách này và được sánh như một loại "ngôn ngữ hợp ngữ (assembly) bậc cao". Modula-3 và C++ là hai trong số các ngôn ngữ được dùng cho mục đích tổng quát với các trình biên dịch đầu tiên tạo ra mã nguồn C, sau đó mã nguồn C này được một trình biên dịch C chuẩn biên dịch. Cách tiếp cận này có nhiều thuận lợi, bao gồm: hiệu quả – bởi vì theo nguyên tắc thì các chương trình có thể chạy nhanh như các chương trình C – và khả chuyển – bởi vì các trình biên dịch có thể được mang đến bất kỳ một hệ thống nào có một trình biên dịch C. Điều này mang lại nhiều lợi ích trong giai đoạn đầu phổ biến ngôn ngữ này.

Lấy một ví dụ khác, giao diện đồ họa của Visual Basic phát sinh các lệnh gán để khởi tạo các đối tượng; người dùng chọn các lệnh gán này từ các danh sách và dùng chuột đặt lên màn hình. Nhiều ngôn ngữ khác cũng có hệ thống phát triển phần mềm "trực quan" và sử dụng "wizard" để tạo ra mã nguồn giao diện chỉ thông qua các thao tác nhấp chuột.

Mặc dù đã có nhiều bộ phát sinh chương trình tiện lợi và có sẵn nhiều ví dụ hay nhưng các ký hiệu vẫn chưa được đánh giá cao và thường được các lập trình viên đơn lẻ sử dụng. Tuy nhiên, chúng ta cũng có thể hưởng một số ích lợi từ việc phát sinh mã nguồn bằng một chương trình. Sau đây là một số ví dụ phát sinh mã C hay C++.

Hệ điều hành Plan 9 phát sinh các thông điệp lỗi từ một tập tin tiêu

đề chứa các tên lỗi và các chú thích; các chú thích được chuyển đổi một cách máy móc sang các chuỗi trong dấu nháy và được đưa vào một mảng được đánh chỉ số bằng cách liệt kê. Đoạn chương trình sau trình bày cấu trúc của tập tin tiêu đề:

```
/* errors.h: thông điệp lỗi chuẩn */

enum {
      Eperm,      /* Permission denied */
      Eio,        /* I/o error */
      Efile,      /* File does not exist */
      Emem,       /* Memory limit reached */
      Espace,     /* Out of file space */
      Egreg       /* It's all Greg's fault */
};
```

Với dữ liệu cho trước như vậy, một chương trình đơn giản có thể tạo ra các khai báo cho thông điệp lỗi như sau:

```
/* machine-generated; không được sửa */
char *err[] = {
      "Permission denied",    /* Eperm */
      "I/O error",            /* Eio */
      "File does not exist",  /* Efile */
      "Memory limit reached", /* Emem */
      "Out of file space",    /* Espace */
      "It's all Greg's fault",/* Egreg */
```

Có một số lợi ích từ cách tiếp cận này. Trước hết, mối liên hệ giữa các giá trị enum và các chuỗi thông báo được đưa ra rõ ràng và dễ dàng độc lập với ngôn ngữ tự nhiên. Ngoài ra, do thông tin chỉ xuất hiện một lần, nghĩa là chỉ có "một chân lý duy nhất" mà từ đó các mã nguồn khác được phát sinh, do đó chỉ cần cập nhật thông tin ở một nơi duy nhất. Giả sử thông tin được cập nhật ở nhiều nơi thì chắc chắn đến một lúc nào đó thông tin sẽ không thống nhất. Cuối cùng, bất kỳ khi nào tập tin tiêu đề thay đổi, tập tin .c sẽ được tạo lại và biên dịch lại. Khi một thông điệp lỗi được thay đổi, thì chỉ cần sửa trong tập tin tiêu đề. Các thông điệp được cập nhật một cách tự động.

Trình phát sinh có thể được viết trong bất kỳ ngôn ngữ nào. Một ngôn ngữ xử lý chuỗi như Perl có thể làm điều này dễ dàng.

```
	# enum.pl: phát sinh chuỗi lỗi từ các chú thích
kiểu enum

	print "/* machine-generated; không được sửa.
*/\n\n";

	print "char *err[] = {\n";


	while (<>) {

		chop;				#xóa dòng mới

		if (/^\s*(E[a-z0-9]+),?/) {	#từ đầu tiên
là E ...

			$name = $1;		#lưu tên

			s/.*\/\* *//;		# xóa

			s/ *\*\///;

			print "\t\"$_\", /* $name */\n";

		}
```

```
}
print "};\n";
```

Các biểu thức có quy tắc lại một lần nữa được dùng đến. Các dòng có các trường đầu tiên trông giống như các dấu nhận diện được theo sau bởi một dấu phẩy là các trường được chọn ra. Thay thế thứ nhất xóa tất cả cho đến ký tự không phải là khoảng trắng đầu tiên của chú thích, trong khi thay thế thứ hai loại bỏ dấu kết thúc chú thích và bất kỳ khoảng trắng nào trước nó.

Để kiểm chứng trình biên dịch, Andy Koenig đưa ra một cách thức tiện lợi để viết mã nguồn C++ nhằm kiểm tra xem trình biên dịch có bắt được các lỗi chương trình hay không. Các đoạn mã nguồn làm cho trình biên dịch thực hiện kiểm tra được gắn vào các chú thích khác thường để mô tả các thông điệp được mong đợi. Mỗi dòng có một chú thích bắt đầu bằng `///` (để phân biệt nó với các chú thích bình thường) và một biểu thức có quy tắc phù hợp với các kiểm tra trên dòng đó. Do đó, hai đoạn mã nguồn sau đây chẳng hạn sẽ phát sinh các kiểm tra:

```
int f() {}
        /// warning.* non-void function .* should
return a value

void g() {return 1;}
        /// error.* void function may not return a
value
```

Nếu chúng ta chạy hàm kiểm tra thứ hai thông qua trình biên dịch C++, thì nó sẽ in ra thông điệp như mong muốn và phù hợp với các biểu thức có quy tắc.

```
% CC x.c
"x.c", line 1: error(321): void function may not
return a value
```

Mỗi đoạn như thế được đưa đến trình biên dịch, và dữ liệu xuất được so sánh với các kết quả kiểm tra biết trước, đây là một tiến trình được quản lý bằng sự kết hợp giữa chương trình Shell và Awk. Các lỗi này cho biết kết quả do trình biên dịch thực hiện được khác với những kết quả mong muốn. Vì các chú thích là các biểu thức có quy tắc nên có một số khác biệt trong dữ liệu xuất; chúng có thể ít nhiều được thay đổi tùy ý phụ thuộc vào yêu cầu.

Ý tưởng về các chú thích có ngữ nghĩa không phải là mới. Chúng xuất hiện trong PostScript, ở đó các chú thích bình thường bắt đầu bằng `%`. Các chú thích theo truyền thống bắt đầu bằng `%%` có thể chứa thêm thông tin phụ về các số trang, các hộp có đường biên, các tên phông chữ và những thông tin tương tự khác.

```
%%PageBoundingBox: 126 307 492 768
%%Pages: 14
%%DocumentFonts: Helvetica Times-Italic Times-
Roman
            LucidaSans-Typewriter
```

Trong tất cả các ví dụ ở trên, điều quan trọng là phải nhận ra được vai trò của các ký hiệu, sự kết hợp của các ngôn ngữ, và việc sử dụng các công cụ.

Bài tập 9-15. Một trong những trường hợp thường gặp là viết một chương trình mà khi nó được thi hành sẽ tạo ra chính nó dưới dạng nguồn. Đây là một trường hợp đặc biệt có tổ chức của một chương trình để viết một chương trình khác. Hãy thử cài đặt chương trình với ngôn ngữ mà bạn thích.

9.6. Sử dụng các macro để phát sinh mã nguồn

Chuyển xuống một vài cấp thấp hơn, chúng ta có thể sử dụng các macro để viết mã lúc biên dịch. Xuyên suốt quyển sách này, việc sử dụng các macro và biên dịch có điều kiện đã được cảnh báo bởi vì chúng đưa ra một phong cách lập trình có nhiều điểm rắc rối. Nhưng chúng cũng có vị trí của nó; sự thay thế loại văn bản đôi khi chính là câu trả lời chính xác cho vấn đề đó. Một ví dụ là sử dụng bộ tiền xử lý macro C/C++ để tập hợp các phần được lặp lại nhiều lần của chương trình.

Chẳng hạn, chương trình tính tốc độ của các cấu trúc ngôn ngữ sơ cấp trong Chương 7 sử dụng bộ tiền xử lý C để tập hợp các kiểm chứng bằng cách gom chúng vào trong mã nguồn trước khi biên dịch. Trọng tâm của việc kiểm tra là đóng gói mã nguồn vào một vòng lặp để khởi tạo bộ đếm thời gian, chạy đoạn mã nguồn này nhiều lần, dừng bộ đếm thời gian, và thông báo kết quả. Tất cả các đoạn mã nguồn lặp lại được mô tả trong một số macro và đoạn mã nguồn được đặt thời gian được truyền vào như một đối số. Macro chính có dạng như sau:

```
#define LOOP(CODE) {                              \
        t0 = clock();                             \
        for (i = 0; i < n; i++) { CODE; }         \
        printf("%7d ", clock() - t0);             \
}
```

Các dấu \ cho phép thân macro ngắt ra thành nhiều dòng. Macro này được dùng trong các "phát biểu" có dạng đặc trưng như sau:

```
LOOP(f1 = f2)
LOOP(f1 = f2 + f3)
LOOP(f1 = f2 - f3)
```

Đôi khi có các lệnh khác được dùng cho việc khởi tạo, tuy nhiên phần định thời gian cơ sở thì được thể hiện trong các phân đoạn chỉ có một

đối số này; các phân đoạn này mở rộng mã nguồn một lượng đáng kể.

Tiến trình xử lý macro cũng có thể được dùng để phát sinh mã nguồn. Bart Locanthi có lần viết một phiên bản của một trình điều khiển đồ họa hai chiều, đó là `bitblt` hay `rasterop`. Khó có thể tạo ra trình điều khiển này một cách nhanh chóng, bởi vì có nhiều đối số kết hợp với nhau theo các cách thức phức tạp. Thông qua việc xử lý cẩn thận các trường hợp có thể xảy ra, Locanthi làm giảm các sự kết hợp đối với các vòng lặp riêng lẻ, các vòng lặp này có thể được tối ưu hoá riêng lẻ. Tiếp đến, mỗi trường hợp có thể được cấu trúc bằng việc thay thế macro, tương tự như ví dụ kiểm tra sự vận hành, với tất cả các trường hợp khác nhau được giải quyết trong một lệnh switch lớn. Mã nguồn nguyên thủy chỉ có vài trăm dòng nhưng kết quả do quá trình xử lý macro tạo ra thì có tới vài ngàn dòng. Mã nguồn được mở rộng bằng macro là không tối ưu, nhưng khi xem xét khó khăn của vấn đề thì đó là cách thực tế và rất dễ để tạo ra nó. Hơn nữa, đối với mã nguồn có thể thực thi tốt thì mã nguồn này cũng mang tính khả chuyển tương ứng.

Bài tập 9-16. Bài tập 7-7 liên quan đến việc ước lượng chi phí cho các điều khiển khác nhau trong C++. Sử dụng các ý tưởng của phần này để tạo ra một phiên bản khác của chương trình.

Bài tập 9-17. Bài tập 7-8 liên quan đến việc tạo lập một mô hình chi phí cho Java, không có khả năng macro. Hãy giải quyết vấn đề bằng cách viết một chương trình khác, có thể dùng bất kỳ một ngôn ngữ nào, để viết phiên bản Java này và kích hoạt tự động các xử lý định thời gian.

9.7. Biên dịch trong khi thực thi

Trong phần trước, chúng ta đã thảo luận về các chương trình có khả năng viết các chương trình. Với mỗi ví dụ, chương trình được phát sinh là ở dạng nguồn; nó cần được biên dịch hay thông dịch để chạy. Nhưng chúng ta cũng có thể phát sinh mã nguồn có thể chạy ngay lập tức bằng cách tạo ra các chỉ dẫn máy thay vì nguồn. Điều này được biết đến như là việc biên dịch "sẵn".

Mặc dù mã nguồn đã được biên dịch cần thiết phải là không khả

chuyển – chỉ chạy trên một bộ xử lý duy nhất – nó có thể chạy rất nhanh.

Hãy xem xét câu lệnh:

```
max(b, c/2)
```

Phép tính này phải xác định giá trị của `c`, chia cho 2, so sánh kết quả với `b` và chọn số lớn hơn. Nếu chúng ta sử dụng máy ảo, được nói đến ở đầu chương, để tính giá trị câu lệnh trên thì chúng ta có thể bỏ qua việc kiểm tra chia 0 trong hàm `divop`. Vì 2 khác 0 nên kiểm tra là dư thừa. Nhưng với các vấn đề nêu ra trong các thiết kế áp dụng cho việc cài đặt máy ảo, thì chúng ta không thể bỏ qua quá trình kiểm tra; mỗi lần thực hiện lệnh chia đều phải so sánh số chia với 0.

Đây chính là điểm mà mã nguồn phát sinh có thể giải quyết một cách linh động. Nếu chúng ta xây dựng mã nguồn cho câu lệnh trên một cách trực tiếp, thay vì dùng lại các hàm được định nghĩa trước đó, chúng ta có thể trách được việc kiểm tra chia 0 đối với các số chia được biết là khác 0. Thật ra, chúng ta thậm chí có thể đi xa hơn nếu toàn bộ biểu thức là hằng số. Chẳng hạn `max(3*3, 4/2)`, chúng ta có thể tính toán ngay khi phát sinh mã nguồn, và thay thế bằng một hằng có giá trị 9. Nếu biểu thức xuất hiện trong một vòng lặp, thì chúng ta có thể giảm được thời gian qua mỗi lần lặp, và nếu vòng lặp thực hiện đủ số lần lặp thì chúng ta sẽ lấy lại được chi phí phải trả cho việc phân tích biểu thức và phát sinh mã nguồn cho nó.

Ý tưởng chính là hệ thống các ký hiệu cung cấp cho chúng ta một cách tổng quát để diễn đạt vấn đề. Tuy nhiên, trình biên dịch cho hệ thống các ký hiệu có thể hiệu chỉnh mã nguồn cho các chi tiết của một tính toán đặc thù. Ví dụ, trong một máy ảo dành cho các biểu thức có quy tắc thì chúng ta có lẽ dùng riêng một hàm để kiểm tra một ký tự:

```
int matchchar(int literal, char *text)
{
        return *text == literal;
}
```

Chúng ta có thể phát sinh mã nguồn cho một mẫu cụ thể. Tuy nhiên, giá trị của đối số `literal` cho trước phải phù hợp, giả sử là `'x'`, do đó thay vào đó chúng ta có thể dùng một toán tử như sau:

```
int matchx(char *text)
{
	return *text == 'x';
}
```

Và do đó, thay vì định nghĩa trước một hàm đặc biệt cho mỗi giá trị ký tự theo dạng thức được đưa ra, chúng ta có thể đơn giản hoá bằng cách phát sinh mã nguồn cho các hàm thật sự cần thiết cho biểu thức hiện hành. Mở rộng ý tưởng cho tập đầy đủ các điều khiển, chúng ta có thể viết một trình biên dịch "sẵn" để dịch một biểu thức có quy tắc hiện hành sang mã nguồn đặc biệt được tối ưu hoá cho biểu thức đó.

Ken Thompson thực hiện chính xác công việc này cho việc cài đặt các biểu thức có quy tắc trên máy IBM 7094 trong năm 1967. Phiên bản này phát sinh các khối nhỏ của các chỉ dẫn nhị phân 7094 cho các điều khiển khác nhau trong biểu thức, kết hợp chúng lại với nhau, và chạy chương trình kết quả bằng cách gọi nó, giống như một hàm bình thường. Các kỹ thuật tương tự có thể được áp dụng để tạo ra các thứ tự chỉ dẫn đặc trưng cho các công việc cập nhật màn hình trong các hệ thống đồ họa, nơi mà có nhiều trường hợp đặc biệt sao cho việc tạo lập mã nguồn linh động cho mỗi trường hợp phát sinh sẽ hiệu quả hơn là viết trước tất cả hay thêm vào các kiểm tra điều kiện trong mã nguồn tổng quát hơn.

Để minh họa các vấn đề liên quan đến việc xây dựng một trình biên dịch "sẵn" thật sự, chúng ta sẽ phải đi sâu nhiều hơn vào các chi tiết của một tập chỉ dẫn (lệnh) cụ thể, nhưng điều này đáng được bỏ ra một chút thời gian để hiểu được cách thức thực hiện của một hệ thống như thế. Phần còn

lại của phần này nhằm cung cấp các ý tưởng và hiểu được bản chất vấn đề, chứ không đi vào các chi tiết thực hiện.

Hãy nhớ lại rằng chúng ta đã đưa ra máy ảo với một cấu trúc như sau:

```
Code code[NCODE];
int stack[NSTACK];
int stackp;
int pc; /* program counter */
    ...
    Tree *t;

    t = parse();
    pc = generate(0, t);
    code[pc].op = NULL;

    stackp = 0;
    pc = 0;
    while (code[pc].op != NULL)
        (*code[pc++].op)();
    return stack[0];
```

Để điều chỉnh đoạn mã nguồn này cho phù hợp với sự biên dịch "sẵn", chúng ta phải thực hiện một vài thay đổi. Trước hết, mảng `code` không còn là một mảng các con trỏ hàm nữa mà phải là một mảng các chỉ dẫn có thể thi hành được. Kiểu của các chỉ thị là `char`, `int`, hay `long` sẽ phụ thuộc vào bộ xử lý mà chúng ta sẽ biên dịch; chúng ta sẽ giả sử là kiểu `int`. Sau khi đoạn mã nguồn được phát sinh, chúng ta sẽ gọi nó như gọi một

hàm. Sẽ không có bất kỳ một bộ đếm chương trình ảo nào bởi vì chu trình chạy của bộ xử lý sẽ đi dọc theo đoạn mã nguồn cho chúng ta; một khi phép tính được thực hiện xong, nó sẽ trả về giá trị, giống như một hàm bình thường. Cũng vậy, chúng ta có thể chọn cách duy trì một ngăn xếp riêng biệt chứa toán hạng cho máy hay sử dụng ngăn xếp riêng của bộ xử lý. Mỗi cách tiếp cận có các thuận lợi riêng, tuy nhiên, chúng ta sẽ chọn phương pháp dùng một ngăn xếp riêng và tập trung vào các chi tiết của chính đoạn mã nguồn này. Cài đặt bây giờ sẽ như sau:

```
typedef int Code;
Code code[NCODE];
int codep;
int stack[NSTACK];
int stackp;
    ...
    Tree *t;
    void (*fn)(void);
    int pc;

    t = parse();
    pc = generate(0, t);
    genreturn(pc);    /* phát sinh dãy function
return */
    stackp = 0;
    flushcaches();    /* đồng bộ bộ nhớ với bộ
vi xử lý */
fn = (void(*)(void)) code;
```

```
                    /* ép kiểu màng sang con trỏ
hàm */
        (*fn)();          /* thực hiện gọi hàm */
        return stack[0];
```

Sau khi hàm `generate` thực hiện xong, hàm `genreturn` đưa ra các chỉ thị yêu cầu mã nguồn được phát sinh trả lại điều khiển cho hàm `eval`.

Hàm `flushcaches` thi hành các bước cần thiết để chuẩn bị bộ xử lý cho việc chạy mã nguồn được phát sinh gần đây. Các máy hiện đại chạy nhanh một phần là vì chúng sử dụng cache cho các chỉ thị, dữ liệu, và các đường ống (pipeline) bên trong lấn một phần qua sự thi hành của các chỉ thị liên tiếp nhau. Các cache và đường ống này mong đợi luồng chỉ thị là tĩnh; nếu chúng ta phát sinh mã nguồn ngay trước khi thi hành, bộ xử lý có thể trở nên bối rối. CPU cần rút ngắn đường ống của nó và làm sạch các cache của nó trước khi nó thi hành các chỉ thị mới phát sinh. Đây là các điều khiển phụ thuộc rất nhiều vào máy; cài đặt của hàm `flushcaches` sẽ khác nhau trên các loại máy khác nhau.

Đoạn mã nguồn nguồn đáng chú ý ở đây là `(void(*)(void))`, được dùng để chuyển đổi địa chỉ của mảng chứa các chỉ thị được phát sinh sang một con trỏ có thể được dùng để gọi đoạn mã nguồn như một hàm.

Về phương diện kỹ thuật, không quá khó để phát sinh mã nguồn, mặc dù có không ít công nghệ cho phép thực hiện việc này một cách hiệu quả. Chúng ta bắt đầu với việc xây dựng các khối. Như trước đó, mảng `code` và chỉ số truy xuất mảng được duy trì trong suốt quá trình biên dịch. Để đơn giản, chúng ta sẽ khai báo chúng đều là toàn cục, như chúng ta đã làm trước đó. Tiếp đến chúng ta có thể viết một hàm để phát sinh ra các chỉ thị

```
/* emit: thêm chỉ thị vào luồng mã nguồn số */
void emit(Code inst)
{
```

```
code[codep++] = inst;
```

Các chỉ thị tự chúng có thể được định nghĩa bởi các macro phụ thuộc bộ xử lý hoặc các hàm nhỏ thiết lập các chỉ thị bằng cách điền vào các trường của từ chỉ thị. Một cách giả thuyết, chúng ta có thể có một hàm gọi là `popreg` dùng để phát sinh mã nguồn để lấy một giá trị ra khỏi ngăn xếp và lưu nó vào một thanh ghi xử lý, và một hàm khác gọi là `pushreg` dùng để phát sinh mã nguồn để lấy giá trị trong một thanh ghi và đưa nó vào ngăn xếp. Hàm `addop` được tổ chức lại của chúng ta sẽ sử dụng chúng như dưới đây, với một vài hằng số được định nghĩa trước đó mô tả các chỉ thị (như `ADDINST`) và dạng thức của chúng (các vị trí `SHIFT` khác nhau định nghĩa định dạng):

```
/* addop: phát sinh chỉ thị ADD */
void addop(void)
{
        Code inst;
        popreg(2);          /* lấy từ ngăn xếp đưa vào thanh ghi 2 */
        popreg(1);          /* lấy từ ngăn xếp đưa vào thanh ghi 1 */
        inst = ADDINST << INSTSHIFT;
        inst |= (R1) << OP1SHIFT;
        inst |= (R2) << OP2SHIFT;
        emit(inst);         /* thao tác ADD R1, R2 */
        pushreg(2);         /* đẩy giá trị thanh ghi 2 vào ngăn xếp */
```

Đây chỉ là điểm khởi đầu, nếu chúng ta định viết một trình biên dịch trực tiếp khi chạy chương trình thực sự thì chúng ta phải thực hiện các tối ưu hoá. Nếu chúng ta thêm một hằng số, chúng ta không cần đẩy hằng số vào ngăn xếp, lấy nó ra khỏi ngăn xếp, và thêm nó vào; chúng ta có thể thêm một cách trực tiếp. Ý tưởng tương tự có thể làm giảm bớt chi phí. Thậm chí như đã viết như trên, tuy nhiên, `addop` sẽ chạy nhanh hơn nhiều so với các phiên bản được viết trước đó bởi vì các điều khiển khác nhau không được tập hợp lại bằng các lời gọi hàm. Thay vào đó, mã nguồn dùng để chạy chúng được đặt vào bộ nhớ như là một khối các lệnh đơn, với bộ đếm chương trình của bộ xử lý thật sự làm tất cả các công việc tập hợp lại cho chúng ta.

Hàm `generate` có vẻ rất phù hợp cho việc thực hiện của máy ảo. Nhưng ở đây là các chỉ thị của máy thực sự thay vì các con trỏ tới các hàm được định nghĩa trước. Và để phát sinh mã nguồn hiệu quả, cần phải có một vài thực hiện về tìm kiếm các hằng số để loại bỏ và các tối ưu hoá khác.

Việc trình bày nhanh gọn của chúng ta về sự phát sinh mã nguồn, chỉ ra cho thấy một cách sơ lược một vài kỹ thuật được dùng bởi các trình biên dịch thực sự và còn rất nhiều vấn đề chưa được đề cập tới. Nó cũng bỏ qua nhiều vấn đề phát sinh bởi các phức tạp của các CPU hiện đại. Nhưng nó cũng minh họa cách thức mà một chương trình có thể phân tích mô tả của một vấn đề tạo ra mã nguồn với mục đích đặc biệt nhằm giải quyết nó một cách hiệu quả. Bạn có thể dùng các ý tưởng này để viết một phiên bản cực kỳ nhanh của `grep`, nhằm thay đổi ngôn ngữ nhỏ của chính bạn, để thiết kế và xây dựng một máy ảo được tối ưu hoá cho tính toán có mục đích đặc biệt, hay thậm chí, với một ít trợ giúp, để viết một trình biên dịch cho một ngôn ngữ thú vị.

Giữa một biểu thức có quy tắc và một chương trình C++ là một khoảng cách lớn, nhưng cả hai chỉ là hệ thống các ký hiệu được dùng để giải quyết các vấn đề. Với một hệ thống ký hiệu đúng, nhiều vấn đề trở nên dễ hơn. Và việc thiết kế và cài đặt hệ thống các ký hiệu có thể rất thú vị.

Bài tập 9-18. Trình biên dịch trong khi thực thi sẽ phát sinh mã nguồn thi hành nhanh hơn nếu nó có thể thay thế các biểu thức chứa các hằng số bằng giá trị của chúng. Khi nó gặp một biểu thức như thế thì nó sẽ tính giá trị của biểu thức đó như thế nào? Bạn hãy thử đề ra một cách để giải quyết.

Bài tập 9-19. Hãy đề ra một cách thức để kiểm tra trình biên dịch trong khi thực thi.

PHỤ LỤC

Phong cách

Dùng tên gợi nhớ cho biến toàn cục, tên ngắn gọn cho biến cục bộ.

Nhất quán.

Dùng tên hàm mang ý nghĩa chủ động.

Chính xác.

Canh chỉnh lề để thấy được cấu trúc.

Dùng hình thức diễn đạt tự nhiên.

Đóng và mở ngoặc để mã nguồn không bị mơ hồ.

Phân tích những biểu thức phức tạp thành những biểu thức đơn giản hơn.

Tính sáng sủa.

Cẩn thận với các hiệu ứng lề.

Sử dụng phong cách canh chỉnh lề và dấu ngoặc một cách nhất quán.

Dùng nhất quán các đặc ngữ.

Áp dụng các *else-if* cho các quyết định rẽ nhánh.

Tránh dùng các macro hàm.

Đóng và mở ngoặc phần chương trình cũng như các tham số của macro.

Đặt tên cho các số tối nghĩa.

Định nghĩa các con số dưới dạng hằng số, chứ không dùng macro.

Sử dụng các hằng ký tự, không nên dùng các số nguyên.

Dùng chính ngôn ngữ đó để tính toán kích thước của đối tượng.

Đừng tô đậm những điều hiển nhiên.

Ghi chú thích cho các hàm và biến toàn cục.

Đừng ghi chú thích cho các đoạn mã nguồn dở mà hãy viết lại nó.

Đừng phủ nhận mã nguồn.

Rõ ràng, không gây nhầm lẫn.

Các phương thức giao tiếp

Che giấu các chi tiết cài đặt.

Dùng tập các cấu trúc dữ liệu chuẩn.

Thực hiện cùng một cách đối với cùng một thể hiện ở mọi nơi.

Giải phóng tài nguyên tương ứng với cách thức cấp phát.

Dò tìm lỗi ở cấp độ thấp, xử lý chúng ở cấp độ cao.

Chỉ sử dụng ngoại lệ cho các tình huống ngoại lệ.

Gỡ rối

Tìm những mẫu tương tự.

Xem xét những thay đổi gần nhất.

Không lặp lại những lỗi đã mắc phải.

Gỡ rối ngay, không nên để lại về sau.

Cần kiểm tra lại ngăn xếp (stack).

Đọc trước khi gõ vào máy.

Giải thích mã nguồn để người khác có thể đọc hiểu.

Phải nghĩ rằng một lỗi đã phát hiện vẫn có thể xảy ra.

Áp dụng chiến lược "chia để trị".

Hiển thị kết quả để khoanh vùng tìm kiếm.

Viết các đoạn mã nguồn tự kiểm tra.

Viết một tập tin lưu vết (log file)

Vẽ hình.

Sử dụng công cụ.

Lưu giữ lại các bản ghi nhận về lỗi.

Kiểm chứng

Kiểm chứng mã nguồn tại các điều kiện biên.

Kiểm tra các điều kiện cần và đủ.

Dùng các khẳng định.

Lập trình cẩn thận.

Kiểm tra các giá trị lỗi trả về.

Thực hiện kiểm chứng ngày càng nhiều.

Kiểm tra những phần đơn giản trước.

Biết trước kết quả cần xuất ra.

Kiểm định các thuộc tính duy trì.

So sánh các phần cài đặt độc lập.

Đo lường tính bao quát của việc kiểm chứng.

Tự động hóa quá trình kiểm chứng lùi.

Tốc độ thực thi chương trình

Tự đông hóa quá trình đo lường thời gian.

Sử dụng profiler.

Tập trung vào các phần cần giải quyết tốt.

Vẽ hình.

Dùng thuật toán và cấu trúc dữ liệu tốt hơn.

Kích hoạt lựa chọn về tối ưu do trình biên dịch cung cấp.

Điều chỉnh mã nguồn.

Không cần tối ưu những phần không quan trọng.

Lựa chọn những biểu thức con.

Thay những phép toán tốn tài nguyên bằng các phép toán ít tốn hơn.

Hạn chế sử dụng vòng lặp.

Thực hiện cache những dữ liệu thường dùng.

Viết một hàm cấp phát tài nguyên cho những mục đích đặc biệt.

Cất giữ dữ liệu đầu vào và đầu ra vào vùng đệm.

Xử lý riêng lẻ các trường hợp đặc biệt.

Tính toán trước các kết quả.

Dùng các giá trị xấp xỉ.

Viết lại bằng ngôn ngữ cấp thấp hơn.

Tiết kiệm không gian bằng cách dung những kiểu dữ liệu nhỏ nhất (nếu có thể được)

Đừng lưu trữ lại những gì bạn có thể dễ dàng tính toán lại.

Tính khả chuyển

Bám sát các chuẩn.

Lập trình theo xu hướng chính.

Chú ý những điểm mà ngôn ngữ thường gây rắc rối.

Thử trên vài trình biên dịch.

Dùng các thư viện chuẩn.

Sử dụng các tính năng đã có sẵn.

Tránh dùng cách biên dịch có điều kiện.

Cục bộ hóa những phụ thuộc hệ thống bằng những tập tin riêng lẻ.

Che giấu những phụ thuộc hệ thống đằng sau các phương thức.

Dùng văn bản để trao đổi dữ liệu.

Dùng thứ tự byte cố định để trao đổi dữ liệu.

Thay đổi tên nếu thay đổi đặc tả.

Duy trì tính tương thích với hệ thống cũng như dữ liệu đang tồn tại.

Đừng giả sử đang sử dụng bảng mã ASCII.

Đừng giả sử đang dùng tiếng Anh.

Tài liệu tham khảo

[1] *Brian Kernighan and P. J. Plauger,* The Elements of Programming Style, McGraw-Hill, 1978.

[2] *Peter van der Linder,* Expert C Programming: Deep C Secrets, Prentice Hall, 1994.

[3] *Don Knuth,* The Art of Computer Programming, Volume 3, 2nd Edition, Addison Wesley, 1998.

[4] *Gerard Holzmann,* Design and Validation of Computer Protocols, Prentice Hall, 1991.

[5] *Matthew Austern,* Generic Programming and the STL, Addison Wesley, 1998.

[6] *Bjarne Stroustrup,* The C++ Programming Language, 3rd Edition, Addison Wesley, 1997.

[7] *Ken Arnold, James Gosling,* The Java Programming Language, 2nd Edition, Addison Wesley, 1998.

[8] *Larry Wall, Tom Christiansen, Randal Schwartz,* Programming Perl, 2nd Edition, O'Reilly, 1996.

[9] *Erich Gamma, Richard Helm, Ralph Johnson,* Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1995.

[10] *John Lakos,* Large-Scale C++ Software Design, Addison Wesley, 1996.

[11] *David Hanson,* C Interfaces and Implementations, Addison Wesley, 1997.

[12] *Steve McConnell,* Rapid Development, Microsoft Press, 1996.

[13] *Kevin Mullet, Darrell Sano,* Designing Visual Interfaces: Communication Oriented Techniques, Prentice Hall, 1995.

[14] *Ben Shneiderman,* Designing the User Interface: Strategies for Effective Human-Computer Interaction, 3rd Addison Wesley, 1997.

[15] *Steve Maguire*, Writing Solid Code, Microsoft Press, 1993.

[16] *Steve McConnell*, Code Complete, Microsoft Press, 1993.

[17] *Jon Bentley*, Communications of the ACM, Addison Wesley, 1986.

[18] *John Hennessy, David Patterson,* Computer Organization and Design: The Hardware/Software Interface, Morgan Kaufman, 1997.

[19] *James Gosling, Bill Joy, Guy Steele,* The Java Language Specification, Addison Wesley, 1996.

[20] *Rich Steven,* Advanced Programming in the Unix Environment, Addison Wesley, 1992.

[21] *Sean Dorward, Rob Pike, David Leo Presotto, Dennis M. Ritchie, Howard W. Trickey*, The Inferno Operating System, Bell Labs Technical Journal, 2,1, Winter, 1997.

[22] *Brian Kernighan, Rob Pike,* The Unix Programming Environment, Prentice Hall, 1984.

[23] *Chris Fraser, David Hanson*, A Retargetable C Compiler: Design and Implementation, Addison Wesley, 1995.

[24] *Tim Lindholm, Frank Yellin,* The Java Virtual Machine Specification, 2nd Edition, Addison Wesley, 1999.

[25] *Brian W. Kernighan, Rob Pike,* The Practice of Programming, Addison Wesley, 1999.

KỸ NĂNG LẬP TRÌNH

Tác giả : Lê Hoài Bắc

Nguyễn Thanh Nghị

Chịu trách nhiệm xuất bản : PGS. TS. TÔ ĐĂNG HẢI

Biên tập và sửa bài : ThS. NGUYỄN HUY TIẾN

NGỌC LINH

Trình bày bìa : HƯƠNG LAN

NHÀ XUẤT BẢN KHOA HỌC VÀ KỸ THUẬT

70, TRẦN HƯNG ĐẠO – HÀ NỘI

In 700 cuốn, khổ 16x24cm, tại Nhà in KH&CN
Giấy phép xuất bản số: 06-287-30/12/2004
In xong và nộp lưu chiểu tháng 8 năm 2005

205189

Giá: 54.000đ