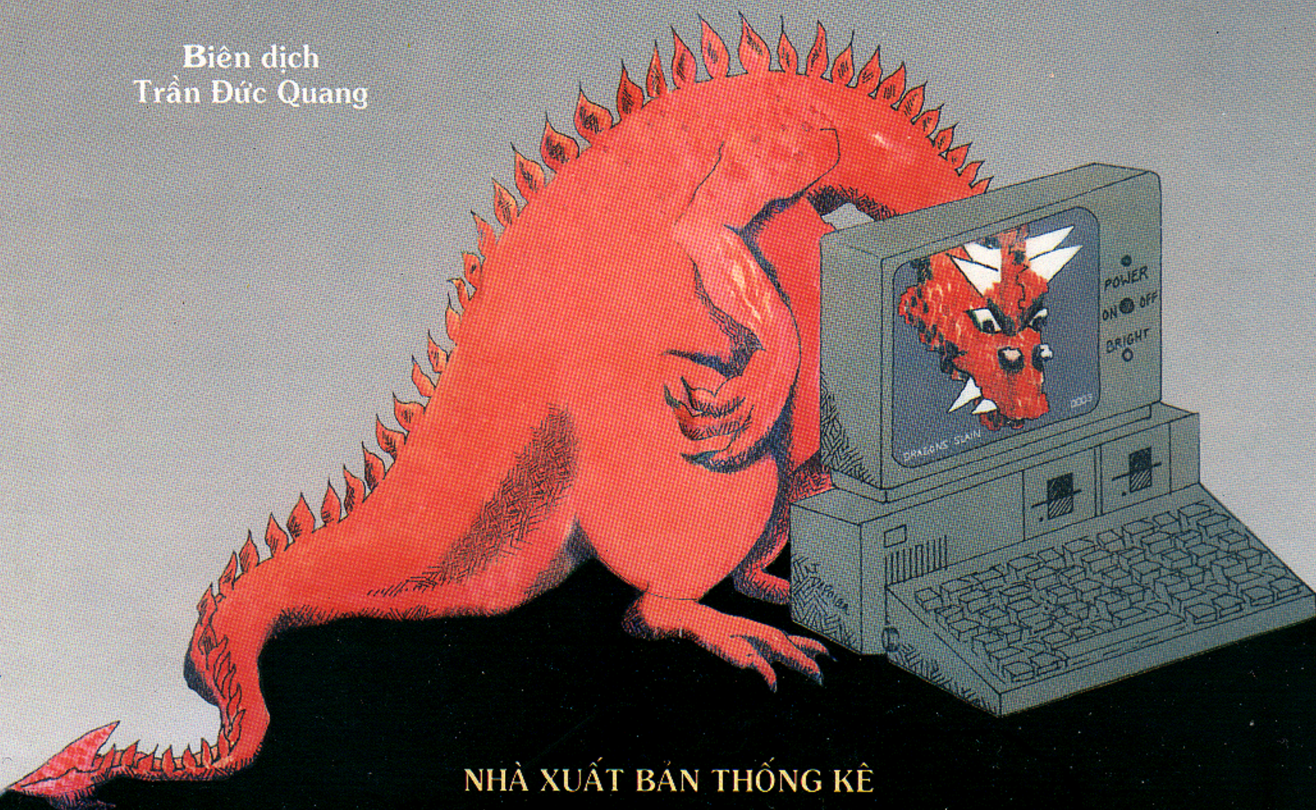
Trình Biên Dịch

NGUYÊN LÝ
KỸ THUẬT & CÔNG CỤ

Tập 1

Alfred V. Aho
Ravi Sethi
Jeffrey D. Ullman

Biên dịch
Trần Đức Quang



NHÀ XUẤT BẢN THỐNG KÊ

Mục Lục

Chương 1 Tổng Quan Về Biên Dịch **1**

1.1 Trình biên dịch 1
1.2 Phân tích chương trình nguồn 6
1.3 Các giai đoạn biên dịch 11
1.4 Anh em của trình biên dịch 17
1.5 Nhóm các giai đoạn 22
1.6 Công cụ xây dựng trình biên dịch 24
Ghi chú về tài liệu tham khảo 26

Chương 2 Một Trình Biên Dịch Một Lượt Đơn Giản **29**

2.1 Tổng quan 29
2.2 Định nghĩa cú pháp 30
2.3 Phiên dịch dựa cú pháp 38
2.4 Phân tích cú pháp 46
2.5 Một chương trình dịch cho các biểu thức đơn giản 54
2.6 Phân tích từ vựng 62
2.7 Kết hợp với bảng ký hiệu 66
2.8 Máy chồng xếp trừu tượng 70
2.9 Kết nối các kỹ thuật 77
Bài tập 87
Bài tập lập trình 90
Ghi chú về tài liệu tham khảo 91

Chương 3 Phân Tích Từ Vựng 93

3.1 Vai trò của thể phân từ vựng ... 94
3.2 Đệm nguyên liệu ... 99
3.3 Đặc tả các thẻ từ ... 102
3.4 Nhận dạng các thẻ từ ... 109
3.5 Một ngôn ngữ đặc tả thể phân từ vựng ... 118
3.6 Automat hữu hạn ... 126
3.7 Biến đổi biểu thức chính qui thành NFA ... 134
3.8 Thiết kế bộ sinh thể phân từ vựng ... 142
3.9 Tối ưu hóa thể so mẫu dựa trên DFA ... 148
Bài tập ... 161
Bài tập lập trình ... 172
Ghi chú về tài liệu tham khảo ... 172

Chương 4 Phân Tích Cú Pháp 175

4.1 Vai trò của thể phân cú pháp ... 176
4.2 Văn phạm phi ngữ cảnh ... 182
4.3 Xây dựng văn phạm ... 189
4.4 Phân tích cú pháp từ trên xuống ... 199
4.5 Phân tích cú pháp từ dưới lên ... 214
4.6 Phân tích cú pháp thứ bậc toán tử ... 222
4.7 Thể phân cú pháp LR ... 236
4.8 Sử dụng các văn phạm đa nghĩa ... 271
4.9 Bộ sinh thể phân cú pháp ... 281
Bài tập ... 291
Ghi chú về tài liệu tham khảo ... 302

Chương 5 Phiên Dịch Dựa Cú Pháp **305**

5.1 Định nghĩa dựa cú pháp ... 306
5.2 Xây dựng cây cú pháp ... 313
5.3 Ước lượng các định nghĩa thuần tính S theo lối từ dưới lên ... 320
5.4 Định nghĩa thuần tính L ... 323
5.5 Phiên dịch từ trên xuống ... 329
5.6 Ước lượng các thuộc tính kế thừa theo lối từ dưới lên ... 336
5.7 Thể ước lượng đệ qui ... 344
5.8 Không gian dành cho giá trị thuộc tính vào lúc biên dịch ... 347
5.9 Dành không gian vào lúc xây dựng trình biên dịch ... 351
5.10 Phân tích các định nghĩa dựa cú pháp ... 358
Bài tập ... 365
Ghi chú về tài liệu tham khảo ... 369

Danh Mục Tài Liệu Tham Khảo **373**

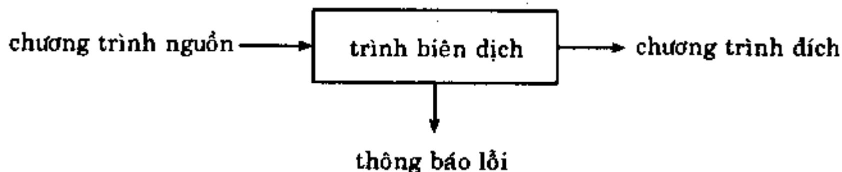
CHƯƠNG 1

Tổng Quan Về Biên Dịch

Các nguyên tắc và kỹ thuật xây dựng *trình biên dịch* (compiler) hiện đã thông dụng đến nỗi các ý tưởng trong cuốn sách này đã được sử dụng rất nhiều trong hoạt động nghề nghiệp của một nhà khoa học máy tính. Việc xây dựng trình biên dịch đòi hỏi phải hiểu biết nhiều lãnh vực: ngôn ngữ lập trình, kiến trúc máy tính, lý thuyết ngôn ngữ, các thuật toán và công nghệ phần mềm. Rất may là chúng ta có thể dùng một số ít các kỹ thuật cơ bản viết trình biên dịch để xây dựng các *chương trình dịch* (translator) cho rất nhiều ngôn ngữ và nhiều loại máy tính. Trong chương này, chúng tôi sẽ giới thiệu về vấn đề biên dịch bằng cách mô tả các thành phần của một trình biên dịch, môi trường hoạt động của trình biên dịch và một số công cụ phần mềm có khả năng hỗ trợ xây dựng trình biên dịch.

1.1 TRÌNH BIÊN DỊCH

Nói đơn giản, trình biên dịch là một chương trình làm nhiệm vụ đọc một chương trình được viết bằng một ngôn ngữ — *ngôn ngữ nguồn* (source language) — rồi dịch nó thành một chương trình tương đương ở một ngôn ngữ khác — *ngôn ngữ đích* (target language) (xem Hình 1.1). Một phần quan trọng trong quá trình dịch là ghi nhận các lỗi trong chương trình nguồn để thông báo lại cho người viết chương trình.



Hình 1.1. Một trình biên dịch.

Chỉ nhìn thoáng qua, số lượng trình biên dịch dường như quá nhiều. Có đến hàng ngàn ngôn ngữ nguồn, từ các ngôn ngữ lập trình truyền thống như Fortran và Pascal đến các ngôn ngữ chuyên dụng dành riêng cho từng lãnh vực ứng dụng khác nhau. Ngôn ngữ đích cũng đa dạng như thế; một ngôn ngữ đích có thể là một ngôn ngữ lập trình khác, hoặc là một ngôn ngữ máy của một loại máy tính, từ máy tính PC cho đến *siêu máy tính* (supercomputer). Trình biên dịch đôi khi cũng được phân thành các loại như loại *một lượt* (single-pass), loại *nhiều lượt* (multi-pass), loại *nạp và tiến hành* (load-and-go), loại *gỡ rối* (debugging) hoặc loại *tối ưu hóa* (optimizing), tùy vào cách thức chúng ta xây dựng hoặc tùy vào chức năng mà chúng thực hiện. Mặc dù tính chất phức tạp, nhiệm vụ cơ bản của mọi trình biên dịch đều như nhau. Nhờ hiểu rõ được nhiệm vụ này mà chúng ta có thể xây dựng trình biên dịch cho rất nhiều loại ngôn ngữ nguồn và các máy đích bằng cách sử dụng các kỹ thuật cơ bản giống nhau.

Hiểu biết của chúng ta về cách thức tổ chức và viết các trình biên dịch đã tiến rất xa kể từ khi trình biên dịch đầu tiên xuất hiện vào đầu thập niên 50. Thực ra rất khó xác định chính xác ngày tháng ra đời của trình biên dịch đầu tiên bởi vì có rất nhiều nhóm thử nghiệm và cài đặt thực hiện độc lập nhau. Phần lớn các thử nghiệm đầu tiên đều giải quyết công việc dịch các công thức số học thành các mã máy.

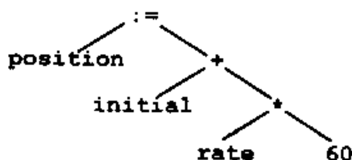
Trong suốt thập niên 50, trình biên dịch đã được xem là những chương trình cực kỳ khó viết. Chẳng hạn như việc cài đặt trình biên dịch Fortran đầu tiên đã phải mất hết 18 năm làm việc của một nhóm (Backus et al. [1957]). Từ đó chúng ta đã dần khám phá ra các kỹ thuật mang tính hệ thống để xử lý nhiều nhiệm vụ quan trọng xảy ra trong quá trình biên dịch. Một số ngôn ngữ để cài đặt, các môi trường lập trình và các công cụ phần mềm cũng được phát triển. Với những thành quả này, việc viết một trình biên dịch có chất lượng hiện có thể được dùng làm đề tài cho sinh viên trong một khóa học một học kỳ về thiết kế trình biên dịch.

Mô hình biên dịch phân tích-tổng hợp

Công việc biên dịch có thể được chia thành hai phần: phân tích và tổng hợp. Phần phân tích sẽ phân rã chương trình nguồn thành các phần cấu thành và tạo ra một dạng biểu diễn trung gian cho chương trình nguồn. Phần tổng hợp sẽ xây dựng ngôn ngữ đích từ dạng biểu diễn trung gian này. Trong hai phần này thì phần tổng hợp đòi hỏi những kỹ thuật đặc biệt. Chúng ta sẽ xem xét quá trình phân tích một cách không hình thức trong Phần 1.2 và phác thảo cách thức tổng hợp mã đích trong một trình biên dịch chuẩn ở Phần 1.3.

Trong quá trình phân tích, các phép toán mà chương trình nguồn phải thực hiện sẽ được xác định và ghi lại trong một cấu trúc phân cấp gọi là một cây. Thông thường người ta sử dụng một loại cây gọi là *cây cú pháp* (syntax tree), trong đó mỗi nút biểu thị cho một phép toán, các con của nút biểu thị cho các đối của phép toán. Một thí dụ

về cây cú pháp cho một câu lệnh gán được trình bày trong Hình 1.2.



Hình 1.2. Cây cú pháp cho `position := initial + rate*60`.

Nhiều công cụ phần mềm hoạt tác trên các chương trình nguồn trước tiên cần phải thực hiện phân tích chúng. Dưới đây là một số thí dụ về những công cụ như thế:

1. *Trình soạn thảo cấu trúc* (structure editor). Trình soạn thảo cấu trúc nhận *nguyên liệu* (input) là một chuỗi lệnh để xây dựng chương trình. Trình soạn thảo cấu trúc không chỉ lo tạo văn bản và hiệu chỉnh văn bản như một trình soạn thảo văn bản thông thường mà còn phân tích đoạn mã nguồn, xây dựng một cấu trúc phân cấp thích hợp trên chương trình nguồn. Vì thế trình soạn thảo cấu trúc có thể thực hiện được nhiều tác vụ hữu ích trong quá trình viết các chương trình. Chẳng hạn nó có thể xác nhận rằng nguyên liệu được tạo ra là chính xác, có thể cung cấp các từ khóa một cách tự động (thí dụ khi người sử dụng gõ **while**, nó tự động điền thêm **do** và nhắc người sử dụng rằng phải có một điều kiện giữa chúng) hoặc có thể nhảy từ vị trí **begin** hoặc dấu ngoặc mở đến vị trí **end** tương ứng hoặc dấu ngoặc đóng tương ứng. Ngoài ra *thành phẩm* (output) của trình soạn thảo cấu trúc thường tương tự như thành phẩm trong giai đoạn phân tích của trình biên dịch.
2. *Trình trang trí* (pretty printer). Trình trang trí sẽ phân tích chương trình và trình bày sao cho cấu trúc của chương trình dễ đọc hơn. Chẳng hạn *phần giải thích* (comment) có thể được hiển thị bằng một font chữ riêng, các câu lệnh có thể được hiển thị lùi vào nhiều mức tương ứng với mức độ lồng trong tổ chức phân cấp của câu lệnh.
3. *Trình kiểm lỗi tĩnh* (static checker). Trình kiểm lỗi tĩnh sẽ đọc chương trình, phân tích rồi cố gắng phát hiện các lỗi ẩn mà không cho chạy chương trình. Bộ phận phân tích thường giống như bộ phận tối ưu hóa trình biên dịch thuộc loại được thảo luận trong Chương 10. Chẳng hạn trình kiểm lỗi tĩnh có thể phát hiện ra rằng một số phần của chương trình nguồn có thể không bao giờ được thực hiện hoặc một biến được sử dụng trước khi được định nghĩa. Ngoài ra nó còn có thể phát hiện các lỗi logic như sử dụng một biến số thực làm con trỏ nhờ kỹ thuật kiểm tra kiểu sẽ được thảo luận trong Chương 6.
4. *Trình thông dịch* (interpreter). Thay vì dịch thành một chương trình đích, trình thông dịch sẽ thực hiện các phép toán được đưa ra trong chương trình nguồn.

Chẳng hạn đối với một câu lệnh gán, trình thông dịch có thể xây dựng một cây giống như trong Hình 1.2 rồi thực hiện các phép toán tại các nút khi nó "tản bộ" qua cây. Ở tại nút gốc, nó phát hiện ra rằng cần thực hiện một phép gán, do vậy nó có thể gọi một *thủ tục* (routine) để ước lượng biểu thức bên phải, lưu kết quả ở vị trí đi kèm với định danh `position`. Tại nút con bên phải, thủ tục này khám phá ra rằng nó phải tính tổng của hai biểu thức. Nó sẽ gọi đệ qui chính nó để tính giá trị của biểu thức `rate*60`. Sau đó nó sẽ cộng giá trị thu được với giá trị của biến `initial`.

Trình thông dịch thường được dùng thực thi các ngôn ngữ kiểu câu lệnh bởi vì mỗi toán tử được thực hiện trong ngôn ngữ này thường là phần khởi động của một chương trình phức tạp như soạn thảo hoặc biên dịch. Tương tự, một số ngôn ngữ "cấp rất cao" như APL hay được thông dịch vì có rất nhiều thông tin về dữ liệu như kích thước và chiều của *mảng* (array) không được biết vào lúc biên dịch.

Theo truyền thống, chúng ta xem trình biên dịch như một chương trình làm nhiệm vụ dịch ngôn ngữ nguồn như Fortran sang ngôn ngữ máy hay *hợp ngữ* (assembly) của một loại máy tính nào đó. Tuy nhiên cũng có nhiều khi công nghệ biên dịch được sử dụng trong các lãnh vực dường như không có liên hệ gì với nó. Trong các thí dụ dưới đây, thành phần phân tích tương tự như thành phần phân tích của một trình biên dịch thông thường.

1. *Trình định dạng văn bản* (text formatter). Trình định dạng văn bản nhận chuỗi ký tự, phần lớn đều là văn bản cần định dạng nhưng cũng có thể có một số lệnh biểu thị *đoạn văn bản* (paragraph), *hình ảnh* (figure) hoặc các cấu trúc toán học như *số mũ* (superscript), *cước số* (subscript). Trong phần tiếp theo chúng tôi sẽ đề cập đến một số phần trong quá trình phân tích do trình định dạng văn bản thực hiện.

2. *Trình biên dịch silicon* (silicon compiler). Trình biên dịch silicon nhận ngôn ngữ nguồn là một ngôn ngữ lập trình thông thường hoặc gần như thế. Tuy nhiên các biến của ngôn ngữ này không biểu thị các vị trí trong bộ nhớ mà biểu thị các tín hiệu logic (0 hoặc 1) hoặc các nhóm tín hiệu trong một *bo chuyển mạch* (switching circuit). Kết xuất là một bản thiết kế mạch trong một ngôn ngữ thích hợp. Hãy xem thêm Johnson [1983], Ullman [1984] hoặc Trickey [1985] về quá trình biên dịch silicon.

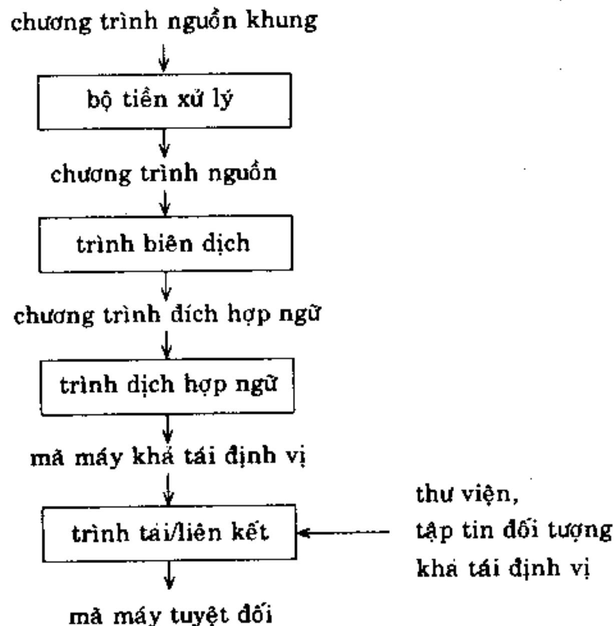
3. *Trình thông dịch câu vấn tin* (query interpreter). Trình thông dịch câu vấn tin sẽ dịch một *vị từ* (predicate) có chứa toán tử quan hệ và logic thành các lệnh để tìm kiếm các mẫu tin thỏa vị từ đó có trong cơ sở dữ liệu. (Xem Ulman [1988] hoặc Date [1986]).

Bối cảnh biên dịch

Ngoài trình biên dịch, chúng ta còn cần dùng nhiều chương trình khác nữa để có thể tạo ra một chương trình đích *chạy được* (executable). Một chương trình nguồn có thể được phân thành nhiều *đơn thể* (module) và được lưu trong các tập tin riêng rẽ. Công việc tập hợp lại các tập tin nguồn đôi khi được giao cho một chương trình riêng biệt gọi là *bộ tiền xử lý* (preprocessor). Bộ tiền xử lý cũng có thể "bung" các ký hiệu tắt được gọi là các macro thành các câu lệnh của ngôn ngữ nguồn.

Hình 1.3 trình bày một quá trình biên dịch điển hình. Chương trình đích được tạo bởi trình biên dịch có thể cần phải được xử lý thêm trước khi chúng có thể chạy được. Trình biên dịch trong Hình 1.3 tạo ra *mã hợp ngữ* (assembly code) để *trình dịch hợp ngữ* (assembler) dịch thành mã máy rồi được liên kết với một số thủ tục trong thư viện thành các mã chạy được trên máy.

Chúng ta sẽ xét các thành phần của một trình biên dịch trong hai phần tiếp theo; các chương trình còn lại trong Hình 1.3 sẽ được thảo luận trong Phần 1.4.



Hình 1.3. Một hệ thống xử lý ngôn ngữ.

1.2 PHÂN TÍCH CHƯƠNG TRÌNH NGUỒN

Trong phần này chúng ta giới thiệu về *quá trình phân tích* (analysis) và minh họa cách dùng nó qua các ngôn ngữ định dạng văn bản. Đề tài này sẽ được phân tích chi tiết hơn trong các Chương 2–4 và 6. Khi biên dịch, quá trình phân tích bao gồm ba giai đoạn:

1. *Phân tích tuyến tính* (linear analysis), trong đó các dòng ký tự tạo ra chương trình nguồn sẽ được đọc từ trái sang phải và được nhóm lại thành các *thẻ từ* (token), đó là các chuỗi ký tự được hợp lại để tạo ra một nghĩa chung.
2. *Phân tích cấu trúc phân cấp* (hierarchical analysis) trong đó các ký tự hoặc các thẻ từ được nhóm thành các nhóm lồng nhau theo kiểu phân cấp để tạo ra một nghĩa chung.
3. *Phân tích ngữ nghĩa* (semantic analysis) trong đó nó thực hiện một số kiểm tra để bảo đảm rằng các thành phần của chương trình kết lại với nhau một cách có nghĩa.

Phân tích từ vựng

Trong một trình biên dịch, phân tích tuyến tính được gọi là *phân tích từ vựng* (lexical analysis) hay hành động *quét nguyên liệu* (scanning). Chẳng hạn khi phân tích từ vựng, các ký tự của câu lệnh gán

```
position := initial + rate*60
```

sẽ được nhóm lại thành các thẻ từ sau:

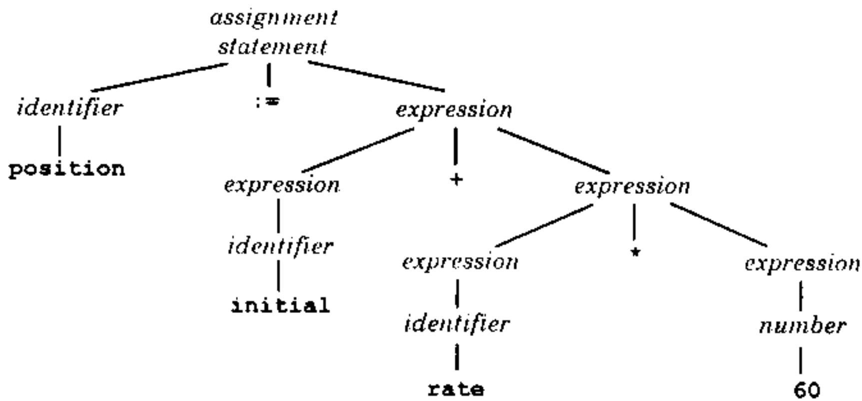
1. Định danh `position`
2. Ký hiệu gán `:=`
3. Định danh `initial`
4. Dấu cộng (+)
5. Định danh `rate`
6. Dấu nhân
7. Số 60

Các *khoảng trống* (blank) phân cách các ký tự của những thẻ từ này sẽ được loại bỏ trong quá trình phân tích từ vựng.

Phân tích cú pháp

Phân tích cấu trúc phân cấp được gọi là *phân tích cú pháp* (syntax analysis hay thường được gọi là parsing). Giai đoạn này thực hiện công việc nhóm các thẻ từ của

chương trình nguồn thành các *ngữ đoạn văn phạm* (grammatical phrase) sẽ được trình biên dịch tổng hợp ra thành phẩm. Thông thường các ngữ đoạn văn phạm của chương trình nguồn được biểu diễn bằng *cây phân tích cú pháp* (parse tree) giống như cây được trình bày trong Hình 1.4.



Hình 1.4. Cây phân tích cú pháp cho câu lệnh `position := initial + rate*60`.

Trong biểu thức `initial + rate*60`, ngữ đoạn `rate*60` là một đơn vị logic vì theo qui ước thông thường cho các biểu thức số học, phép nhân được thực hiện trước phép cộng. Bởi vì sau biểu thức `initial + rate` là dấu * nên nó sẽ không được nhóm lại thành một ngữ đoạn ở Hình 1.4.

Cấu trúc phân cấp của một chương trình thường được biểu diễn nhờ các qui tắc đệ qui. Chẳng hạn các qui tắc sau có thể là thành phần trong định nghĩa *biểu thức* (expression):

1. Một *định danh* (identifier) là một biểu thức.
2. Một *con số* (number) là một biểu thức.
3. Nếu $expression_1$ và $expression_2$ là các biểu thức thì

   $expression_1 + expression_2$
   $expression_1 * expression_2$
   $(expression_1)$

   đều là các biểu thức

Qui tắc (1) và (2) là các qui tắc cơ sở (không đệ qui) còn qui tắc (3) định nghĩa các biểu thức theo các toán tử được áp dụng cho các biểu thức khác. Vì thế theo qui tắc (1), `initial` và `rate` là các biểu thức. Theo qui tắc (2), `60` là một biểu thức trong khi đó theo qui tắc (3), đầu tiên có thể suy ra rằng `rate*60` là một biểu thức rồi cuối cùng kết

luận `initial + rate*60` cũng là một biểu thức.

Tương tự, nhiều ngôn ngữ định nghĩa các *câu lệnh* (statement) một cách đệ qui bằng các qui tắc như:

1. Nếu $identifier_1$ là một định danh và $expression_2$ là một biểu thức thì

   $identifier_1$:= $expression_2$

   là một câu lệnh.

2. Nếu $expression_1$ là một biểu thức và $statement_2$ là một câu lệnh thì

   while ($expression_1$) **do** $statement_2$
   if ($expression_1$) **then** $statement_2$

   là các câu lệnh.

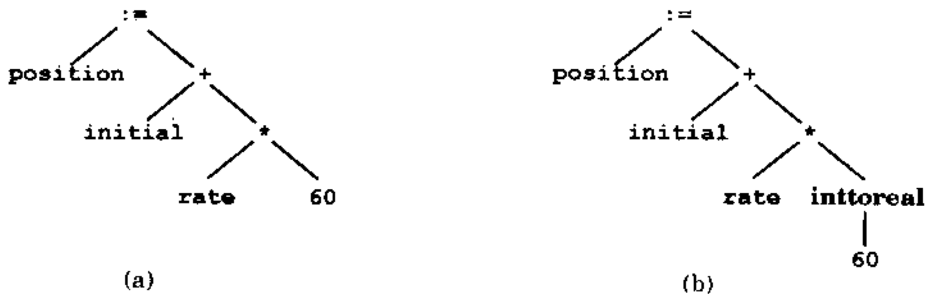
Phân chia giữa giai đoạn phân tích từ vựng và phân tích cú pháp cũng hết sức tùy ý. Chúng tôi thường chọn cách phân chia có thể đơn giản hóa toàn bộ công việc phân tích. Một yếu tố cần xét đến khi phân chia đó là xem bản thân các *kết cấu* (construct) của ngôn ngữ nguồn có đệ qui hay không. Kết cấu từ vựng không đòi hỏi đệ qui, trong khi đó các kết cấu cú pháp thường cần đệ qui. *Văn phạm phi ngữ cảnh* (context-free grammar) là sự hình thức hóa cho các qui tắc đệ qui, có thể được sử dụng để hướng dẫn quá trình phân tích cú pháp. Chúng sẽ được giới thiệu trong Chương 2 và được nghiên cứu kỹ lưỡng trong Chương 4.

Thí dụ như chúng ta không cần dùng đệ qui nhận diện các định danh bởi vì thường thì đó là chuỗi các chữ cái và ký số bắt đầu bằng một chữ cái. Bình thường chúng ta nhận diện các định danh bằng cách chỉ cần quét qua chuỗi nguyên liệu cho đến lúc gặp một ký tự không phải là chữ cái cũng không phải là ký số, rồi nhóm tất cả các chữ cái và ký số vừa nhận được cho đến lúc đó thành một thẻ từ của định danh. Các ký tự được nhóm lại như thế sẽ được ghi vào trong một bảng được gọi là *bảng ký hiệu* (symbol table) và được lấy ra khỏi nguyên liệu rồi tiếp tục xử lý thẻ từ kế tiếp.

Ngược lại thì kiểu quét tuyến tính này không đủ mạnh để phân tích các biểu thức hoặc các câu lệnh. Chẳng hạn chúng ta không thể đối sánh được các dấu ngoặc trong các biểu thức, hoặc đối sánh giữa **begin** và **end** trong các câu lệnh mà không phải đưa ra một loại cấu trúc phân cấp hoặc cấu trúc lồng trên nguyên liệu.

Cây phân tích cú pháp trong Hình 1.4 mô tả cấu trúc cú pháp của dòng nguyên liệu. Một dạng biểu diễn nội tại thông dụng hơn của cấu trúc cú pháp này được trình bày trong *cây cú pháp* (syntax tree) của Hình 1.5(a). Cây cú pháp là một dạng biểu diễn thu gọn của cây phân tích cú pháp, trong đó các toán tử xuất hiện như các nút nội và các toán hạng của một toán tử là các con của nút toán tử đó. Việc xây dựng những cây như trong Hình 1.5(a) được thảo luận trong Phần 5.2. Chúng ta sẽ nghiên cứu về

quá trình dịch dựa cú pháp (syntax-directed translation) trong Chương 2 và chi tiết hơn trong Chương 5, trong đó trình biên dịch sử dụng cấu trúc phân cấp của nguyên liệu để tạo ra thành phẩm.



Hình 1.5. Phân tích ngữ nghĩa sẽ chèn thêm bước đổi số nguyên thành số thực.

Phân tích ngữ nghĩa

Giai đoạn phân tích ngữ nghĩa sẽ kiểm tra các lỗi ngữ nghĩa của chương trình nguồn và thu nhận các thông tin về kiểu cho giai đoạn tạo mã tiếp theo. Giai đoạn này sử dụng cấu trúc phân cấp được xác định trong giai đoạn phân tích cú pháp để xác định toán tử và toán hạng của các biểu thức và câu lệnh.

Một phần quan trọng trong giai đoạn phân tích ngữ nghĩa là *kiểm tra kiểu* (type checking). Ở đây trình biên dịch sẽ kiểm tra, dựa theo đặc tả của ngôn ngữ nguồn, xem các toán hạng của một toán tử có hợp lệ hay không. Chẳng hạn nhiều định nghĩa của các ngôn ngữ lập trình yêu cầu trình biên dịch ghi nhận lỗi mỗi khi một số thực được sử dụng làm chỉ mục cho một mảng. Tuy nhiên đặc tả của ngôn ngữ có thể cho phép việc *áp đặt toán hạng* (operand coercion), thí dụ như khi một *toán tử số học hai ngôi* (binary arithmetic operator) được áp dụng cho một số nguyên và một số thực. Trong trường hợp này, trình biên dịch có thể phải chuyển số nguyên thành số thực. Kiểm tra kiểu và phân tích ngữ nghĩa được thảo luận trong Chương 6 (Tập II).

Thí dụ 1.1. Bên trong máy, mẫu bit biểu diễn số nguyên nói chung khác với mẫu bit biểu diễn số thực, ngay cả khi số nguyên và số thực có cùng giá trị. Giả sử rằng tất cả các định danh trong Hình 1.5 đã được khai báo là số thực và bản thân 60 lại là số nguyên. Việc kiểm tra kiểu của Hình 1.5(a) phát hiện ra rằng phép nhân * được áp dụng cho một số thực `rate` và một số nguyên 60. Cách tiếp cận thông thường là chuyển số nguyên thành số thực. Điều này được thực hiện trong Hình 1.5(b) bằng cách tạo ra một nút "bổ sung" dành cho toán tử **inttoreal**, với tác dụng là đổi số nguyên thành số thực. Một cách khác, bởi vì toán hạng của **inttoreal** là một hằng, trình biên dịch có thể thay một hằng nguyên bằng một hằng thực. □

Phân tích trong các chương trình định dạng văn bản

Chúng ta có thể xem việc đưa nguyên liệu vào một trình định dạng văn bản là việc xác định một cây phân cấp cho các *hộp* (box), đó là các vùng hình chữ nhật có chứa các mẫu bit, biểu diễn các điểm sáng và tối cần được in ra bằng các thiết bị xuất.

Thí dụ, hệ thống TEX (Knuth [1984a]) xem xét nguyên liệu của nó theo cách này. Mỗi ký tự không phải là thành phần của một lệnh biểu diễn một hộp chứa mẫu bit cho ký tự đó theo một font chữ và kích thước thích hợp. Các ký tự liên tục không được phân cách bởi các *khoảng trắng* (white space), chẳng hạn như các *ký tự trống* (blank), *ký tự xuống dòng* (newline), được nhóm lại thành các *từ* (word), gồm một chuỗi các hộp được xếp ngang như trong Hình 1.6. Nhóm các ký tự thành các từ (hoặc lệnh) là khía cạnh từ vựng của việc phân tích trong một trình định dạng văn bản.

Các hộp trong TEX có thể được xây dựng từ các hộp nhỏ hơn bằng các tổ hợp dọc hoặc ngang một cách tùy ý. Chẳng hạn,

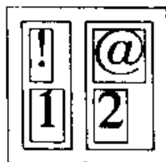
```
\hbox{ <list of boxes> }
```

nhóm các hộp lại bằng cách xếp chúng nằm ngang cạnh nhau trong khi đó toán tử **\vbox** nhóm một danh sách các hộp chồng lên nhau. Vì vậy trong TEX nếu chúng ta ghi

```
\hbox{ \vbox{ !  1}  \vbox{@  2}}
```

chúng ta sẽ có được kết quả như trong Hình 1.7. Việc xác định cách sắp xếp phân cấp của các hộp qua nguyên liệu là thành phần của quá trình phân tích trong TEX.

Hình 1.6. Nhóm các ký tự và các từ vào trong các hộp.



Hình 1.7. Cây phân cấp các hộp trong TEX.

Một thí dụ khác, bộ tiền xử lý toán học EQN (Kernighan and Cherry [1975]), hoặc trình xử lý toán học trong TEX xây dựng các biểu thức toán học từ các toán tử như **sub**

và **sup** tương ứng cho *cước số* hay *chỉ số dưới* (subscript) và *chỉ số mũ* (superscript). Nếu EQN gặp một đoạn văn bản nhập liệu có dạng

BOX **sub** *box*

nó sẽ thu nhỏ kích thước của *box* và gắn nó vào *BOX* gần góc dưới phải như được minh họa trong Hình 1.8. Tương tự, toán tử **sup** gắn *box* tại góc trên phải.

Hình 1.8. Xây dựng cấu trúc cước số trong các văn bản toán học.

Những toán tử này có thể được sử dụng đệ qui; chẳng hạn đoạn nguyên liệu EQN

```
a sub {i sup 2}
```

tạo ra a_{i^2}. Nhóm các toán tử **sub** và **sup** thành các thẻ từ là thành phần của quá trình phân tích từ vựng của EQN. Tuy nhiên cấu trúc cú pháp của văn bản cũng cần để xác định kích thước và vị trí đặt của một hộp.

1.3 CÁC GIAI ĐOẠN BIÊN DỊCH

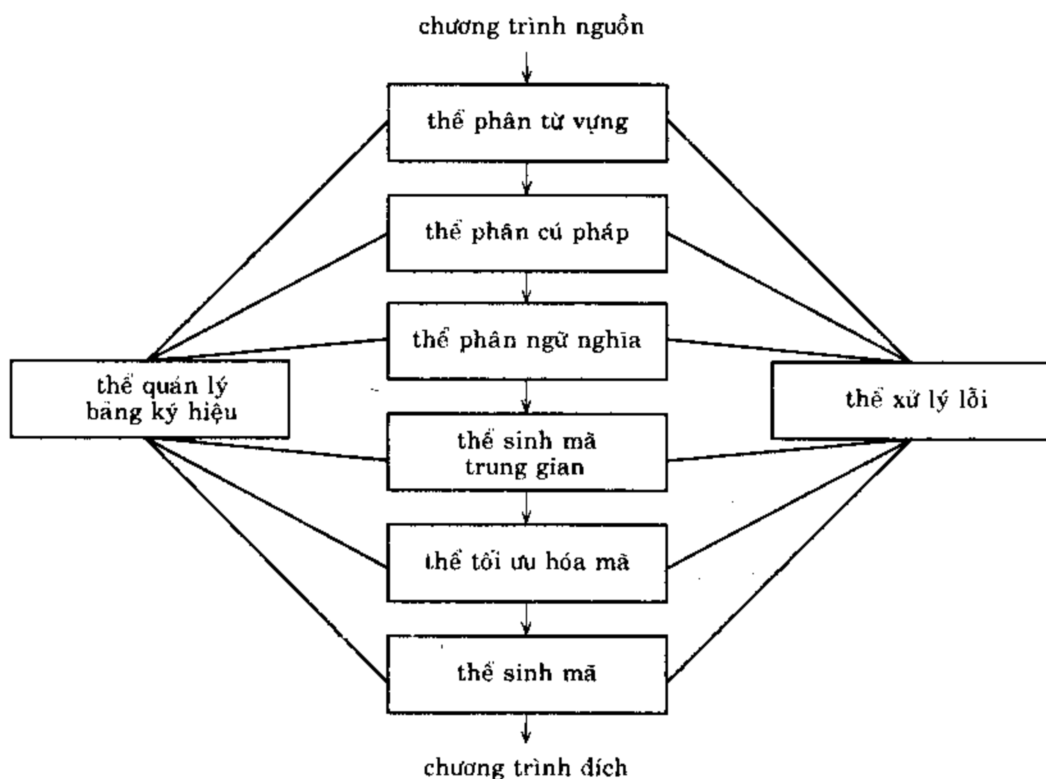
Về khái niệm, một trình biên dịch hoạt động theo từng giai đoạn, mỗi giai đoạn chuyển chương trình nguồn từ một dạng biểu diễn này sang một dạng biểu diễn khác. Một cách phân rã điển hình của một trình biên dịch được trình bày trong Hình 1.9. Trong thực tế, một số giai đoạn có thể được nhóm lại, như sẽ được nói đến trong Phần 1.5, và dạng biểu diễn trung gian giữa các giai đoạn được nhóm lại này không nhất thiết phải được xây dựng cụ thể.

Ba giai đoạn đầu tiên, đảm trách hết phần phân tích của trình biên dịch, đã được giới thiệu ở phần trước. Hai tác vụ khác là quản lý *bảng ký hiệu* (symbol table) và xử lý lỗi sẽ được trình bày xen kẽ với cả sáu giai đoạn, phân tích từ vựng, phân tích cú pháp, phân tích ngữ nghĩa, tạo mã trung gian, tối ưu hóa mã và phát sinh mã. Một cách không hình thức, chúng tôi cũng gọi đó là giai đoạn quản lý bảng ký hiệu và giai đoạn xử lý lỗi.

Quản lý bảng ký hiệu

Một nhiệm vụ quan trọng của trình biên dịch là ghi lại các định danh được sử dụng trong chương trình nguồn và thu thập thông tin về các thuộc tính khác nhau của mỗi

định danh. Những thuộc tính này có thể cung cấp các thông tin về vị trí lưu trữ được cấp phát cho một định danh, kiểu và tầm vực của định danh (là phạm vi chương trình mà định danh có giá trị), và nếu định danh là tên của thủ tục thì thuộc tính là các thông tin về số lượng và kiểu của các đối, phương pháp truyền đối (thí dụ truyền bằng tham trỏ) và kiểu trả về của thủ tục nếu có.



Hình 1.9. Các giai đoạn của một trình biên dịch.

Bảng ký hiệu (symbol table) là một cấu trúc dữ liệu chứa một mẫu tin dành cho mỗi định danh trong đó các trường được dành cho các thuộc tính của định danh. Cấu trúc dữ liệu này cho phép chúng ta tìm ra nhanh chóng mẫu tin của mỗi định danh và cũng có thể lưu trữ và truy xuất dữ liệu trong đó một cách nhanh chóng. Bảng ký hiệu sẽ được thảo luận trong Chương 2 và Chương 7.

Khi một định danh trong chương trình nguồn được *thể phân từ vựng* (lexical analyzer) phát hiện ra, nó sẽ đưa định danh này vào trong bảng ký hiệu. Tuy nhiên thông thường các thuộc tính của một định danh không thể xác định được trong giai đoạn

phân tích từ vựng. Chẳng hạn với một khai báo trong Pascal như

```
var position, initial, rate : real ;
```

thì khi nhận ra `position`, `initial` và `rate`, thể phân từ vựng chưa biết kiểu của chúng là số thực.

Các giai đoạn còn lại sẽ đưa thông tin về các định danh vào bảng ký hiệu rồi sử dụng thông tin này theo nhiều cách khác nhau. Chẳng hạn khi phân tích ngữ nghĩa và tạo mã trung gian, chúng ta cần biết kiểu của các định danh, nhờ đó có thể kiểm tra để biết rằng chương trình nguồn sử dụng đúng đắn và như vậy có thể tạo ra các thao tác phù hợp với chúng. *Thể sinh mã* (code generator) thường đưa các thông tin chi tiết về vị trí lưu trữ dành cho định danh và sử dụng chúng khi cần.

Phát hiện và ghi nhận lỗi

Mỗi giai đoạn đều có thể gặp các lỗi. Tuy nhiên sau khi phát hiện ra lỗi, mỗi giai đoạn phải có cách xử lý lỗi để có thể tiếp tục biên dịch, và như thế cho phép phát hiện thêm nhiều lỗi khác trong chương trình nguồn. Một trình biên dịch cứ phải dừng lại khi phát hiện lỗi sẽ không hữu ích lắm.

Giai đoạn phân tích cú pháp và ngữ nghĩa thường xử lý một phần khá lớn các lỗi được trình biên dịch phát hiện. Giai đoạn phân tích từ vựng có thể phát hiện các lỗi trong đó các ký tự còn lại trong phần nguyên liệu không thể tạo ra một thẻ từ của ngôn ngữ đang dùng. Các lỗi do chuỗi thẻ từ vi phạm các qui tắc cấu trúc (cú pháp) sẽ do giai đoạn phân tích cú pháp dò tìm. Trong giai đoạn phân tích ngữ nghĩa, trình biên dịch sẽ cố gắng phát hiện các *kết cấu* (construct) không có ý nghĩa đối với thao tác được thực hiện dù rằng chúng hoàn toàn đúng về mặt cú pháp, thí dụ như trường hợp chúng ta cho cộng hai định danh, một là tên của một mảng, còn một là tên của một thủ tục. Chúng ta sẽ thảo luận quá trình xử lý lỗi của mỗi giai đoạn trong phần thảo luận tương ứng của từng giai đoạn.

Các giai đoạn phân tích

Khi quá trình dịch đang tiến hành, dạng thức biểu diễn nội tại của chương trình nguồn trong trình biên dịch sẽ thay đổi. Chúng tôi sẽ minh họa các dạng thức biểu diễn này bằng cách xét quá trình dịch câu lệnh:

```
position := initial + rate * 60                                    (1.1)
```

Hình 1.10 trình bày dạng thức biểu diễn của câu lệnh này sau mỗi giai đoạn.

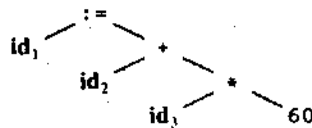
Giai đoạn phân tích từ vựng đọc các ký tự trong chương trình nguồn, nhóm chúng lại thành các *thẻ từ* (token), mỗi thẻ từ biểu diễn một chuỗi ký tự liên đới cạnh nhau như một định danh, một từ khóa (`if`, `while`, vân vân), một ký tự phân cách hoặc một toán tử nhiều ký tự như `:=`. Chuỗi ký tự tạo ra một thẻ từ được gọi là *từ tố* (lexeme).

```
position := initial + rate * 60
```

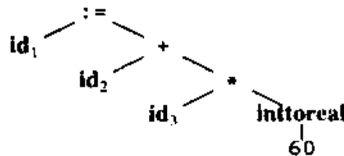
lexical analyzer

id_1 := id_2 + id_3 * 60

syntax analyzer



semantic analyzer



SYMBOL TABLE

1	position	...
2	initial	...
3	rate	...
4		

intermediate code generator

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

code optimizer

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```

code generator

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

Hình 1.10. Quá trình dịch một câu lệnh.

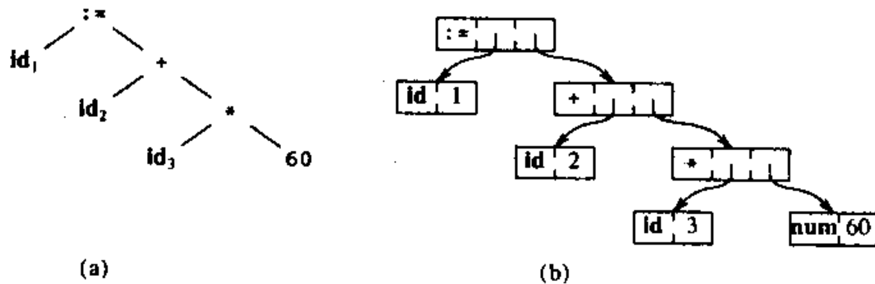
Nhiều thẻ từ còn được bổ sung một giá trị đi kèm gọi là *trị từ* (lexical value). Chẳng hạn khi phát hiện một định danh như **rate**, *thể phân từ vựng* (lexical analyzer) không chỉ tạo ra một thẻ từ (chẳng hạn là **id**) nhưng còn phải nhập từ tố **rate** vào trong bảng ký hiệu nếu nó chưa có trong bảng. Giá trị từ tố đi kèm với thẻ từ **id** này chỉ đến *mục ghi* (entry) của **rate** trong bảng ký hiệu.

Trong phần này, chúng ta sử dụng các ký hiệu $\mathbf{id}_1$, $\mathbf{id}_2$ và $\mathbf{id}_3$ tương ứng biểu thị cho **position**, **initial** và **rate** để nhấn mạnh rằng dạng thức biểu diễn nội tại của một định danh khác với chuỗi ký tự tạo ra định danh này. Dạng thức biểu diễn của (1.1) sau giai đoạn phân tích từ vựng có thể như sau:

$$\mathbf{id}_1 := \mathbf{id}_2 + \mathbf{id}_3 * 60 \tag{1.2}$$

Chúng ta cũng phải tạo các thẻ từ cho toán tử := và số 60 để phản ánh đúng dạng thức biểu diễn nội tại của chúng nhưng tạm để lại phần này cho Chương 2. Phân tích từ vựng được giới thiệu chi tiết trong Chương 3.

Giai đoạn thứ hai và thứ ba, là giai đoạn phân tích cú pháp và phân tích ngữ nghĩa cũng đã giới thiệu trong Phần 1.2. Phân tích cú pháp xây dựng một cấu trúc cây qua chuỗi thẻ từ. Chúng ta sẽ mô tả cấu trúc này bằng cây cú pháp như trong Hình 1.11(a). Cấu trúc dữ liệu điển hình cho cây được trình bày trong Hình 1.11(b), trong đó một nút nội là một mẩu tin có một trường dành cho toán tử và hai trường chứa các con trỏ chỉ đến các mẩu tin cho các con bên phải và bên trái. Nút lá là một mẩu tin có hai hoặc nhiều trường, một trường để xác định thẻ từ tại nút lá đó, và những trường khác để lưu các thông tin về thẻ từ. Những thông tin bổ sung về các kết cấu ngôn ngữ có thể được lưu lại bằng cách thêm một số trường vào mẩu tin dành cho các nút đó. Chúng ta sẽ thảo luận giai đoạn phân tích cú pháp và ngữ nghĩa trong Chương 4 và Chương 6.



Hình 1.11. Cấu trúc dữ liệu trong (b) để biểu diễn cây trong (a).

Giai đoạn sinh mã trung gian

Sau khi phân tích cú pháp và ngữ nghĩa, một số trình biên dịch sẽ tạo ra một dạng biểu diễn trung gian của chương trình nguồn. Chúng ta có thể xem dạng biểu diễn này

như một chương trình dành cho một máy trừu tượng. Chúng có hai đặc tính quan trọng: dễ tạo và dễ dịch sang chương trình đích.

Dạng biểu diễn trung gian có rất nhiều loại. Trong Chương 8 chúng ta sẽ xem xét một dạng gọi là "*mã ba địa chỉ*" (three-address code). Nó giống như hợp ngữ của một máy, trong đó mỗi vị trí của bộ nhớ có thể đóng vai trò như một *thanh ghi* (register). Mã ba địa chỉ chứa một dãy các *chỉ thị* (instruction), mỗi chỉ thị có tối đa ba đối. Chương trình nguồn trong biểu thức (1.1) có thể xuất hiện ở dạng mã ba địa chỉ như sau:

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2                (1.3)
id1 := temp3
```

Dạng trung gian này có một số tính chất. Thứ nhất, mỗi chỉ thị ba địa chỉ có tối đa một toán tử ngoài toán tử gán ra. Vì thế khi tạo ra những chỉ thị này, trình biên dịch phải quyết định thứ tự các thao tác được thực hiện; phép nhân đi trước phép cộng trong chương trình nguồn của (1.1). Thứ hai, trình biên dịch phải tạo ra một tên tạm để giữ giá trị do chỉ thị tính ra. Thứ ba, một số chỉ thị ba địa chỉ có ít hơn ba toán hạng, ví dụ như các chỉ thị dầu và chỉ thị cuối cùng trong (1.3).

Trong Chương 8 (Tập II), chúng ta sẽ đề cập đến những dạng biểu diễn trung gian chính được sử dụng trong các trình biên dịch. Nói chung, những dạng biểu diễn này phải thực hiện được nhiệu thao tác hơn là chỉ tính các biểu thức; chúng phải xử lý các kết cấu điều khiển và các lời gọi thủ tục. Chương 5 và Chương 8 sẽ trình bày các thuật toán sinh mã trung gian cho một số kết cấu của các ngôn ngữ lập trình điển hình.

Giai đoạn tối ưu mã

Giai đoạn tối ưu mã cố gắng cải thiện mã trung gian để tạo ra được các mã máy chạy nhanh hơn. Một số phương pháp tối ưu hóa hoàn toàn tầm thường. Chẳng hạn một thuật toán tự nhiên là tạo ra mã trung gian (1.3) bằng cách sử dụng một chỉ thị cho mỗi toán tử trong dạng biểu diễn cây sau khi đã phân tích ngữ nghĩa, dù rằng vẫn có những cách tính tốt hơn bằng cách dùng hai chỉ thị

```
temp1 := id3 * 60.0
id1 := id2 + temp1                  (1.4)
```

Không có gì sai trong thuật toán đơn giản trên bởi vì vấn đề này có thể được giải quyết trong giai đoạn tối ưu hóa mã. Nghĩa là trình biên dịch có thể suy ra rằng việc đổi số `60` sang dạng biểu diễn số thực có thể chỉ cần thực hiện một lần vào lúc biên dịch, vì thế có thể lược bỏ phép toán **inttoreal**. Ngoài ra `temp3` chỉ được dùng một lần để truyền giá trị của nó cho `id1`. Do đó sẽ tốt hơn nếu thế `id1` vào chỗ `temp3`, như thế

câu lệnh cuối cùng của (1.3) không còn cần đến nữa và được thay bằng (1.4).

Có một khác biệt rất lớn giữa khối lượng tối ưu hóa mã được các trình biên dịch khác nhau thực hiện. Trong những trình biên dịch được gọi là "trình biên dịch chuyên tối ưu", một phần thời gian đáng kể được dành cho giai đoạn này. Tuy nhiên cũng có những phương pháp tối ưu giúp cải thiện đáng kể thời gian chạy của chương trình nguồn mà không làm chậm đi công việc biên dịch quá nhiều. Nhiều phương pháp như thế sẽ được thảo luận trong Chương 9, còn ở Chương 10 sẽ đề cập đến công nghệ đã được những trình biên dịch chuyên tối ưu mạnh nhất sử dụng.

Giai đoạn sinh mã

Giai đoạn cuối cùng của biên dịch là sinh mã đích, bình thường là mã máy hay mã hợp ngữ. Các vị trí vùng nhớ được chọn lựa cho mỗi biến được chương trình sử dụng. Sau đó các chỉ thị trung gian được dịch lần lượt thành chuỗi các chỉ thị mã máy. Vấn đề quyết định là việc gán các biến cho các thanh ghi.

Chẳng hạn sử dụng các thanh ghi 1 và 2, quá trình dịch mã của (1.4) có thể trở thành:

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1                                    (1.5)
ADDF R2, R1
MOVF R1, id1
```

Toán hạng thứ nhất và thứ hai của mỗi chỉ thị tương ứng mô tả nguồn và đích. Chữ F trong mỗi chỉ thị cho chúng ta biết rằng những chỉ thị đang xử lý các số chấm động. Đoạn mã này di chuyển nội dung ở địa chỉ[1] id3 vào thanh ghi 2, sau đó nhân nó với số thực 60.0. Dấu # để xác định rằng 60.0 được xem như một hằng. Chỉ thị thứ ba di chuyển id2 vào thanh ghi 1 và cộng giá trị đã được tính trước đó trong thanh ghi 2 vào cho nó. Cuối cùng giá trị trong thanh ghi 1 được chuyển vào địa chỉ của id1, vì thế đoạn mã này thực hiện phép gán trong Hình 1.10. Chương 9 (Tập II) sẽ đề cập đến quá trình phát sinh mã.

1.4 ANH EM CỦA TRÌNH BIÊN DỊCH

Như chúng ta đã thấy trong Hình 1.3, nguyên liệu cho trình biên dịch có thể được một hoặc nhiều bộ tiền xử lý tạo ra, và thành phẩm của trình biên dịch có thể cần phải

[1] Chúng ta đã bỏ qua một vấn đề quan trọng về việc cấp phát chỗ cho các định danh của chương trình nguồn. Như chúng ta sẽ thấy trong Chương 7, tổ chức lưu trữ vào lúc chạy phụ thuộc vào ngôn ngữ được biên dịch. Các quyết định cấp phát chỗ được thực hiện trong giai đoạn sinh mã trung gian hoặc giai đoạn sinh mã.

được xử lý tiếp trước khi thu được kết quả ở dạng mã máy. Trong phần này chúng ta sẽ thảo luận về môi trường hoạt tác của một trình biên dịch điển hình.

Bộ tiền xử lý

Bộ tiền xử lý (preprocessor) tạo ra nguyên liệu cho các trình biên dịch. Chúng có thể thực hiện các chức năng sau:

1. Xử lý macro. Bộ tiền xử lý có thể cho phép người dùng định nghĩa các macro, là dạng tắt của các kết cấu dài.
2. Gộp thêm tập tin. Trình biên dịch có thể gộp các *tập tin tiêu đề* (header file) vào trong đoạn chương trình. Thí dụ như trình tiền biên dịch C đưa nội dung của tập tin `<global.h>` vào vị trí của câu lệnh `#include <global.h>` khi nó xử lý tập tin có chứa câu lệnh này.
3. Bộ tiền xử lý "biết suy nghĩ". Những bộ tiền xử lý này tăng cường cho các ngôn ngữ xưa cũ bằng các tiện ích nhằm tạo ra các cấu trúc dữ liệu và kết cấu điều khiển hiện đại hơn. Thí dụ như trình biên dịch có thể cung cấp cho người dùng các macro cài sẵn cho các kết cấu như câu lệnh `while` hoặc `if` khi chúng không có trong ngôn ngữ lập trình.
4. Các mở rộng ngôn ngữ. Các bộ tiền xử lý này cố gắng tăng thêm sức mạnh cho ngôn ngữ qua các macro cài sẵn. Chẳng hạn ngôn ngữ Equel (Stonebraker et al., [1976]) là một ngôn ngữ vấn tin được gắn vào trong C. Các câu lệnh bắt đầu bằng `##` được bộ tiền xử lý thao tác là những câu lệnh truy xuất CSDL, không liên quan gì đến C, và được dịch thành các lời gọi thực hiện các truy xuất CSDL.

Các bộ xử lý macro lo giải quyết hai loại câu lệnh: định nghĩa macro và sử dụng macro. Các định nghĩa macro thông thường được chỉ ra qua một ký tự nào đó hay một từ khóa như `define` hoặc `macro`. Chúng gồm có một *tên* (name) cho macro đang được định nghĩa và *phần thân* (body) tạo ra định nghĩa của nó. Thông thường các bộ xử lý macro cho phép dùng các *tham số hình thức* (formal parameter) trong định nghĩa, nghĩa là các ký hiệu sẽ được thay bằng các giá trị (một "giá trị" là một chuỗi ký tự trong ngữ cảnh này). Việc sử dụng một macro bao gồm việc đặt tên macro và cung cấp các *tham số thực sự* (actual parameter), nghĩa là giá trị cho các tham số hình thức. Bộ xử lý macro sẽ thay tham số thực vào tham số hình thức trong phần thân của macro; sau đó phần thân này được thay vào chỗ có sử dụng macro.

Thí dụ 1.2. Hệ thống T_EX được nói đến trong Phần 1.2 chứa một tiện ích macro tổng quát. Định nghĩa macro có dạng

```
\define <tên macro> <khuôn mẫu> {<thân>}
```

Tên macro là một chuỗi chữ cái có một dấu gạch ngược (\) đặt trước. Khuôn mẫu là

một chuỗi ký tự bất kỳ có dạng `#1, #2, . . . , #9` được xem như các tham số hình thức. Những ký hiệu này cũng có thể xuất hiện trong phần thân nhiều lần. Thí dụ macro sau đây định nghĩa một đoạn trích dẫn tạp chí *Journal of the ACM*.

```
\define\JACM #1;#2;#3.
   {{\sl J. ACM} {\bf #1}:#2, pp. #3.}
```

Tên macro là `\JACM` và khuôn mẫu là "`#1;#2;#3.`"; các dấu chấm phẩy ngăn cách các tham số, và sau tham số cuối cùng là một dấu chấm. Khi sử dụng macro chúng ta viết đúng khuôn mẫu, còn các tham số hình thức có thể được thay bằng các chuỗi tùy ý.[2] Vì thế chúng ta có thể viết

```
\JACM 17;4;715-728.
```

Và hy vọng sẽ in ra được

J. ACM **17**:4, pp. 715–728.

Phần của thân `{\sl J. ACM}` yêu cầu phải in nghiêng (do từ slanted) chuỗi "*J. ACM*". Biểu thức `{\bf #1}` cho biết rằng tham số thực đầu tiên phải **in đậm** (boldface); tham số này là số volume.[3]

$\TeX$ cho phép dùng dấu ngắt câu hoặc một chuỗi văn bản để ngăn cách giữa volume, tạp chí và số trang trong định nghĩa của macro `\JACM`. Chúng ta cũng có thể không sử dụng dấu ngắt câu, trong trường hợp đó $\TeX$ sẽ lấy mỗi tham số thực là một ký tự duy nhất hoặc là một chuỗi được bao quanh bởi dấu `{ }`. □

Trình dịch hợp ngữ

Một số trình biên dịch tạo ra mã hợp ngữ, giống như trong (1.5), và được chuyển cho *trình dịch hợp ngữ* (assembler) để xử lý tiếp. Một số trình biên dịch khác thực hiện luôn công việc của trình dịch hợp ngữ, tạo ra *mã máy khả tái định vị* (relocatable machine code) mà chúng có thể được chuyển trực tiếp đến *trình tải* (loader). Chúng tôi giả thiết rằng độc giả đã từng thấy hoạt động của một trình dịch hợp ngữ; ở đây chúng ta chỉ xem lại mối liên hệ giữa mã hợp ngữ và mã máy.

Mã hợp ngữ (assembly code) là một dạng mã máy dễ nhớ, trong đó chúng ta sử dụng tên thay cho các mã nhị phân của các phép toán,[4] và tên cũng được dùng cho địa

---

[2] Hầu như là mọi chuỗi, bởi vì khi quét từ trái sang phải qua macro, và ngay khi thấy một ký hiệu khớp với các chữ đi sau ký hiệu `#i` trong khuôn mẫu, chuỗi đi trước được xem là đã khớp được với `#i`. Vì thế nếu thay `ab;cd` cho `#1`, chúng ta nhận thấy rằng chỉ có `ab` được khớp với `#1` và `cd` được khớp với `#2`.

[3] Các tạp chí máy tính thường được xuất bản hàng tháng hay hai tháng. Cụ thể, tạp chí Journal of ACM ra mỗi tháng một số, và một năm 12 số (12 kỳ) được gọi là một volume. Volume được đánh số từ 1 trở đi, tính từ năm xuất bản đầu tiên. Thí dụ ở đoạn trích dẫn trên, bài viết đang trích dẫn được đăng trên tạp chí JACM, Volume 17, kỳ 4, từ trang 715 đến trang 728. Xin xem các mẫu trích dẫn trong Danh mục các tài liệu tham khảo ở cuối sách. (ND)

chỉ bộ nhớ. Một chuỗi *chỉ thị hợp ngữ* (assembly instruction) điển hình có thể là

```
MOV a, R1
ADD #2, R1                (1.6)
MOV R1, b
```

Đoạn mã này chuyển nội dung ở địa chỉ **a** vào thanh ghi 1 (register), cộng 2 vào, xử lý nội dung của thanh ghi 1 như số chấm cố định, và cuối cùng lưu kết quả vào vị trí được đặt tên là `b`. Vì thế nó tính `b := a + 2`.

Thông thường các hợp ngữ cũng có các macro, tương tự như các tiện ích macro trong bộ tiền xử lý macro đã được thảo luận.

Hợp ngữ hai lượt

Dạng đơn giản nhất của một trình dịch hợp ngữ duyệt hai lượt trên *nguyên liệu* (input), mỗi *lượt* (pass) sẽ đọc tập tin nguyên liệu một lần. Trong lượt đầu, tất cả các định danh biểu thị cho các vị trí lưu trữ được xác định và được lưu trong một bảng ký hiệu (tách biệt với bảng ký hiệu của trình biên dịch). Các định danh được gán cho các vị trí nhớ khi chúng được gặp lần đầu tiên, vì thế sau khi đọc (1.6), bảng ký hiệu có thể chứa các mục ghi như được trình bày trong Hình 1.12. Trong hình đó, chúng ta đã giả thiết rằng một *từ nhớ* (word) chứa bốn byte dành cho mỗi định danh, và các địa chỉ bắt đầu từ byte 0.

ĐỊNH DANH	ĐỊA CHỈ
a	0
b	4

Hình 1.12. Một bảng ký hiệu của trình dịch hợp ngữ chứa các định danh của (1.6).

Trong lượt thứ hai, trình dịch hợp ngữ quét lại nguyên liệu một lần nữa. Lần này, nó dịch mỗi mã của phép toán thành chuỗi bit biểu thị cho phép toán đó bằng ngôn ngữ máy, và dịch mỗi định danh biểu thị vị trí thành địa chỉ tương ứng với định danh trong bảng ký hiệu.

Thành phẩm (output) của lượt thứ hai thường là một mã máy khả tái định vị; bởi vì nó có thể được tải vào bộ nhớ bắt đầu từ một vị trí L nào đó; nghĩa là nếu cộng L vào tất cả các địa chỉ trong chương trình, thì mọi tham chiếu đều đúng. Vì thế thành phẩm của trình dịch hợp ngữ phải phân biệt những chỉ thị có tham chiếu đến những

---

[1] Ở mức mã hợp ngữ và mã máy, mỗi phép toán đơn giản như cộng, trừ, lưu trữ, vân vân, được gọi là một *chỉ thị* (instruction). Mỗi câu lệnh trong ngôn ngữ cấp cao thường được dịch thành nhiều chỉ thị của mã máy. (ND)

địa chỉ khả tái định vị.

Thí dụ 1.3. Dưới đây là một đoạn mã máy giả định được dịch từ các chỉ thị hợp ngữ (1.6).

```
0001 01 00 00000000 *
0011 01 10 00000010                    (1.7)
0010 01 00 00000100 *
```

Chúng ta tưởng tượng có một *từ nhỏ* (small word) dành cho chỉ thị, trong đó bốn bit đầu tiên là mã chỉ thị với giá trị 0001, 0010 và 0011 lần lượt biểu thị cho *chỉ thị tải* (load), *lưu vào* (store) và *cộng*. Tải và lưu ở đây có nghĩa là di chuyển từ bộ nhớ vào thanh ghi và ngược lại. Hai bit tiếp theo biểu thị thanh ghi, trong đó 01 muốn nói đến thanh ghi 1 trong cả ba chỉ thị trên. Hai bit sau đó biểu thị một "*dấu chỉ*" (tag) với 00 có nghĩa là chế độ địa chỉ thông thường, ở đó tám bit cuối cùng tham chiếu đến một địa chỉ bộ nhớ. Dấu chỉ 10 có nghĩa là chế độ trực tiếp, trong đó tám bit cuối cùng được dùng trực tiếp làm toán hạng. Chế độ này xuất hiện trong chỉ thị thứ hai của (1.7).

Chúng ta cũng thấy trong (1.7) một dấu sao * ở chỉ thị thứ nhất và thứ ba. Dấu này biểu thị *bit tái định vị* (relocation bit) đi kèm với mỗi toán hạng trong mã máy khả tái định vị. Giả sử rằng không gian địa chỉ chứa dữ liệu được tải vào vùng bộ nhớ bắt đầu từ vị trí L. Sự hiện diện của dấu * có nghĩa là phải cộng thêm L vào địa chỉ của chỉ thị. Vì thế nếu L = 00001111, nghĩa là 15 (hệ thập phân) thì a và b phải nằm ở vị trí tương ứng là 15 và 19, và các chỉ thị của (1.7) sẽ xuất hiện dưới dạng mã máy tuyệt đối hay không tái định vị được là

```
0001 01 00 00001111
0011 01 10 00000010                    (1.8)
0010 01 00 00010011
```

chú ý rằng không có dấu * đi kèm với chỉ thị thứ hai trong (1.7), vì thế không cộng thêm L vào cho địa chỉ của nó trong (1.8). Điều này là hợp lý vì các bit biểu thị hằng 2, không phải vị trí 2. □

Trình tải và trình hiệu chỉnh-liên kết

Thông thường một chương trình được gọi là *trình tải* (loader) thực hiện cả hai chức năng tải và hiệu chỉnh-liên kết. Quá trình tải bao gồm lấy mã máy khả tái định vị, thay đổi các địa chỉ như đã thảo luận trong Thí dụ 1.3 rồi đặt các chỉ thị đã thay đổi và dữ liệu vào bộ nhớ tại các địa chỉ thích hợp.

Trình hiệu chỉnh-liên kết (link-editor) cho phép chúng ta tạo ra một chương trình duy nhất từ nhiều tập tin chứa các mã máy khả tái định vị. Những tập tin này có thể là kết quả nhiều lần biên dịch khác nhau, và một hoặc nhiều tập tin thư viện gồm các thủ tục được hệ thống cung cấp và có sẵn cho mọi chương trình cần dùng đến chúng.

Nếu các tập tin được dùng chung với nhau một cách hiệu quả thì có thể phải dùng đến một số *tham chiếu ngoài* (external reference), trong đó mã chương trình của một tập tin có tham chiếu đến một vị trí trong một tập tin khác. Tham chiếu này có thể chỉ đến một vị trí được định nghĩa trong một tập tin nhưng được dùng trong một tập tin khác, hoặc có thể chỉ đến một điểm vào của một thủ tục xuất hiện trong một tập tin và được gọi từ một tập tin khác. Tập tin mã máy khả tái định vị phải giữ thông tin này trong bảng ký hiệu cho mỗi vị trí dữ liệu hoặc nhãn chỉ thị được tham chiếu ngoài. Nếu không biết trước chỗ nào cần phải tham chiếu, chúng ta phải gộp toàn bộ bảng ký hiệu hợp ngữ làm thành phần của mã máy khả tái định vị.

Thí dụ, đoạn mã của (1.7) có thể được đặt trước bởi

```
a    0
b    4
```

Nếu một tập tin được tải với (1.7) có tham chiếu đến b, thế thì tham chiếu đó sẽ được thay bằng 4 cộng với offset mà các vị trí dữ liệu trong tập tin (1.7) đã được tái định vị.

1.5 NHÓM CÁC GIAI ĐOẠN

Khi thảo luận về các giai đoạn ở Phần 1.3, chúng ta đã xem xét về tổ chức logic của một trình biên dịch. Khi cài đặt, hoạt động của nhiều giai đoạn thường được nhóm lại với nhau.

Kỳ đầu và kỳ sau

Thường thì các giai đoạn được gom lại thành *kỳ đầu* (front end) và *kỳ sau* (back end). Kỳ đầu gồm có các giai đoạn hoặc các phần giai đoạn phụ thuộc nhiều vào *ngôn ngữ nguồn* (source language) và hầu như độc lập với máy đích. Thông thường nó bao gồm giai đoạn *phân tích từ vựng* (lexical analysis) và *phân tích cú pháp* (syntactic analysis), quá trình tạo *bảng ký hiệu* (symbol table), *phân tích ngữ nghĩa* (semantic analysis) và *sinh mã trung gian* (generation of intermediate code). Một phần công việc tối ưu hóa mã cũng có thể được thực hiện ở kỳ đầu. Kỳ đầu cũng phải thực hiện xử lý lỗi xuất hiện ở từng giai đoạn vừa nêu.

Kỳ sau bao gồm các phần phụ thuộc vào máy đích và nói chung chúng không phụ thuộc vào ngôn ngữ nguồn mà là ngôn ngữ trung gian. Trong kỳ sau chúng ta gặp một số vấn đề tối ưu hóa mã, phát sinh mã cùng với việc xử lý lỗi và các thao tác trên bảng ký hiệu.

Chúng ta có thể lấy kỳ đầu của một trình biên dịch rồi thực hiện lại kỳ sau, tạo ra một trình biên dịch cho chính ngôn ngữ nguồn đó trên một mảy khác. Nếu kỳ sau được thiết kế cẩn thận thì có thể không phải thiết kế lại quá nhiều; điều này sẽ được thảo luận trong Chương 9. Người ta cũng muốn biên dịch nhiều ngôn ngữ khác nhau thành

một ngôn ngữ trung gian và dùng một kỳ sau chung cho nhiều kỳ đầu khác nhau, vì thế thu được nhiều trình biên dịch cho một máy. Tuy nhiên vì những khác biệt tinh tế về quan điểm của các ngôn ngữ nên mới chỉ có ít thành công theo hướng này.

Các lượt

Một số giai đoạn biên dịch thường được cài đặt bằng một *lượt* (pass) duy nhất, bao gồm việc đọc tập tin nguyên liệu và ghi ra tập tin thành phẩm. Trong thực hành có khác biệt rất lớn trong phương thức nhóm các giai đoạn của trình biên dịch thành các lượt, vì thế chúng tôi sẽ thảo luận về trình biên dịch dựa theo các giai đoạn chứ không phải theo lượt. Chương 12 sẽ thảo luận về một số trình biên dịch tiêu biểu và bàn thêm về cách thức cấu trúc các giai đoạn vào các lượt.

Như chúng tôi đã đề cập, người ta hay nhóm nhiều giai đoạn vào một lượt, và hoạt động của các giai đoạn này được thực hiện đan xen lẫn nhau. Thí dụ các giai đoạn phân tích từ vựng, phân tích cú pháp, phân tích ngữ nghĩa và phát sinh mã trung gian có thể được nhóm lại thành một lượt. Nếu như thế thì dòng thẻ từ sau giai đoạn phân tích có thể được dịch trực tiếp thành mã trung gian. Chi tiết hơn, chúng ta xem *thể phân cú pháp* (parser) là "nhân vật điều phối". Nó cố gắng khám phá cấu trúc ngữ pháp trên những thẻ từ nó gặp; nó thu nhận các thẻ từ này khi cần đến chúng bằng cách yêu cầu *thể phân từ vựng* (lexical analyzer) tìm thẻ từ tiếp theo. Khi cấu trúc ngữ pháp được khám phá ra, thể phân cú pháp sẽ gọi *thể sinh mã trung gian* (intermediate code generator) để thực hiện việc phân tích ngữ nghĩa và sinh ra một phần mã. Một trình biên dịch được tổ chức theo cách này sẽ được trình bày trong Chương 2.

Thu gọn số lượt

Người ta chỉ muốn có một số ít lượt thôi bởi vì mỗi lượt đều mất thời gian đọc và ghi ra tập tin trung gian. Ngược lại nếu chúng ta gom quá nhiều giai đoạn vào trong một lượt thì có thể phải duy trì toàn bộ chương trình trong bộ nhớ, bởi vì một giai đoạn có thể cần thông tin theo một thứ tự khác với thứ tự nó đã được tạo ra. Dạng biểu diễn trung gian của chương trình có thể lớn hơn nhiều so với chương trình nguồn hoặc chương trình đích, vì thế không gian cần dùng có thể không phải là vấn đề có thể bỏ qua được.

Đối với một số giai đoạn, nhóm chúng vào một lượt làm nảy sinh một số vấn đề. Chẳng hạn, như chúng ta đã nói ở trên, giao tiếp giữa thể phân từ vựng và thể phân cú pháp thường bị hạn chế qua một thẻ từ duy nhất. Mặt khác sẽ rất khó thực hiện việc tạo mã trước khi tạo xong mã trung gian. Thí dụ các ngôn ngữ như PL/I và Algol 68 cho phép các biến được dùng trước khi khai báo. Chúng ta không thể tạo ra mã đích cho một kết cấu nếu không biết được kiểu của các biến có mặt trong kết cấu đó. Tương tự phần lớn các ngôn ngữ đều cho phép dùng lệnh `goto` để nhảy tới trước trong chương trình. Chúng ta không thể xác định được địa chỉ đích của một lệnh nhảy như

thế cho đến khi chúng ta thấy được mã nguồn ở trong đoạn đó và mã đích được sinh ra cho nó.

Trong một số trường hợp có thể để lại một khoảng trống dành cho các thông tin hiện đang thiếu, và điền vào khoảng này khi có được thông tin đó. Đặc biệt là việc sinh mã trung gian và mã đích thường có thể được trộn vào một lượt bằng cách sử dụng một kỹ thuật có tên là "*điền vào sau*" (backpatching). Mặc dù không thể giải thích được mọi chi tiết khi chưa xem xét quá trình sinh mã trung gian trong Chương 8, chúng ta có thể minh họa kỹ thuật điền vào sau qua một trình dịch hợp ngữ. Cần nhớ rằng trong phần trước chúng ta đã thảo luận một trình dịch hợp ngữ hai lượt, trong đó lượt thứ nhất khám phá tất cả mọi định danh biểu thị vị trí bộ nhớ và suy diễn địa chỉ khi chúng được khám phá. Sau đó lượt thứ hai sẽ thay địa chỉ cụ thể cho các định danh này.

Chúng ta có thể tổ hợp hành động của các lượt như sau. Khi gặp một câu lệnh hợp ngữ tham chiếu tới trước, chẳng hạn

```
GOTO target
```

chúng ta tạo ra một chỉ thị khung với mã máy cho `GOTO` và các khoảng trống dành cho địa chỉ. Tất cả mọi chỉ thị có khoảng trống dành cho địa chỉ của `target` được lưu trong một danh sách kèm với mục ghi cho `target` trong bảng ký hiệu. Các khoảng trống sẽ được điền vào khi chúng ta gặp một chỉ thị như

```
target: MOV foobar, R1
```

và xác định được giá trị của `target`; nó là địa chỉ của chỉ thị ở trên. Sau đó chúng ta "điền vào sau" bằng cách đi dọc xuống danh sách của `target` chứa tất cả mọi chỉ thị có dùng đến địa chỉ, thay nó vào các khoảng trống trong trường địa chỉ của những chỉ thị đó. Cách tiếp cận này rất dễ cài đặt nếu các chỉ thị được giữ trong bộ nhớ cho đến khi xác định được tất cả mọi địa chỉ.

Đây là một cách tiếp cận hợp lý cho một trình dịch hợp ngữ với tất cả thành phẩm của nó đều được lưu trong bộ nhớ. Vì các dạng biểu diễn trung gian và dạng cuối cùng cho một trình dịch hợp ngữ nói một cách đơn giản là giống nhau, và chắc chắn rằng có chiều dài gần bằng nhau, kỹ thuật điền vào sau trên toàn bộ chiều dài của chương trình dịch hợp ngữ không phải là không khả thi. Tuy nhiên trong một trình biên dịch, với mã trung gian cần dùng nhiều không gian, chúng ta cần phải cẩn thận về khoảng cách xảy ra tình trạng điền vào lại.

1.6 CÔNG CỤ XÂY DỰNG TRÌNH BIÊN DỊCH

Các nhà viết trình biên dịch, cũng giống như mọi lập trình viên khác, có thể cần dùng đến các công cụ phần mềm như các *bộ gỡ rối* (debugger), các chương trình quản lý

phiên bản, vân vân. Trong Chương 11 (Tập II) chúng ta sẽ xem xét một số công cụ cài đặt trình biên dịch. Ngoài những công cụ phát triển phần mềm này, một số công cụ chuyên dụng hơn cũng đã được phát triển giúp cài đặt nhiều giai đoạn khác nhau của trình biên dịch. Trong phần này chúng ta sẽ nói sơ lược về chúng và trong từng chương sẽ phân tích chi tiết hơn.

Không bao lâu sau khi các trình biên dịch đầu tiên vừa được viết xong, các hệ thống hỗ trợ cho công việc viết trình biên dịch đã xuất hiện. Những hệ thống này thường được gọi là *trình biên dịch-trình biên dịch* (compiler-compiler), *bộ sinh trình biên dịch* (compiler generator) hoặc *hệ thống viết chương trình dịch*. Nhìn chung, chúng chỉ tập trung vào một mô hình cụ thể của ngôn ngữ, và chúng rất thích hợp cho việc tạo ra các trình biên dịch của các ngôn ngữ tương tự như mô hình đó.

Thí dụ như người ta giả thiết rằng các thể phân từ vựng cho mọi ngôn ngữ đều như nhau, ngoại trừ các từ khóa cụ thể và các dấu hiệu cần được nhận diện. Nhiều loại trình biên dịch-trình biên dịch thực sự sinh ra các thủ tục phân tích từ vựng cố định để dùng trong trình biên dịch được tạo ra. Những thủ tục này chỉ khác nhau ở danh sách từ khóa cần phải nhận ra, và danh sách này là tất cả những gì cần thiết phải được cung cấp từ người sử dụng. Cách tiếp cận này rất có giá trị, nhưng có thể không hoạt động được nếu nó phải nhận diện các thẻ từ không tiêu chuẩn, chẳng hạn như các định danh có chứa một số ký tự khác ngoài chữ cái và ký số.

Một số công cụ tổng quát khác đã được tạo ra để thiết kế tự động các thành phần của trình biên dịch cụ thể. Những công cụ này sử dụng các ngôn ngữ chuyên dụng để đặc tả và cài đặt các thành phần, và có nhiều công cụ sử dụng các thuật toán hết sức phức tạp. Những công cụ thành công nhất là những công cụ che dấu được các chi tiết thuật toán phát sinh, tạo ra các thành phần có thể dễ dàng được tích hợp với phần còn lại của một trình biên dịch. Dưới đây là danh sách một số công cụ xây dựng trình biên dịch đáng chú ý:

1. *Bộ sinh thể phân cú pháp* (parser generator). Chúng tạo ra *thể phân cú pháp* (parser), thường là dựa trên một *văn phạm phi ngữ cảnh* (context-free grammar). Trong những trình biên dịch đầu tiên, phân tích cú pháp tiêu tốn không những phần lớn thời gian chạy của trình biên dịch mà còn tốn rất nhiều công sức viết trình biên dịch. Giai đoạn này đến nay được xem như một trong những giai đoạn dễ cài đặt nhất. Nhiều ngôn ngữ "nhỏ bé" được dùng để chế bản cho cuốn sách này, chẳng hạn như PIC (Kernighan [1982]) và EQN, đã được cài đặt trong vài ngày bằng cách dùng bộ sinh thể phân cú pháp được mô tả trong Phần 4.7. Nhiều bộ sinh thể phân cú pháp sử dụng các thuật toán phân tích cú pháp rất mạnh và quá phức tạp nên không thể thực hiện được bằng thủ công.

2. *Bộ sinh thể nhận nguyên liệu* (scanner generator). Chúng tạo ra các thể phân từ vựng một cách tự động, thường qua một đặc tả dựa trên các biểu thức chính qui được thảo luận trong Chương 3. Tổ chức cơ bản của thể phân từ vựng thực chất là

một automat hữu hạn. Một bộ sinh thể nhận nguyên liệu và việc cài đặt nó được thảo luận trong Phần 3.5 và 3.8.

3. *Động cơ dịch dựa cú pháp* (syntax-directed translation engine). Chúng tạo ra các thủ tục duyệt cây phân tích cú pháp, như ở Hình 1.4, và sinh mã trung gian. Ý tưởng cơ bản là một hoặc nhiều "bản dịch" được liên kết với mỗi nút của cây phân tích cú pháp, và mỗi bản dịch được định nghĩa theo các bản dịch tại các nút lân cận của nó trong cây. Những động cơ như thế được thảo luận trong Chương 5.

4. *Bộ tự động sinh mã* (automatic code generator). Một công cụ như thế nhận một tập qui tắc định nghĩa phương pháp dịch mỗi thao tác trong ngôn ngữ trung gian thành ngôn ngữ máy cho máy đích. Những qui tắc này phải chứa đủ chi tiết để chúng ta có thể xử lý được các phương pháp truy xuất dữ liệu khác nhau; thí dụ các biến có thể nằm trong các *thanh ghi* (register), ở một vị trí cố định (tĩnh) trong bộ nhớ, hoặc có thể được cấp phát một vị trí trên *chồng xếp* (stack). Kỹ thuật cơ bản là "đối sánh mẫu". Các lệnh của mã trung gian được thay bằng các mẫu biểu diễn các chuỗi chỉ thị máy sao cho các giả thiết về không gian lưu trữ các biến tương hợp giữa mẫu này với mẫu khác. Vì thường có nhiều chọn lựa liên quan đến vị trí đặt các biến (thí dụ ở một trong số thanh ghi hoặc trong bộ nhớ), có nhiều cách có thể bố trí mã trung gian với một tập khuôn mẫu đã cho, và chúng ta cần chọn ra được một cách bố trí thích hợp mà không bị bùng nổ tổ hợp về thời gian chạy của trình biên dịch. Các công cụ thuộc loại này được đề cập đến trong Chương 9 (Tập II).

5. *Động cơ phân tích dòng dữ liệu* (data-flow engine). Nhiều thông tin cần cho việc tối ưu hóa mã đều cần phải phân tích dòng dữ liệu, là việc thu thập thông tin về cách thức truyền giá trị từ phần này của chương trình đến mỗi phần khác. Các tác vụ thuộc loại này có thể được thực hiện chủ yếu bằng một thủ tục giống nhau, trong đó người sử dụng sẽ cung cấp các chi tiết về mối liên hệ giữa các câu lệnh mã trung gian và thông tin cần thu thập. Một công cụ thuộc loại này được mô tả trong Phần 10.11.

GHI CHÚ VỀ TÀI LIỆU THAM KHẢO

Viết về lịch sử xây dựng trình biên dịch vào năm 1962, Knuth [1962] đã nhận xét rằng, "Trong lãnh vực này đã có một số lượng khám phá khác thường và hoàn toàn độc lập nhau về cùng một kỹ thuật bởi nhiều nhà nghiên cứu." Ông tiếp tục nhận xét rằng thực sự rất nhiều cá nhân đã phát hiện ra "nhiều bình diện khác nhau của một kỹ thuật, và nó đã được "trau chuốt" qua nhiều năm để trở thành một thuật toán hết sức tuyệt vời mà không có người nào trong số những người sản sinh ra nó biết được hết." Phần ghi chú về tài liệu tham khảo trong chương này chỉ nhằm giúp độc giả nghiên cứu thêm ở các tài liệu chuyên ngành.

Các ghi chép về lịch sử phát triển các ngôn ngữ lập trình và trình biên dịch cho đến khi Fortran ra đời có thể tìm đọc trong Knuth and Trabb Pardo [1977]. Wexelblat [1981] chứa các thu thập về nhiều ngôn ngữ lập trình từ những người đã tham gia phát triển chúng.

Một số bài báo ban đầu về biên dịch đã được tập hợp trong Rosen [1967] và Pollack [1972]. Số báo tháng 1 năm 1961 của *Communication of the ACM* cung cấp một cái nhìn tổng quan về tình hình phát triển trình biên dịch vào thời điểm đó. Một phân tích chi tiết về trình biên dịch Algol 60 có trong Randel and Russell [1964].

Khởi điểm từ những năm đầu thập niên 60 với việc nghiên cứu cú pháp, các nghiên cứu lý thuyết đã có ảnh hưởng sâu sắc đến sự phát triển của công nghệ trình biên dịch, ít nhất cũng có ảnh hưởng tương tự như đối với các lãnh vực khác của khoa học máy tính. Sự hấp dẫn của cú pháp kể từ đó đã giảm dần, nhưng biên dịch về tổng thể vẫn tiếp tục là một đề tài nghiên cứu đầy hấp dẫn. Thành quả của nó sẽ trở nên rõ ràng qua những thảo luận chi tiết hơn về vấn đề biên dịch trong những chương tiếp theo.

CHƯƠNG 2

Một Trình Biên Dịch Một Lượt Đơn Giản

Chương này sẽ giới thiệu về các vấn đề sẽ được thảo luận qua các chương từ Chương 3 đến Chương 8 của quyển sách này. Chúng tôi sẽ trình bày một số kỹ thuật biên dịch cơ bản, được minh họa bằng cách phát triển một chương trình C dịch một *biểu thức trung vị* (infix expression) thành một *biểu thức hậu vị* (postfix expression). Ở đây, chúng ta tập trung vào *kỳ đầu* (front end) của trình biên dịch, nghĩa là các giai đoạn phân tích từ vựng, phân tích cú pháp, và sinh mã trung gian. Chương 9 và 10 sẽ bàn đến vấn đề phát sinh và tối ưu mã.

2.1 TỔNG QUAN

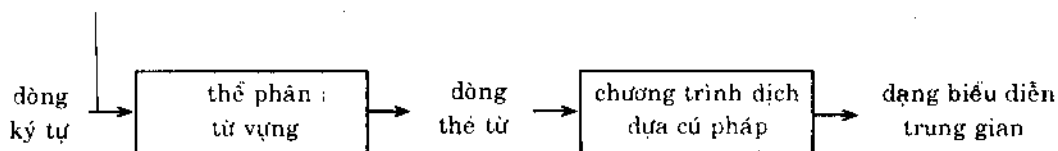
Một ngôn ngữ lập trình có thể được định nghĩa bằng cách mô tả xem một chương trình của nó trông như thế nào (*cú pháp* của ngôn ngữ), và các chương trình muốn nói gì (*ngữ nghĩa* của ngôn ngữ). Để đặc tả cú pháp của một ngôn ngữ, chúng ta trình bày một ký pháp đã được sử dụng rộng rãi, đó là *văn phạm phi ngữ cảnh* (context-free grammar) hay ký pháp BNF (Backus-Naur Form). Với các ký pháp hiện có, ngữ nghĩa của ngôn ngữ khó mô tả hơn nhiều so với cú pháp. Do vậy để đặc tả ngữ nghĩa của một ngôn ngữ, chúng ta sẽ sử dụng các mô tả không hình thức và đưa ra các thí dụ gợi ý.

Bên cạnh việc đặc tả cú pháp của một ngôn ngữ, văn phạm phi ngữ cảnh còn có thể hướng dẫn quá trình dịch các chương trình. Một kỹ thuật biên dịch hướng văn phạm có tên là *phiên dịch dựa cú pháp* (syntax-directed translation), rất có ích cho việc tổ chức kỳ đầu của trình biên dịch và sẽ được dùng rộng rãi trong chương này.

Khi thảo luận về kỹ thuật phiên dịch dựa cú pháp, chúng ta sẽ xây dựng một trình biên dịch dùng để chuyển một biểu thức dạng trung vị sang dạng hậu vị, là ký pháp với các toán tử nằm sau các toán hạng của chúng. Chẳng hạn, dạng hậu vị của biểu thức `9-5+2` là `95-2+`. Ký pháp hậu vị có thể được chuyển trực tiếp thành chương trình máy tính, thực hiện tất cả các tính toán bằng cách sử dụng *chồng xếp* (stack). Chúng ta sẽ bắt đầu qua việc xây dựng một chương trình đơn giản có thể dịch các biểu thức

chứa các *ký số* (digit) được phân cách bởi các dấu cộng và trừ sang dạng hậu vị. Khi những ý tưởng cơ bản đã trở nên rõ ràng hơn, chúng ta sẽ mở rộng chương trình, cho phép xử lý nhiều *kết cấu* (construct) tổng quát hơn của các ngôn ngữ lập trình. Mỗi *chương trình dịch* (translator) của chúng ta được tạo ra bằng cách mở rộng có hệ thống chương trình dịch trước đó.

Trong trình biên dịch của chúng ta, *thể phân tích từ vựng* (lexical analyzer) hay nói gọn là *thể phân từ vựng* sẽ biến đổi dòng ký tự nhập (nguyên liệu) thành dòng các *thẻ từ* (token), là *nguyên liệu* (input) cho giai đoạn kế tiếp như được trình bày trong Hình 2.1. *Chương trình dịch dựa cú pháp* (syntax-directed translator) trong hình này là tổ hợp của *thể phân cú pháp* (syntax analyzer) và *thể sinh mã trung gian* (intermediate-code generator). Một lý do để khởi sự với các biểu thức chứa ký số và các toán tử đó là công việc phân tích từ vựng rất dễ; mỗi ký tự nguyên liệu tạo ra một thẻ từ duy nhất. Về sau chúng ta sẽ mở rộng ngôn ngữ, đưa cả các kết cấu từ vựng vào, chẳng hạn như các số, *định danh* (identifier) và *từ khóa* (keyword). Đối với ngôn ngữ mở rộng này chúng ta sẽ xây dựng một thể phân từ vựng lo tập hợp các ký tự nguyên liệu kế cận nhau ứng với các thẻ từ thích hợp. Việc xây dựng thể phân từ vựng sẽ được thảo luận chi tiết trong Chương 3.



Hình 2.1. Cấu trúc kỳ đầu của trình biên dịch.

2.2 ĐỊNH NGHĨA CÚ PHÁP

Trong phần này chúng ta giới thiệu một hệ ký pháp có tên gọi là *văn phạm phi ngữ cảnh* (context-free grammar) hoặc nói gọn là *văn phạm* (grammar), được dùng để xác định cú pháp của một ngôn ngữ. Văn phạm này sẽ được sử dụng trong suốt cuốn sách để đặc tả kỳ đầu của một trình biên dịch.

Một văn phạm thường mô tả cấu trúc phân cấp của nhiều *kết cấu* (construct) của các ngôn ngữ lập trình. Chẳng hạn câu lệnh **if-else** trong C có dạng

if (expression) statement **else** statement

Có nghĩa nó là một ghép nối của từ khóa **if**, một dấu ngoặc mở, một *biểu thức* (expres-

sion), một dấu ngoặc đóng, một *câu lệnh* (statement), từ khóa **else** và một câu lệnh khác. (Trong C không có từ khóa **then**). Nếu sử dụng biến *expr* để biểu thị cho một biểu thức và biến *stmt* để biểu thị cho một câu lệnh, qui tắc này có thể được diễn tả như sau

$$stmt \rightarrow \textbf{if} \ (\ expr \ ) \ stmt \ \textbf{else} \ stmt \quad (2.1)$$

trong đó mũi tên có thể được đọc là "có thể có dạng." Một qui tắc như thế được gọi là một *luật sinh* (production). Trong một luật sinh, một thành phần từ vựng như từ khóa **if** và các dấu ngoặc được gọi là các *thẻ từ* (token). Các biến như *expr* và *stmt* biểu thị chuỗi các thẻ từ và được gọi là các *ký hiệu chưa tận* hoặc nói gọn là *chưa tận* (nonterminal).

Một *văn phạm phi ngữ cảnh* có bốn thành phần:

1. Một tập các thẻ từ, được xem là các *ký hiệu tận* hay nói gọn là *tận* (terminal).
2. Một tập các *chưa tận* (nonterminal).
3. Một tập luật sinh, trong đó mỗi luật sinh bao gồm một chưa tận, được gọi là *vế trái* của luật sinh, một mũi tên, và một chuỗi các thẻ từ và/hoặc các chưa tận, là *vế phải* của luật sinh.
4. Một ký hiệu khởi đầu, đó là một chưa tận.

Chúng ta tuân theo qui ước đặc tả văn phạm bằng cách liệt kê các luật sinh, trước tiên là các luật sinh cho ký hiệu khởi đầu. Chúng ta giả sử rằng các *ký số* (digit), các *dấu* (sign) như <=, và chuỗi in đậm như **while** là các tận. Một tên in nghiêng là chưa tận và mọi tên hoặc ký hiệu không in nghiêng đều được giả thiết là một thẻ từ.[1] Để tiện ghi chép, các luật sinh có cùng chưa tận ở vế trái sẽ được nhóm lại thành một ở vế phải, trong đó mỗi vế phải được phân cách bởi một gạch đứng | và được đọc là "hoặc".

Thí dụ 2.1. Nhiều thí dụ trong chương này sử dụng các biểu thức chứa các ký số và các dấu cộng, trừ, thí dụ, 9–5+2, 3–1, và 7. Bởi vì dấu cộng hoặc dấu trừ phải xuất hiện giữa hai ký số, chúng ta sẽ xem những biểu thức như thế là "các *danh sách* (list) ký số được phân cách bởi các dấu cộng hoặc trừ." Văn phạm sau đây mô tả cú pháp của những biểu thức này. Các luật sinh là:

$$list \rightarrow list + digit \quad (2.2)$$
$$list \rightarrow list - digit \quad (2.3)$$
$$list \rightarrow digit \quad (2.4)$$
$$digit \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \quad (2.5)$$

[1] Các chữ cái in nghiêng sẽ được dùng cho các mục đích khác khi nghiên cứu chi tiết về văn phạm trong Chương 4. Thí dụ chúng ta sẽ dùng *X*, *Y*, *Z* để nói về một ký hiệu có thể là thẻ từ hoặc chưa tận. Tuy nhiên tên in nghiêng chứa hai hoặc nhiều ký tự sẽ tiếp tục biểu diễn cho chưa tận.

Các vế phải của ba luật sinh với chưa tận *list* ở vế trái có thể nhóm lại:

list → *list* + *digit* | *list* – *digit* | *digit*

Theo qui ước, các thẻ từ của văn phạm là các ký hiệu

+ – 0 1 2 3 4 5 6 7 8 9

Các chưa tận là các tên in nghiêng *list* và *digit*, với *list* là *ký hiệu khởi đầu* bởi vì luật sinh của nó được trình bày trước tiên. □

Chúng ta nói một luật sinh *là của* một chưa tận nếu chưa tận xuất hiện ở vế trái của luật sinh đó. Chuỗi thẻ từ là một chuỗi gồm zero hoặc nhiều thẻ từ. Chuỗi chứa zero thẻ từ, ký hiệu là ε, được gọi là *chuỗi rỗng* (empty string).

Một văn phạm *dẫn xuất* (derive)[2] các chuỗi (hoặc phái sinh các chuỗi), bắt đầu là ký hiệu khởi đầu và thay thế lập đi lập lại một chưa tận bằng vế phải của một luật sinh của chưa tận đó. Các chuỗi thẻ từ có thể được dẫn xuất từ ký hiệu khởi đầu tạo thành *ngôn ngữ* (language) được định nghĩa bởi văn phạm đó.

Thí dụ 2.2. Ngôn ngữ được định nghĩa bởi văn phạm của Thí dụ 2.1 bao gồm các danh sách ký số được phân cách bởi các dấu cộng hoặc trừ.

Mười luật sinh cho chưa tận *digit* cho phép nó đại diện cho một trong số các thẻ từ 0, 1, . . . , 9. Từ luật sinh (2.4), bản thân một ký số là một danh sách. Các luật sinh (2.2) và (2.3) biểu diễn sự kiện là nếu chúng ta lấy một danh sách bất kỳ và cho vào sau nó một dấu cộng hoặc dấu trừ rồi sau đó là một ký số khác, chúng ta sẽ được một danh sách mới.

Rõ ràng các luật sinh (2.2) đến (2.5) đều cần cho việc định nghĩa ngôn ngữ đang được chúng ta xem xét. Chẳng hạn chúng ta có thể suy ra rằng 9–5+2 là một *list* như sau.

a) 9 là một *list* theo luật sinh (2.4) bởi vì 9 là một *digit*.

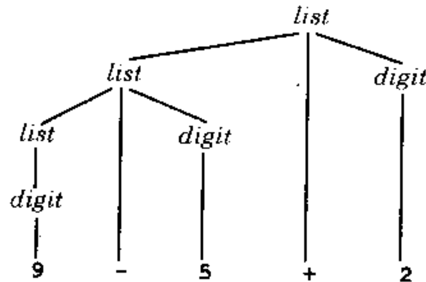
b) 9–5 là một *list* theo luật sinh (2.3) bởi vì 9 là một *list* và 5 là một *digit*.

c) 9–5+2 là một *list* theo luật sinh (2.2) vì 9–5 là một *list* và 2 là một *digit*.

Suy luận này được minh họa qua cây của Hình 2.2. Mỗi nút trong cây được gắn nhãn bằng một ký hiệu văn phạm. Một nút nội và các con của nó tương ứng với một luật sinh; nút nội tương ứng với vế trái của luật sinh, các con tương ứng với vế phải. Những cây như thế được gọi là *cây phân tích cú pháp* (parse tree) và sẽ được thảo luận ở bên dưới. □

---

[2] Các nhà ngôn ngữ học dịch là *phái sinh*. Chúng tôi xem như đây là một gợi ý chọn lựa và có khi dùng từ phái sinh thay cho dẫn xuất. (ND)



Hình 2.2. Cây phân tích cú pháp cho 9–5+2 theo văn phạm trong Thí dụ 2.1.

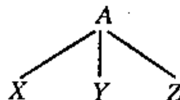
Thí dụ 2.3. Một loại danh sách hơi khác là chuỗi các câu lệnh được phân cách bởi dấu chấm phẩy trong các khối **begin-end** của ngôn ngữ Pascal. Một đặc điểm của những danh sách như thế là giữa các thẻ từ **begin-end** có thể là một danh sách rỗng. Chúng ta có thể xây dựng một văn phạm cho các khối **begin-end** bằng các luật sinh:

block → **begin** *opt_stmts* **end**
opt_stmts → *stmt_list* | ε
stmt_list → *stmt_list* **;** *stmt* | *stmt*

Chú ý rằng đối với *opt_stmts* (optional statement, câu lệnh có thể có hoặc không cũng được), vế phải thứ hai có thể có là ε, biểu thị cho chuỗi rỗng. Nghĩa là *opt_stmts* có thể được thay bằng một chuỗi rỗng, vì thế một *block* có thể chỉ chứa chuỗi có hai thẻ **begin-end**. Chú ý rằng các luật sinh cho *stmt_list* thì tương tự như những luật sinh cho *list* trong Thí dụ 2.1, với dấu chấm phẩy nằm ở vị trí một toán tử số học và *stmt* ở vị trí của *digit*. Chúng ta không trình bày các luật sinh cho *stmt*. Nói cụ thể, chúng ta sẽ thảo luận về các luật sinh thích hợp cho nhiều loại câu lệnh, chẳng hạn câu lệnh **if**, câu lệnh gán, vân vân. □

Cây phân tích cú pháp

Một *cây phân tích cú pháp* (parse tree) cho thấy bằng hình ảnh xem làm thế nào dẫn xuất ra một chuỗi của ngôn ngữ từ *ký hiệu khởi đầu* (start) của một văn phạm. Nếu chưa tận A có luật sinh $A \to XYZ$ thì cây phân tích cú pháp có thể có một nút nội có nhãn A và ba con có nhãn lần lượt từ trái sang phải là X, Y và Z.



Về hình thức, cho trước một văn phạm phi ngữ cảnh, một cây phân tích cú pháp là một cây có những đặc tính sau:

1. Gốc co nhãn là ký hiệu khởi đầu.
2. Mỗi nút lá có nhãn là một thẻ từ hoặc ε.
3. Mỗi nút nội có nhãn là một chưa tận.
4. Nếu A là một chưa tận, là nhãn của một nút nội nào đó và $X_1, X_2, \ldots, X_n$ là nhãn cho các con của nút đó từ trái sang phải thì $A \rightarrow X_1X_2 \ldots X_n$ là một luật sinh. Ở đây $X_1, X_2, \ldots, X_n$ biểu thị cho các ký hiệu tận hoặc chưa tận. Trường hợp đặc biệt nếu $A \rightarrow \varepsilon$ thì một nút được gán nhãn A có thể có một con duy nhất có nhãn ε.

Thí dụ 2.4. Trong Hình 2.2, gốc được gắn nhãn *list*, là ký hiệu khởi đầu của văn phạm trong Thí dụ 2.1. Các con của gốc được gắn nhãn từ trái sang phải là *list*, + và *digit*. Chú ý rằng

list → *list* + *digit*

là một luật sinh trong văn phạm của Thí dụ 2.1. Với dấu – chúng ta cũng có một mẫu như thế được lặp lại ở con bên trái của gốc, và ba nút được gắn nhãn *digit* đều có một con có nhãn là một ký số. □

Các nút lá của một cây phân tích cú pháp khi được đọc từ trái sang phải tạo thành *hoa lợi* (yield) của cây, đó là chuỗi *được sinh ra* hoặc *được dẫn xuất* từ chưa tận nằm tại gốc của cây. Trong Hình 2.2, hoa lợi là 9–5+2. Trong hình đó, tất cả các nút lá đều được trình bày ở mức thấp nhất. Nhưng không nhất thiết phải vẽ thẳng các nút lá như trên. Mọi cây đều có một thứ tự tự nhiên từ trái sang phải cho các nút lá của nó dựa trên ý tưởng là nếu a và b là hai con cùng cha, và a ở bên trái của b thì tất cả các hậu duệ của a đều ở bên trái các hậu duệ của b.

Một định nghĩa khác cho ngôn ngữ được sinh ra bởi một văn phạm là xem nó như một tập các chuỗi có thể sinh ra từ một cây phân tích cú pháp nào đó. Quá trình tìm cây phân tích cú pháp cho một chuỗi thẻ từ được gọi là phân tích cú pháp chuỗi đó.

Tính đa nghĩa

Chúng ta phải cẩn thận khi nói đến cấu trúc của một chuỗi theo một văn phạm. Mặc dù rõ ràng là mỗi cây phân tích cú pháp đều dẫn xuất chính xác chuỗi được đọc từ các nút lá nhưng một văn phạm có thể có nhiều cây phân tích cú pháp sinh ra chuỗi thẻ từ đã cho. Một văn phạm như thế được gọi là *đa nghĩa* (ambiguous).[3] Để chứng tỏ một văn phạm là đa nghĩa, chúng ta cần phải tìm được một chuỗi thẻ từ có nhiều cây phân

[3] Nguyên gốc là hai nghĩa; một số tác giả gọi là nhập nhằng. Ở đây chúng tôi dùng *đa nghĩa* để dịch thuật ngữ ambiguous và *đơn nghĩa* hay *không đa nghĩa* để dịch thuật ngữ unambiguous. (ND)

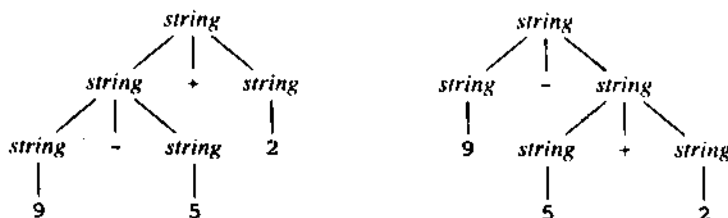
tích cú pháp. Bởi vì một chuỗi với nhiều cây phân tích cú pháp thường có nhiều nghĩa, để biên dịch các chương trình ứng dụng chúng ta cần thiết kế các văn phạm đơn nghĩa, hoặc sử dụng các văn phạm đa nghĩa kèm với các qui tắc bổ sung nhằm giải quyết tính chất đa nghĩa.

Thí dụ 2.5. Giả sử chúng ta không phân biệt giữa *digit* và *list* như trong Thí dụ 2.1. Khi đó có thể viết lại văn phạm là

string → *string* + *string* | *string* - *string* | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Trộn khái niệm *digit* và *list* thành chưa tận *string* sẽ rất có nghĩa bởi vì một *digit* chỉ là trường hợp đặc biệt của một *list*.

Tuy nhiên Hình 2.3 cho thấy rằng một biểu thức như 9–5+2 bây giờ có nhiều cây phân tích cú pháp. Hai cây cho 9–5+2 tương ứng với hai cách đưa dấu ngoặc vào biểu thức đó: (9–5)+2 và 9–(5+2). Biểu thức thứ hai cho ra kết quả là 2 chứ không phải là 6. Văn phạm của Thí dụ 2.1 không chấp nhận cách diễn giải này. □



Hình 2.3. Hai cây phân tích cú pháp cho 9–5+2.

Tính kết hợp của toán tử

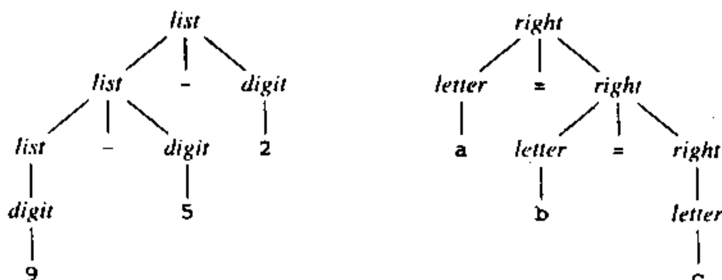
Theo qui ước, 9+5+2 tương đương với (9+5)+2 và 9–5–2 tương đương với (9–5)–2. Khi một toán hạng như 5 có các toán tử ở cả bên phải lẫn bên trái, chúng ta cần có những qui định để quyết định xem toán tử nào sẽ nhận toán hạng đó. Chúng ta nói rằng toán tử + *kết hợp về bên trái* bởi vì toán hạng có các dấu cộng ở cả hai bên của nó sẽ được toán tử bên trái nhận. Trong phần lớn ngôn ngữ lập trình, bốn toán tử số học là *cộng* (addition), *trừ* (substraction), *nhân* (multiplication) và *chia* (division) đều có *tính kết hợp trái* (left associativity).

Một số toán tử thông dụng khác như toán tử lấy lũy thừa có tính kết hợp phải. Một thí dụ khác, toán tử gán = trong C có tính kết hợp phải; trong C, biểu thức a=b=c được xử lý giống như biểu thức a=(b=c).

Các chuỗi như a=b=c có một toán tử kết hợp phải được sinh ra từ văn phạm sau:

right → *letter* = *right* | *letter*
letter → a | b | . . . | z

Khác biệt giữa một cây phân tích cú pháp cho toán tử kết hợp trái như – với cây phân tích cú pháp cho toán tử kết hợp phải như = được trình bày trong Hình 2.4. Chú ý rằng cây cho 9-5-2 phát triển xuống dưới hướng về bên trái, còn cây cho a=b=c phát triển hướng sang bên phải.



Hình 2.4. Các cây phân tích cú pháp cho các toán tử kết hợp trái va kết hợp phải.

Thứ bậc của các toán tử

Xét biểu thức 9+5*2. Có hai cách diễn giải biểu thức này: (9+5)*2 hoặc 9+(5*2). Tính kết hợp của + và * không giải quyết được tính đa nghĩa. Vì lý do này mà chúng ta cần phải biết được *thứ bậc* (precedence)[4] tương đối của các toán tử khi có sự hiện diện nhiều loại toán tử.

Chúng ta nói rằng toán tử * *có thứ bậc cao hơn* toán tử + nếu * lấy các toán hạng của nó trước toán tử +. Trong số học, phép nhân và phép chia có thứ bậc cao hơn phép cộng và phép trừ. Vì thế * lấy 5 trước trong cả hai biểu thức 9+5*2 và 9*5+2; nghĩa là những biểu thức này lần lượt tương đương với 9+(5*2) và (9*5)+2.

Cú pháp cho biểu thức. Văn phạm cho các biểu thức số học có thể được xây dựng từ một bảng trình bày tính kết hợp và thứ bậc của các toán tử. Chúng ta bắt đầu với bốn toán tử số học thông thường và một bảng thứ bậc, trình bày các toán tử theo thứ bậc ngày càng cao, các toán tử có cùng thứ bậc ở trên cùng một hàng:

kết hợp trái: + –
kết hợp trái: * /

Chúng ta tạo ra hai chưa tận *expr* và *term* cho hai mức thứ bậc, và một chưa tận

[4] Còn được dịch là độ ưu tiên. Chúng tôi dành thuật ngữ "độ ưu tiên" để dịch priority. (ND)

factor nữa để tạo ra những đơn vị cơ bản trong biểu thức. Các đơn vị cơ bản hiện có trong biểu thức là các *ký số* (digit) và các biểu thức có đóng mở ngoặc đơn.

factor → **digit** | (*expr*)

Bây giờ hãy xét đến các toán tử hai ngôi * và / với thứ bậc cao nhất. Bởi vì những toán tử này có tính kết hợp trái, luật sinh của chúng tương tự như luật sinh cho các *list* với tính kết hợp trái.

term → *term* * *factor*
  | *term* / *factor*
  | *factor*

Tương tự, *expr* sinh ra *danh sách* (list) các *term* được phân cách bởi toán tử cộng và trừ.

expr → *expr* + *term*
  | *expr* – *term*
  | *term*

Do vậy chúng ta thu được văn phạm

expr → *expr* + *term* | *expr* – *term* | *term*
term → *term* * *factor* | *term* / *factor* | *factor*
factor → **digit** | (*expr*)

Văn phạm này xử lý một biểu thức như một danh sách các *term* được phân cách bởi dấu * hoặc /. Chú ý rằng bất kỳ một biểu thức nào có đóng mở ngoặc đều là một *factor*, vì thế với các dấu ngoặc chúng ta có thể xây dựng các biểu thức tùy nghi lồng nhiều cấp vào nhau.

Cú pháp các câu lệnh. Từ khóa cho phép chúng ta nhận ra các câu lệnh trong phần lớn các ngôn ngữ. Tất cả mọi câu lệnh Pascal đều bắt đầu bằng một từ khóa ngoại trừ lệnh gán và các lời gọi thủ tục. Một số câu lệnh Pascal được định nghĩa bằng văn phạm (đa nghĩa) sau đây, trong đó thẻ từ **id** biểu thị cho một *định danh* (identifier).

stmt → **id** := *expr*
  | **if** *expr* **then** *stmt*
  | **if** *expr* **then** *stmt* **else** *stmt*
  | **while** *expr* **do** *stmt*
  | **begin** *opt_stmts* **end**

Chưa tận *opt_stmts* sinh ra một danh sách câu lệnh có thể rỗng, được phân cách bởi các dấu chấm phẩy nhờ các luật sinh trong Thí dụ 2.3.

2.3 PHIÊN DỊCH DỰA CÚ PHÁP

Để dịch một kết cấu ngôn ngữ lập trình, trình biên dịch cần phải theo dõi nhiều đại lượng khác ngoài mã lệnh cần tạo ra cho kết cấu. Thí dụ trình biên dịch phải biết *kiểu* (type) của kết cấu, vị trí của chỉ thị đầu tiên trong mã đích, hoặc số lượng các chỉ thị được sinh ra. Vì lý do đó chúng ta sẽ nói một cách trừu tượng về các *thuộc tính* (attribute) đi kèm với kết cấu. Một thuộc tính có thể biểu thị cho một đại lượng bất kỳ, thí dụ như một kiểu, một chuỗi, một vị trí bộ nhớ, hoặc những thứ khác, vân vân.

Trong phần này chúng ta giới thiệu một hệ hình thức có tên gọi là *định nghĩa dựa cú pháp* (syntax-directed definition) nhằm đặc tả việc phiên dịch các kết cấu ngôn ngữ lập trình. Một định nghĩa dựa cú pháp đặc tả việc phiên dịch một kết cấu theo các thuộc tính đi kèm với các thành phần cú pháp của nó. Trong những chương sau, định nghĩa dựa cú pháp được dùng để đặc tả một số quá trình dịch xảy ra trong tiền kỳ của một trình biên dịch.

Chúng ta cũng đưa ra một ký pháp có tính thủ tục hơn được gọi là *lược đồ dịch* (translation scheme) để đặc tả quá trình dịch. Qua suốt chương này chúng ta sử dụng lược đồ dịch để dịch các biểu thức trung vị thành dạng hậu vị. Thảo luận chi tiết về định nghĩa dựa cú pháp và cách cài đặt được đưa ra trong Chương 5.

Ký pháp hậu vị

Ký pháp hậu vị (postfix notation) cho một biểu thức E có thể được định nghĩa qui nạp như sau:

1. Nếu E là một biến hoặc hằng thì ký pháp hậu vị cho E là chính E.
2. Nếu E là một biểu thức có dạng E_1 *op* E_2, trong đó *op* là một toán tử hai ngôi, thì ký pháp hậu vị cho E là E_1' E_2' *op*, trong đó E_1' và E_2' tương ứng là các ký pháp hậu vị cho E_1 và E_2.
3. Nếu E là một biểu thức có dạng (E_1) thì ký pháp hậu vị cho E_1 cũng là ký pháp hậu vị cho E.

Chúng ta không cần dùng các dấu ngoặc trong ký pháp hậu vị bởi vì vị trí và ngôi (số lượng các đối) của các toán tử chỉ cho phép có một cách diễn giải duy nhất cho một biểu thức hậu vị. Thí dụ ký pháp hậu vị cho `(9-5)+2` là `95-2+` và ký pháp hậu vị cho `9-(5+2)` là `952+-`.

Định nghĩa dựa cú pháp

Định nghĩa dựa cú pháp (syntax-directed definition) sử dụng văn phạm phi ngữ cảnh để đặc tả cấu trúc cú pháp của nguyên liệu. Nó liên kết mỗi ký hiệu văn phạm với một tập thuộc tính, và mỗi luật sinh với một tập *qui tắc ngữ nghĩa* (semantic rule) để tính giá trị các thuộc tính đi kèm với những ký hiệu có trong luật sinh. Văn phạm và tập

qui tắc ngữ nghĩa cấu tạo nên định nghĩa dựa cú pháp.

Phiên dịch (translation) là một *ánh xạ nguyên liệu–thành phẩm* (input-output mapping). Thành phẩm cho mỗi nguyên liệu x được đặc tả bằng cách sau. Trước tiên là xây dựng cây phân tích cú pháp cho x. Giả sử nút n trong cây được gán nhãn là ký hiệu văn phạm X. Chúng ta viết $X.a$ để biểu thị giá trị thuộc tính a của X tại nút đó. Giá trị của $X.a$ tại n được tính ra nhờ qui tắc ngữ nghĩa cho thuộc tính a đi kèm với luật sinh X được dùng tại nút n. Một cây phân tích cú pháp có trình bày các giá trị thuộc tính tại mỗi nút được gọi là *cây phân tích cú pháp có chú giải* (annotated parse tree).

Thuộc tính tổng hợp

Một thuộc tính được gọi là *tổng hợp* (synthesized) nếu giá trị của nó tại một nút trong cây được xác định từ giá trị tại các con của nút đó. Thuộc tính tổng hợp có một đặc tính đáng chú ý, đó là chúng có thể được ước lượng trong khi duyệt cây phân tích cú pháp từ dưới lên. Trong chương này chúng ta chỉ dùng các thuộc tính tổng hợp; Chương 5 sẽ xem xét thêm các *thuộc tính kế thừa* (inherited attribute).

Thí dụ 2.6. Định nghĩa dựa cú pháp để dịch biểu thức các ký số phân cách bởi các dấu cộng hoặc trừ thành dạng hậu vị được trình bày trong Hình 2.5. Kèm với mỗi chưa tận là một thuộc tính t nhận trị là chuỗi biểu diễn ký pháp hậu vị cho biểu thức được tạo ra bởi chưa tận đó trong một cây phân tích cú pháp.

LUẬT SINH	QUI TẮC NGỮ NGHĨA
$expr \rightarrow expr_1 + term$	$expr.t := expr_1.t \,\Vert\, term.t \,\Vert\, \text{'+'}$
$expr \rightarrow expr_1 - term$	$expr.t := expr_1.t \,\Vert\, term.t \,\Vert\, \text{'-'}$
$expr \rightarrow term$	$expr.t := term.t$
$term \rightarrow 0$	$term.t := \text{'0'}$
$term \rightarrow 1$	$term.t := \text{'1'}$
. . .	. . .
$term \rightarrow 9$	$term.t := \text{'9'}$

Hình 2.5. Định nghĩa dựa cú pháp dịch dạng trung vị thành dạng hậu vị.

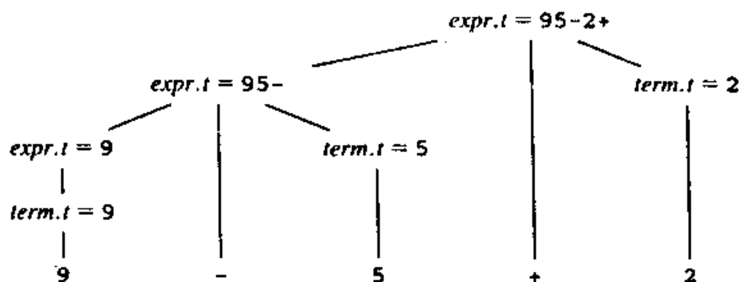
Dạng hậu vị của một ký số là chính nó; thí dụ qui tắc ngữ nghĩa đi kèm với luật sinh $term \rightarrow 9$ định nghĩa $term.t$ là 9 mỗi khi luật sinh này được dùng tại một nút trong một cây phân tích cú pháp. Khi luật sinh $expr \rightarrow term$ được áp dụng, giá trị của $term.t$ trở thành giá trị của $expr.t$.

Luật sinh $expr \rightarrow expr_1 + term$ dẫn xuất ra một biểu thức chứa một toán tử cộng (chỉ số trong $expr_1$ nhằm phân biệt thể hiện của $expr$ ở vế phải với thể hiện ở vế trái). Toán hạng trái của toán tử cộng là $expr_1$ và toán hạng phải là $term$. Qui tắc ngữ nghĩa

$expr.t := expr_1.t \parallel term.t \parallel$ '+'

đi kèm với luật sinh này định nghĩa giá trị của thuộc tính *expr.t* bằng cách ghép nối các dạng hậu vị $expr_1.t$ và *term.t* của các toán hạng trái và phải tương ứng, rồi nối thêm dấu cộng vào đó. Toán tử || trong các qui tắc ngữ nghĩa biểu thị *phép ghép nối* (concatenation).

Hình 2.6 là cây phân tích cú pháp có chú giải tương ứng với cây của Hình 2.2. Giá trị của thuộc tính *t* tại mỗi nút được tính ra nhờ qui tắc ngữ nghĩa đi kèm với luật sinh tại nút đó. Giá trị của thuộc tính tại gốc là ký pháp hậu vị cho chuỗi được sinh ra bởi cây phân tích cú pháp. □



Hình 2.6. Giá trị thuộc tính tại các nút của một cây phân tích cú pháp.

Thí dụ 2.7. Giả sử một *người máy* (robot) có thể được yêu cầu di chuyển từng bước theo các hướng *đông* (east), *bắc* (north), *tây* (west), hoặc *nam* (south) từ vị trí hiện tại. Một *dãy* (sequence) các *chỉ thị* (instruction) được sinh ra bởi văn phạm:

seq → *seq instr* | **begin**
instr → **east** | **north** | **west** | **south**

Dịch chuyển vị trí của người máy khi nhận được dòng nguyên liệu

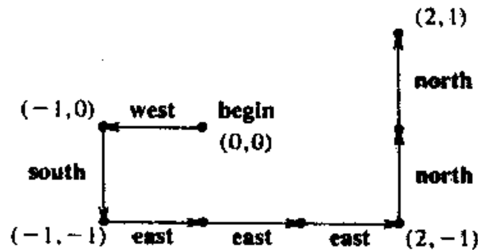
begin west south east east east north north

được trình bày trong Hình 2.7.

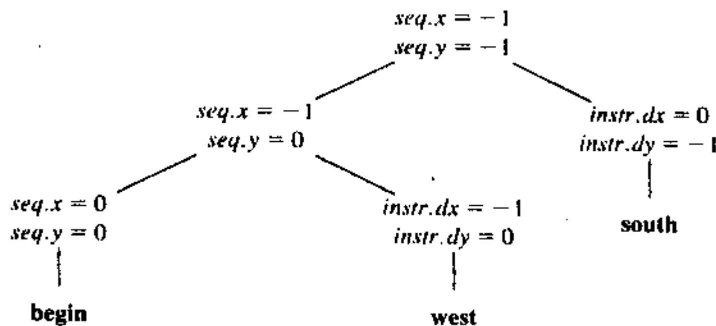
Trong hình này, một vị trí được đánh dấu bởi một cặp (x, y), trong đó x và y biểu thị cho số bước đi tương ứng sang hướng đông và bắc từ vị trí khởi đầu. (Nếu x âm thì người máy đi về hướng tây so với vị trí khởi đầu; tương tự nếu y âm thì người máy đi sang hướng nam.)

Chúng ta hãy xây dựng một định nghĩa dựa cú pháp, cho phép dịch một dãy chỉ thị thành vị trí của người máy. Chúng ta sẽ dùng hai thuộc tính, *seq.x* và *seq.y*, để theo

dõi vị trí khi thực hiện dãy chỉ thị được tạo ra bởi ký hiệu chưa tận *seq*. Khởi đầu, *seq* sinh ra **begin**, và *seq.x*, *seq.y* đều được gán giá trị 0 như được trình bày ở nút nội tận cùng bên trái của cây phân tích cú pháp cho **begin west south** trong Hình 2.8.



Hình 2.7. Theo dõi vị trí của một người máy.



Hình 2.8. Cây phân tích cú pháp có chú giải cho **begin west south**.

Thay đổi về vị trí do từng chỉ thị dẫn xuất từ *instr* được cho bằng các thuộc tính *instr.dx* và *instr.dy*. Chẳng hạn nếu *instr* dẫn xuất ra **west** thì *instr.dx* = −1 và *instr.dy* = 0. Giả sử một dãy *seq* được tạo ra bằng cách cho sau dãy seq_1 một chỉ thị mới *instr*. Vị trí mới của người máy khi đó được cho bằng qui tắc

$$seq.x := seq_1.x + instr.dx$$
$$seg.y := seq_1.y + instr.dy$$

Một định nghĩa dựa cú pháp dịch một dãy chỉ thị thành một vị trí của người máy được đưa ra trong Hình 2.9. □

LUẬT SINH	QUI TẮC NGỮ NGHĨA
$seq \rightarrow$ **begin**	$seq.x := 0$ $seq.y := 0$
$seq \rightarrow seq_1\ instr$	$seq.x := seq_1.x + instr.dx$ $seq.y := seq_1.y + instr.dy$
$instr \rightarrow$ **east**	$instr.dx := 1$ $instr.dy := 0$
$instr \rightarrow$ **north**	$instr.dx := 0$ $instr.dy := 1$
$instr \rightarrow$ **west**	$instr.dx := -1$ $instr.dy := 0$
$instr \rightarrow$ **south**	$instr.dx := 0$ $instr.dy := -1$

Hình 2.9. Định nghĩa dựa cú pháp cho vị trí của người máy.

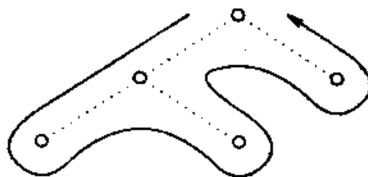
Duyệt theo hướng sâu

Một định nghĩa dựa cú pháp không đưa ra một thứ tự ước lượng cụ thể cho các thuộc tính trên một cây phân tích cú pháp; mọi thứ tự ước lượng một thuộc tính *a* sau tất cả những thuộc tính mà *a* phụ thuộc đều được. Nói chung chúng ta phải ước lượng một số thuộc tính khi lần đầu tiên đến được một nút trong khi duyệt xét cây phân tích cú pháp còn các thuộc tính khác sẽ được ước lượng sau khi đã thăm tất cả các con của nó hoặc tại một điểm nào đó trong lúc thăm các con của nút đó. Các thứ tự ước lượng thích hợp sẽ được thảo luận chi tiết hơn trong Chương 5.

Quá trình phiên dịch trong chương này đều có thể được cài đặt bằng cách ước lượng các qui tắc ngữ nghĩa cho các thuộc tính trên cây phân tích cú pháp theo một thứ tự đã được xác định trước. Duyệt cây sẽ bắt đầu từ gốc và thăm mỗi nút của cây theo một thứ tự nào đó. Trong chương này, các qui tắc ngữ nghĩa sẽ được ước lượng bằng cách sử dụng *lối duyệt theo hướng sâu* (depth-first traversal) được định nghĩa trong Hình 2.10. Khởi đầu duyệt từ gốc và thăm lần lượt (đệ qui) các con của mỗi nút theo thứ tự từ trái sang phải như trong Hình 2.11. Các qui tắc ngữ nghĩa tại một nút đã cho được ước lượng một khi đã thăm hết tất cả các *hậu duệ* (descendant) của nút đó. Lối duyệt cây này được gọi là "theo hướng sâu" bởi vì nó thăm một con (chưa được thăm) của một nút mỗi khi có thể được, vì thế nó có xu hướng thăm các nút xa gốc trước nhất.

```
procedure visit(n: node);
begin
   for mỗi con m của n tính từ trái sang phải do
      visit(m);
   ước lượng các qui tắc ngữ nghĩa tại nút n
end
```

Hình 2.10. Duyệt theo hướng sâu trên một cây.



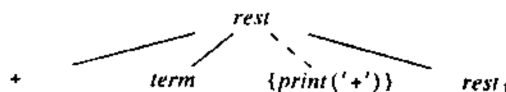
Hình 2.11. Thí dụ về phương pháp duyệt theo hướng sâu trên một cây.

Lược đồ dịch

Trong phần còn lại của chương này, chúng ta sử dụng một đặc tả kiểu thủ tục để định nghĩa một quá trình dịch. *Một lược đồ dịch* (transition scheme) là một văn phạm phi ngữ cảnh, trong đó các *đoạn chương trình* (program fragment), được gọi là *hành động ngữ nghĩa* (semantic action), được gắn vào vế phải của luật sinh. Một lược đồ dịch giống như một định nghĩa dựa cú pháp, ngoại trừ thứ tự ước lượng các qui tắc ngữ nghĩa được trình bày tường minh. Vị trí của hành động cần thực thi được trình bày bằng cách đưa nó vào giữa hai dấu ngoặc móc và viết ở vế phải của một luật sinh giống như dưới đây.

$$rest \rightarrow + \ term \ \{ \ print('+') \ \} \ rest_1$$

Một lược đồ dịch tạo ra *thành phẩm* (output) cho mỗi câu x sinh ra từ văn phạm đã cho bằng cách cho thực thi các hành động này theo đúng thứ tự xuất hiện trong khi duyệt theo hướng sâu trên cây phân tích cú pháp của x. Chẳng hạn xét thử một cây phân tích cú pháp với một nút có nhãn *rest* biểu diễn cho luật sinh ở trên. Hành động { *print*('+') } sẽ được thực hiện sau khi cây con của *term* đã được duyệt xong nhưng trước khi xét cây con của $rest_1$.



Hình 2.12. Một nút lá được xây dựng cho hành động ngữ nghĩa.

Khi vẽ ra một cây phân tích cú pháp cho một lược đồ dịch, chúng ta chỉ rõ một hành động bằng cách xây dựng một nút con cho nó, nối với nút của luật sinh bằng một đường đứt nét. Chẳng hạn phần cây phân tích cú pháp cho luật sinh và hành động ở trên được vẽ như trong Hình 2.12. Nút dành cho một hành động ngữ nghĩa không có con, vì thế nó được thực hiện khi xét nút đó lần đầu.

Đưa ra một bản dịch

Trong chương này, hành động ngữ nghĩa trong lược đồ dịch sẽ ghi kết quả của quá trình phiên dịch vào một tập tin, mỗi lần một chuỗi hoặc một ký tự. Chẳng hạn chúng ta dịch 9-5+2 thành 95-2+ bằng cách in mỗi ký tự trong 9-5+2 đúng một lần mà không phải ghi lại quá trình dịch của các biểu thức con. Khi tạo ra thành phẩm dần dần theo lối này, thứ tự in ra các ký tự sẽ quan trọng.

Chú ý rằng các định nghĩa dựa cú pháp đã được phân tích đến lúc này đều có tính chất quan trọng sau: chuỗi biểu diễn cho bản dịch của chưa tận ở vế trái của mỗi luật sinh là sự ghép nối các bản dịch của các chưa tận ở vế phải theo đúng thứ tự của chúng trong luật sinh và có thể có thêm một số chuỗi khác xen vào giữa. Một định nghĩa dựa cú pháp có tính chất này được xem là *đơn giản* (simple). Thí dụ, xét luật sinh thứ nhất và qui tắc ngữ nghĩa trong định nghĩa dựa cú pháp của Hình 2.5:

LUẬT SINH	QUI TẮC NGỮ NGHĨA	
$expr \rightarrow expr_1 + term$	$expr.t := expr_1.t \;\|\|\; term.t \;\|\|\; \text{'+'}$	(2.6)

Ở đây bản dịch *expr.t* là sự ghép nối các bản dịch của $expr_1$ và *term*, theo sau là ký hiệu +. Chú ý rằng $expr_1$ xuất hiện trước *term* ở vế phải của luật sinh.

Một chuỗi bổ sung xuất hiện giữa *term.t* và $rest_1.t$ trong

LUẬT SINH	QUI TẮC NGỮ NGHĨA	
$rest \rightarrow + \; term \; rest_1$	$rest.t := term.t \;\|\|\; \text{'+'} \;\|\|\; rest_1.t$	(2.7)

nhưng một lần nữa, chưa tận *term* xuất hiện trước $rest_1$ ở vế phải.

Các định nghĩa dựa cú pháp đơn giản có thể được cài đặt với các lược đồ dịch trong đó các hành động sẽ in ra các chuỗi bổ sung theo đúng thứ tự xuất hiện trong định nghĩa đó. Các hành động trong luật sinh dưới đây sẽ in ra chuỗi bổ sung tương ứng trong (2.6) và (2.7):

$$expr \rightarrow expr_1 + term \; \{ \; print(\text{'+'}) \; \}$$
$$rest \rightarrow + \; term \; \{ \; print(\text{'+'}) \; \} \; rest_1$$

Thí dụ 2.8. Hình 2.5 có chứa một định nghĩa đơn giản để dịch các biểu thức thành dạng hậu vị. Một lược đồ dịch suy ra từ định nghĩa này được cho trong Hình 2.13 và cây phân tích cú pháp có các hành động kèm theo cho biểu thức 9-5+2 được trình bày

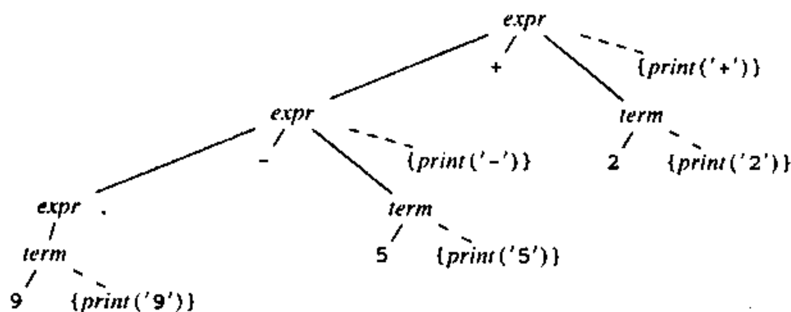
trong Hình 2.14. Chú ý rằng mặc dù Hình 2.6 và 2.14 biểu diễn cùng một ánh xạ *nguyên liệu-thành phẩm*, bản dịch trong hai trường hợp này được xây dựng khác nhau; Hình 2.6 gắn thành phẩm vào gốc của cây, còn trong Hình 2.14 lại in ra dần dần.

expr	→	*expr* + *term*	{ *print*('+') }
expr	→	*expr* − *term*	{ *print*('−') }
expr	→	*term*	
term	→	0	{ *print*('0') }
term	→	1	{ *print*('1') }
	. . .		
term	→	9	{ *print*('9') }

Hình 2.13. Các hành động dịch biểu thức thành ký pháp hậu vị.

Gốc của Hình 2.14 biểu diễn luật sinh thứ nhất trong Hình 2.13. Khi duyệt theo hướng sâu, đầu tiên chúng ta thực hiện tất cả các hành động trong cây con cho toán hạng trái *expr* khi duyệt qua cây con tận trái của gốc. Sau đó chúng ta thăm nút lá + nhưng không thực hiện hành động nào. Kế tiếp chúng ta thực hiện các hành động trong cây con cho toán hạng bên phải là *term* và cuối cùng là thực hiện hành động ngữ nghĩa { *print*('+') }.

Vì các luật sinh cho *term* chỉ có một ký số ở vế phải, ký số đó được in ra bằng các hành động cho các luật sinh đó. Đối với luật sinh *expr* → *term* chúng ta không cần phải cho ra kết quả nào và chỉ cần phải in toán tử bằng hành động của hai luật sinh đầu tiên. Khi duyệt theo hướng sâu, các hành động trong Hình 2.14 sẽ in ra 95−2+. □



Hình 2.14. Các hành động dịch 9−5+2 thành 95−2+.

Xem như một qui tắc tổng quát, phần lớn các phương pháp phân tích cú pháp đều xử lý nguyên liệu của chúng từ trái sang phải theo lối "*thiển cận*" (greedy fashion); nghĩa là trước khi đọc thẻ từ tiếp theo, chúng xây dựng càng nhiều phần cho cây phân

tích cú pháp càng tốt. Trong một lược đồ dịch đơn giản (là lược đồ dẫn xuất từ một định nghĩa dựa cú pháp đơn giản), các hành động cũng được thực hiện theo thứ tự từ trái sang phải. Vì thế để cài đặt một lược đồ dịch đơn giản, chúng ta có thể thực hiện các hành động ngữ nghĩa trong lúc phân tích cú pháp mà không nhất thiết phải xây dựng cây phân tích cú pháp.

2.4 PHÂN TÍCH CÚ PHÁP

Phân tích cú pháp là quá trình xác định xem một chuỗi *thẻ từ* (token) có thể được sinh ra từ một văn phạm hay không. Khi thảo luận về vấn đề này, chúng ta xem như đang xây dựng một cây phân tích cú pháp mặc dù một trình biên dịch có thể không xây dựng một cây như thế. Tuy nhiên *thể phân cú pháp* (parser) phải có khả năng xây dựng nó, nếu không thì việc phiên dịch không bảo đảm được tính đúng đắn.

Trong phần này chúng ta sẽ giới thiệu một phương pháp phân tích cú pháp được dùng để xây dựng các *chương trình dịch dựa cú pháp* (syntax-directed translator). Chương trình C hoàn chỉnh cài đặt lược đồ dịch của Hình 2.13 sẽ được trình bày trong phần kế tiếp. Một khả năng khác là dùng một công cụ phần mềm để tạo trực tiếp chương trình dịch từ một lược đồ dịch. Phần 4.9 có mô tả về một công cụ như thế; nó có thể cài đặt lược đồ dịch của Hình 2.13 mà không cần sửa đổi gì.

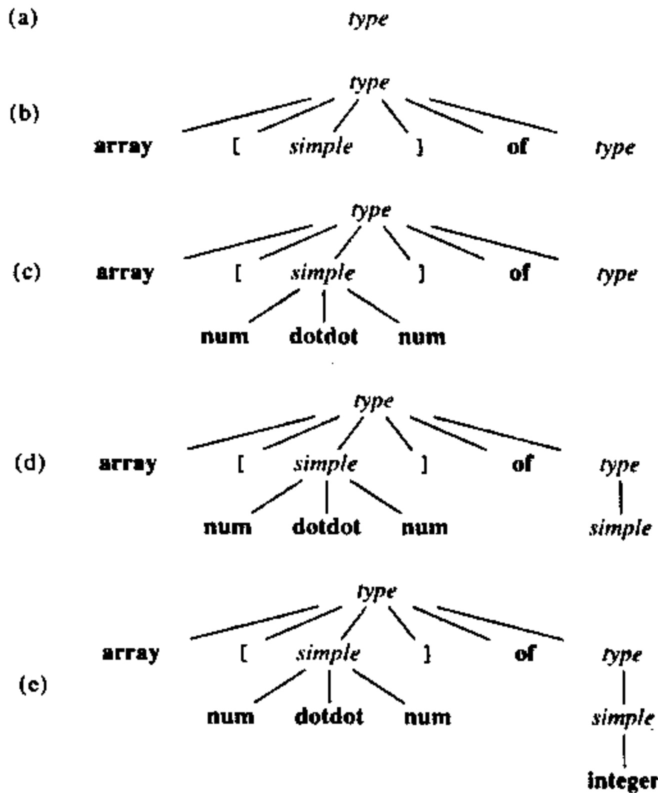
Thể phân cú pháp có thể được xây dựng cho một văn phạm bất kỳ. Tuy nhiên các văn phạm được dùng trong thực tế thường có dạng đặc biệt. Đối với một văn phạm phi ngữ cảnh chúng ta đều có một thể phân cú pháp cần thời gian tối đa là $O(n^3)$ để phân tích cú pháp cho một chuỗi gồm n thẻ từ. Nhưng chi phí thời gian lập phương là quá cao. Cho trước một ngôn ngữ lập trình, nói chung chúng ta có thể xây dựng một văn phạm sao cho có thể phân tích nó một cách nhanh chóng. Các thuật toán tuyến tính là đủ để phân tích mọi ngôn ngữ gặp trong thực hành. Thể phân cú pháp cho các ngôn ngữ lập trình hầu như luôn quét nguyên liệu từ trái sang phải, mỗi lần một thẻ từ.

Phần lớn các phương pháp phân tích đều rơi vào một trong hai lớp, được gọi là *phương pháp từ trên xuống* và *phương pháp từ dưới lên*. Những thuật ngữ này muốn nói đến thứ tự xây dựng các nút trong cây phân tích cú pháp. Trong phương pháp đầu, quá trình xây dựng bắt đầu tại gốc, tiến hành hướng xuống các nút lá, còn trong phương pháp sau thì tiến hành từ các nút lá hướng về gốc. Tính thông dụng của các thể phân cú pháp từ trên xuống là do có thể xây dựng được chúng một cách hiệu quả theo lối thủ công. Tuy nhiên phân tích cú pháp từ dưới lên lại có thể xử lý được một lớp văn phạm và lược đồ dịch phong phú hơn, vì thế các công cụ phần mềm giúp xây dựng các thể phân cú pháp một cách trực tiếp từ các văn phạm đều có xu hướng sử dụng các phương pháp từ dưới lên.

Phân tích cú pháp từ trên xuống

Chúng ta sẽ giới thiệu kỹ thuật phân tích cú pháp từ trên xuống qua việc xem xét một văn phạm rất thích hợp cho lớp phương pháp này. Ở những đoạn sau chúng ta sẽ xét đến phương pháp xây dựng các thể phân cú pháp từ trên xuống nói chung. Văn phạm sau đây tạo ra một tập con các kiểu dữ liệu của Pascal. Chúng ta dùng thẻ từ **dotdot** thay cho ". ." để nhấn mạnh rằng chuỗi ký tự này được xử lý như một đơn vị.

type → *simple*
　　| ↑ **id**
　　| **array** [*simple*] **of** *type*
simple → **integer** (2.8)
　　| **char**
　　| **num dotdot num**



Hình 2.15. Các bước khi xây dựng một cây phân tích cú pháp từ trên xuống.

Xây dựng cây phân tích cú pháp theo lối từ trên xuống được thực hiện từ gốc, với nhãn là chưa tận khởi đầu, rồi thực hiện hai bước sau đây lặp đi lặp lại (xem thí dụ trong Hình 2.15).

1. Tại nút *n* có nhãn là chưa tận *A*, chọn một trong những luật sinh cho *A* và cho các ký hiệu ở vế phải của luật sinh này làm con của *n*.
2. Tìm nút kế tiếp để xây dựng một cây con tại đó.

Đối với một số văn phạm, các bước trên có thể được cài đặt trong khi quét chuỗi nguyên liệu từ trái sang phải. Thẻ từ hiện đang được quét trong nguyên liệu thường được gọi là *ký hiệu sải với* (lookahead symbol). Lúc ban đầu, ký hiệu sải với là thẻ từ đầu tiên, nghĩa là thẻ từ tận trái của chuỗi nguyên liệu. Hình 2.16 minh họa việc phân tích cú pháp chuỗi

array [**num dotdot num**] **of integer**

Ban đầu, thẻ từ **array** là ký hiệu sải với và phần đã biết của cây phân tích cú pháp bao gồm gốc, có nhãn là chưa tận khởi đầu *type* trong Hình 2.16(a). Mục đích là xây dựng phần còn lại của cây sao cho chuỗi được sinh ra bởi cây sẽ *so khớp được* (đối sánh được) với chuỗi nguyên liệu.

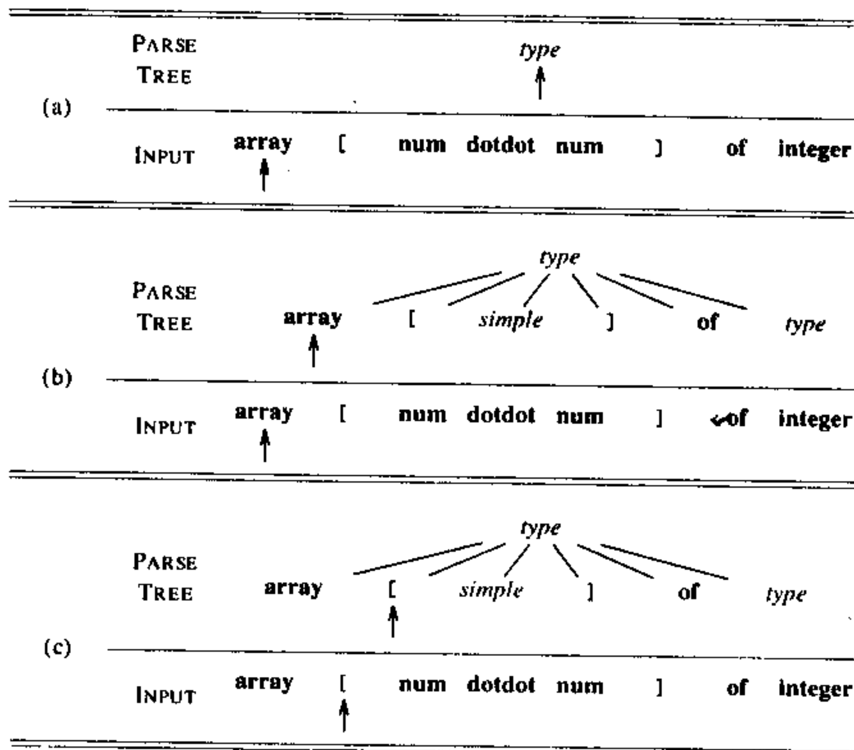
Để có được một đối sánh, chưa tận *type* trong Hình 2.16(a) phải dẫn xuất ra một chuỗi bắt đầu bằng sải với **array**. Trong văn phạm (2.8), chỉ có một luật sinh cho *type* có thể dẫn xuất một chuỗi như thế nên chúng ta sẽ chọn luật sinh đó và xây dựng các con cho gốc có nhãn là những ký hiệu ở vế phải của luật sinh.

Mỗi hình trong số ba hình của Hình 2.16 có các mũi tên chỉ ra ký hiệu sải với trong nguyên liệu và nút đang được xét trong cây phân tích cú pháp. Khi xây dựng các con của một nút thì bước kế tiếp chúng ta sẽ xét con tận trái. Trong Hình 2.16(b), các con vừa được xây dựng tại gốc, và con tận trái với nhãn là **array** sẽ được xét.

Trong cây phân tích cú pháp, khi một tận và nút cho tận đó đối sánh được với ký hiệu sải với thì chúng ta dịch tới trước một bước, cả ở cây phân tích cú pháp và ở nguyên liệu. Thẻ từ kế tiếp trong nguyên liệu trở thành sải với mới và con kế tiếp trong cây sẽ được xét. Trong Hình 2.16(c), mũi tên trong cây đã được dịch tới con kế tiếp của gốc và mũi tên trong nguyên liệu đã được dịch tới thẻ từ tiếp theo là [. Sau khi dịch tới trước, mũi tên trong cây sẽ chỉ đến con có nhãn là chưa tận *simple*. Khi một nút có nhãn là một chưa tận được xét đến, chúng ta sẽ lặp lại quá trình chọn một luật sinh cho chưa tận đó.

Nói chung, việc chọn một luật sinh cho một ký hiệu chưa tận có thể được thực hiện theo kiểu *thử và sai*; nghĩa là chúng ta có thể phải thử một luật sinh rồi phải quay lại để thử một luật sinh khác nếu luật sinh thứ nhất không phù hợp. Một luật sinh sẽ không phù hợp nếu sau khi dùng nó, chúng ta không thể tạo ra một cây so khớp được

với chuỗi nguyên liệu. Tuy nhiên có một trường hợp đặc biệt có tên là *phân tích cú pháp dự đoán* (predictive parsing) không có tình trạng phải thử lại (trở lui).



Hình 2.16. Phân tích cú pháp từ trên xuống trong khi đọc nguyên liệu từ trái sang phải.

Phân tích cú pháp dự đoán

Phân tích cú pháp đệ qui-xuống (recursive-descent parsing) là một phương pháp phân tích từ trên xuống, trong đó chúng ta cho thực thi một tập thủ tục đệ qui để xử lý chuỗi nguyên liệu. Mỗi chưa tận của văn phạm được kèm với một thủ tục. Ở đây chúng ta xét một hình thái đặc biệt của phân tích cú pháp đệ qui-xuống, đó là *phân tích cú pháp dự đoán* (predictive parsing), trong đó sải với giúp xác định đúng thủ tục cần chọn cho mỗi chưa tận. Loạt thủ tục được gọi trong khi xử lý chuỗi nguyên liệu ngầm định nghĩa một cây phân tích cú pháp cho nguyên liệu.

Thể phân cú pháp dự đoán trong Hình 2.17 gồm có các thủ tục cho chưa tận *type* và *simple* của văn phạm (2.8) và một thủ tục bổ sung *match*. Chúng ta dùng *match* nhằm đơn giản hóa đoạn mã cho *type* và *simple*; nó dịch tới thẻ từ tiếp theo nếu đối *t*

của nó so khớp được với sải với. Vì thế *match* làm thay đổi biến *lookahead*, đó là thẻ từ hiện đang được quét trong nguyên liệu.

```
procedure match(t: token);
begin
    if lookahead = t then
        lookahead := nexttoken
    else error
end;

procedure type;
begin
    if lookahead thuộc {integer, char, num} then
        simple
    else if lookahead = '↑' then begin
        match('↑'); match(id)
    end
    else if lookahead = array then begin
        match(array); match('['); simple; match(']'); match(of); type
    end
    else error
end;

procedure simple;
begin
    if lookahead = integer then
        match(integer)
    else if lookahead = char then
        match(char)
    else if lookahead = num then begin
        match(num); match(dotdot); match(num)
    end
    else error
end;
```

Hình 2.17. Đoạn mã giả cho một thể phân cú pháp dự đoán.

Phân tích cú pháp bắt đầu bằng một lời gọi đến thủ tục cho chưa tận khởi đầu *type*. Với cùng nguyên liệu như trong Hình 2.16, thoạt đầu *lookahead* là thẻ từ thứ nhất **array**. Thủ tục *type* thực thi đoạn mã

match(**array**); *match*('['); *simple*; *match*(']'); *match*(**of**); *type* (2.9)

tương ứng với vế phải của luật sinh

type → **array** [*simple*] **of** *type*

Chú ý rằng mỗi tận ở vế phải được đối sánh với ký hiệu sải với và mỗi chưa tận ở vế phải dẫn đến một lời gọi thủ tục của nó.

Với nguyên liệu của Hình 2.16, sau khi các thẻ từ **array** và [đã đối sánh, ký hiệu sải với sẽ là **num**. Lúc này, thủ tục *simple* được gọi và đoạn mã

match(**num**); *match*(**dotdot**); *match*(**num**)

trong phần thân của nó được thực thi.

Ký hiệu sải với hướng dẫn việc chọn lựa luật sinh. Nếu vế phải của một luật sinh bắt đầu bằng một thẻ từ thì luật sinh có thể được dùng khi ký hiệu sải với đối sánh được với thẻ từ. Bây giờ hãy xét một vế phải bắt đầu bằng một chưa tận như

type → *simple* (2.10)

Luật sinh này được dùng nếu ký hiệu sải với có thể được sinh ra từ *simple*. Thí dụ trong khi thực hiện đoạn mã (2.9), giả sử ký hiệu sải với là **integer** khi quyền điều khiển được trao cho lời gọi thủ tục *type*. Không có luật sinh nào cho *type* bắt đầu bằng thẻ từ **integer**. Tuy nhiên một luật sinh cho *simple* lại có, vì thế luật sinh (2.10) được dùng bằng cách yêu cầu *type* gọi thủ tục *simple* trên sải với **integer**.

Phân tích cú pháp dự đoán dựa vào thông tin về những ký hiệu đầu tiên được sinh ra bởi vế phải của một luật sinh. Nói chính xác hơn, gọi α là vế phải của một luật sinh cho chưa tận A. Chúng ta định nghĩa FIRST(α) là tập thẻ từ xuất hiện như những ký hiệu đầu tiên của một hoặc nhiều chuỗi được sinh ra từ α. Nếu α là ε hoặc có thể sinh ra ε, thì ε cũng thuộc FIRST(α).[5] Chẳng hạn,

FIRST(*simple*) = { **integer, char, num** }
FIRST(↑ **id**) = { ↑ }
FIRST(**array** [*simple*] **of** *type*) = { **array** }

Trong thực hành, do có nhiều vế phải của các luật sinh bắt đầu bằng các thẻ từ nên đã làm đơn giản việc xây dựng các tập FIRST. Một thuật toán để tính FIRST sẽ được trình bày trong Phần 4.4.

Các tập FIRST phải được xét đến nếu có hai luật sinh $A \to \alpha$ và $A \to \beta$. Phân tích đệ qui xuống không trở lui đòi hỏi rằng FIRST(α) và FIRST(β) phải rời nhau. Vì thế ký

[5] Các luật sinh có ε ở vế phải làm cho việc xác định các ký hiệu đầu tiên sinh ra từ một chưa tận thêm phức tạp. Thí dụ nếu chưa tận B có thể dẫn xuất một chuỗi rỗng và có một luật sinh $A \to BC$ thì ký hiệu đầu tiên được sinh ra từ C cũng có thể là ký hiệu đầu tiên sinh ra từ A. Nếu C cũng sinh ra ε thì cả FIRST(A) và FIRST(BC) đều chứa ε.

hiệu sải với có thể được dùng để quyết định xem phải sử dụng luật sinh nào; nếu ký hiệu sải với thuộc FIRST(α) thì α được dùng. Ngược lại nếu ký hiệu sải với thuộc FIRST(β) thì dùng β.

Sử dụng ε-luật sinh

Các luật sinh có ε ở vế phải cần phải được xử lý đặc biệt. Thể phân cú pháp đệ qui-xuống sẽ dùng luật sinh ε làm trường hợp mặc định khi không sử dụng được một luật sinh khác. Chẳng hạn như xét:

stmt → **begin** *opt_stmts* **end**
opt_stmts → *stmt_list* | ε

Khi phân tích cú pháp cho *opt_stmts*, nếu ký hiệu sải với không thuộc FIRST(*stmt_list*) thì luật sinh ε được dùng. Chọn lựa này hoàn toàn chính xác nếu ký hiệu sải với là **end**. Mọi ký hiệu sải với không phải là **end** sẽ gây ra lỗi và được phát hiện trong khi phân tích *stmt*.

Thiết kế thể phân cú pháp dự đoán

Thể phân cú pháp dự đoán là một chương trình gồm có một thủ tục cho mỗi ký hiệu chưa tận. Mỗi thủ tục sẽ thực hiện hai việc:

1. Nó quyết định xem sẽ dùng luật sinh nào nhờ vào ký hiệu sải với. Luật sinh có vế phải α sẽ được dùng nếu ký hiệu sải với thuộc FIRST(α). Nếu có xung đột giữa hai vế phải trên một ký hiệu sải với nào đó thì chúng ta không thể dùng phương pháp phân tích cú pháp này trên văn phạm đang có. Một luật sinh có ε ở vế phải sẽ được dùng nếu sải với không thuộc tập FIRST của một vế phải khác.

2. Thủ tục sẽ sử dụng một luật sinh bằng cách mô phỏng vế phải. Một chưa tận gây ra một lời gọi đến thủ tục cho ký hiệu đó, và một thẻ từ đối sánh được với ký hiệu sải với gây ra hành động đọc thẻ từ kế tiếp. Nếu tại một thời điểm nào đó, thẻ từ trong luật sinh không đối sánh được với ký hiệu sải với thì sẽ phải khai báo một lỗi. Hình 2.17 là kết quả áp dụng những qui tắc này cho văn phạm (2.8).

Cũng giống như khi tạo ra một lược đồ dịch bằng cách mở rộng một văn phạm, một chương trình dịch dựa cú pháp có thể được tạo ra bằng cách mở rộng một thể phân cú pháp dự đoán. Một thuật toán thực hiện điều này sẽ được cho trong Phần 5.5. Hiện tại, chúng ta chỉ cần dùng phương pháp xây dựng khá hạn chế dưới đây bởi vì các lược đồ dịch được cài đặt trong chương này không gắn các thuộc tính vào các ký hiệu chưa tận:

1. Xây dựng một thể phân cú pháp dự đoán, bỏ qua các hành động trong các luật sinh.

2. Sao chép các hành động từ lược đồ dịch vào thể phân cú pháp này. Nếu một hành động xuất hiện sau ký hiệu văn phạm X trong luật sinh p thì nó được sao chép sau đoạn mã cài đặt X. Ngược lại nếu nó xuất hiện ở đầu luật sinh thì nó được sao chép ngay trước đoạn mã cài đặt luật sinh.

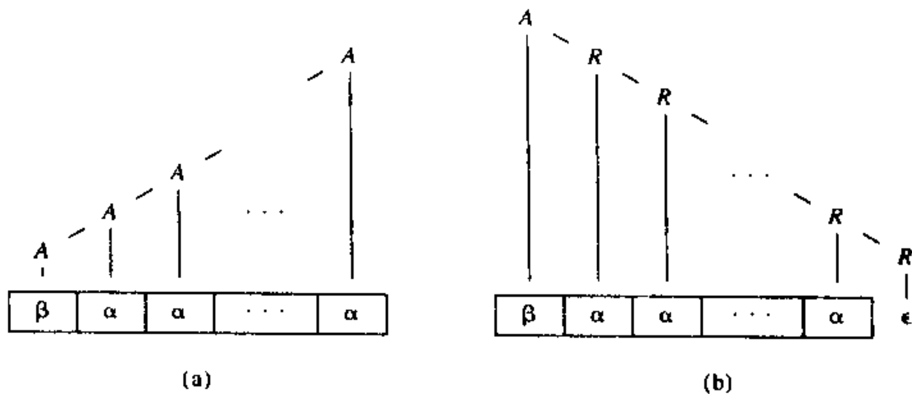
Chúng ta sẽ xây dựng một chương trình dịch như thế ở phần tiếp theo.

Đệ qui trái

Rất có thể một thể phân cú pháp đệ qui-xuống bị "lạc" vào một vòng lặp vô tận. Tình huống này xảy ra với các luật sinh đệ qui trái như

$$expr \rightarrow expr + term$$

trong đó ký hiệu tận trái ở vế phải giống với chưa tận ở vế trái của luật sinh. Giả sử rằng thủ tục cho *expr* quyết định áp dụng luật sinh này. Vế phải bắt đầu với *expr* nên thủ tục cho *expr* được gọi đệ qui và thể phân cú pháp sẽ lặp vô tận. Chú ý rằng ký hiệu sải với chỉ thay đổi khi một tận ở vế phải đối sánh được. Do luật sinh bắt đầu bằng chưa tận *expr*, không có thay đổi nào đối với chuỗi nguyên liệu xảy ra giữa các lần gọi đệ qui và như thế gây ra một vòng lặp vô tận.



Hình 2.18. Sinh ra một chuỗi theo kiểu đệ qui trái và đệ qui phải.

Có thể loại bỏ một luật sinh đệ qui trái bằng cách viết lại luật sinh đó. Xét một chưa tận A với hai luật sinh

$$A \rightarrow A\alpha \mid \beta$$

trong đó α và β là các dãy ký hiệu tận và chưa tận không bắt đầu bằng A. Thí dụ trong luật sinh

$$expr \rightarrow expr + term \mid term$$

$A = expr$, $\alpha = + \ term$, và $\beta = term$

Ký hiệu chưa tận A là đệ qui trái bởi vì luật sinh $A \rightarrow A\alpha$ có A là ký hiệu tận trái ở vế phải. Áp dụng lặp lại luật sinh này sẽ sinh ra một chuỗi các α ở bên phải của A như trong Hình 2.18(a). Cuối cùng khi A được thay bằng β, chúng ta có một β, theo sau là một chuỗi gồm zero hoặc nhiều α.

Chúng ta có thể có được cùng kết quả như trong Hình 2.18(a) bằng cách viết lại các luật sinh cho A theo cách sau.

$$\begin{aligned} A &\rightarrow \beta R \\ R &\rightarrow \alpha R \mid \epsilon \end{aligned} \tag{2.11}$$

Ở đây R là một chưa tận mới. Luật sinh $R \rightarrow \alpha R$ là đệ qui phải bởi vì luật sinh cho R có R là ký hiệu cuối cùng ở vế phải. Các luật sinh đệ qui phải khiến các cây tăng trưởng xuống hướng sang phải như trong Hình 2.18(b). Các cây tăng trưởng xuống hướng về bên phải gây khó khăn cho việc dịch các biểu thức có chứa các toán tử kết hợp trái như toán tử trừ chẳng hạn. Tuy nhiên trong phần kế tiếp chúng ta sẽ thấy rằng bản dịch đúng đắn các biểu thức sang ký pháp hậu vị vẫn có thể thực hiện được bằng cách thiết kế cẩn thận lược đồ dịch dựa trên một văn phạm đệ qui phải.

Trong Chương 4, chúng ta sẽ xét các dạng tổng quát hơn của đệ qui trái và trình bày một phương pháp loại bỏ tất cả đệ qui trái ra khỏi một văn phạm.

2.5 MỘT CHƯƠNG TRÌNH DỊCH CHO CÁC BIỂU THỨC ĐƠN GIẢN

Sử dụng các kỹ thuật của ba phần vừa qua, chúng ta sẽ xây dựng một chương trình dịch dựa cú pháp dưới dạng một chương trình C, có khả năng dịch các biểu thức số học thành dạng hậu vị. Để bảo đảm cho chương trình khởi đầu đủ nhỏ để dễ quản lý, chúng ta bắt đầu với các biểu thức chứa các *ký số* (digit) được phân cách bởi các dấu cộng và trừ. Ngôn ngữ này sẽ được mở rộng trong hai phần tiếp theo để bao gồm cả các số, *định danh* (identifier) và những toán tử khác nữa. Bởi vì các biểu thức thường xuất hiện như những kết cấu trong rất nhiều ngôn ngữ nên rất có ích khi nghiên cứu chi tiết công việc phiên dịch chúng.

expr	→	*expr* + *term*	{ *print* ('+') }
expr	→	*expr* − *term*	{ *print* ('−') }
expr	→	*term*	
term	→	0	{ *print* ('0') }
term	→	1	{ *print* ('1') }
	. . .		
term	→	9	{ *print* ('9') }

Hình 2.19. Đặc tả khởi đầu cho chương trình dịch trung vị–hậu vị.

văn phạm tạo thuận lợi cho việc phân tích. Mặt khác chúng ta lại cần một văn phạm có thể dịch dễ dàng. Giải pháp rõ ràng là phải loại bỏ đệ qui trái. Tuy nhiên điều này cần phải được thực hiện cẩn thận như được minh họa qua thí dụ sau.

Thí dụ 2.9. Văn phạm dưới đây không thích hợp cho việc dịch các biểu thức sang dạng hậu vị, dù rằng nó cũng sinh ra chính ngôn ngữ của văn phạm trong Hình 2.19 và có thể sử dụng được cho phân tích cú pháp đệ qui-xuống.

$$
\begin{aligned}
\mathit{expr} &\rightarrow \mathit{term}\ \ \mathit{rest} \\
\mathit{rest} &\rightarrow +\ \mathit{expr}\ \mid\ -\ \mathit{expr}\ \mid\ \varepsilon \\
\mathit{term} &\rightarrow 0 \mid 1 \mid \ldots \mid 9
\end{aligned}
$$

Văn phạm này có vấn đề là toán hạng của các toán tử sinh ra bởi $\mathit{rest} \rightarrow +\ \mathit{expr}$ và $\mathit{rest} \rightarrow -\ \mathit{expr}$ không dễ nhận ra từ những luật sinh này. Không có chọn lựa nào bên dưới được chấp nhận để tạo ra bản dịch *rest.t* từ bản dịch *expr.t*:

$$\mathit{rest} \rightarrow -\ \mathit{expr} \quad \{\ \mathit{rest.t} := \text{'-'} \ \|\ \mathit{expr.t}\ \} \tag{2.12}$$

$$\mathit{rest} \rightarrow -\ \mathit{expr} \quad \{\ \mathit{rest.t} := \mathit{expr.t} \ \|\ \text{'-'}\ \} \tag{2.13}$$

(Chúng tôi chỉ trình bày luật sinh và hành động ngữ nghĩa cho toán tử trừ). Bản dịch của 9–5 là 95–. Tuy nhiên nếu chúng ta sử dụng hành động trong (2.12) thì dấu trừ xuất hiện trước *expr.t* và 9–5 được giữ nguyên là 9–5 trong bản dịch.

Mặt khác nếu chúng ta dùng (2.13) và dùng qui tắc tương tự cho dấu cộng, các toán tử đều di chuyển về bên phải và 9–5+2 được dịch không đúng thành 952+– (bản dịch đúng là 95–2+). □

Lắp ghép với lược đồ dịch

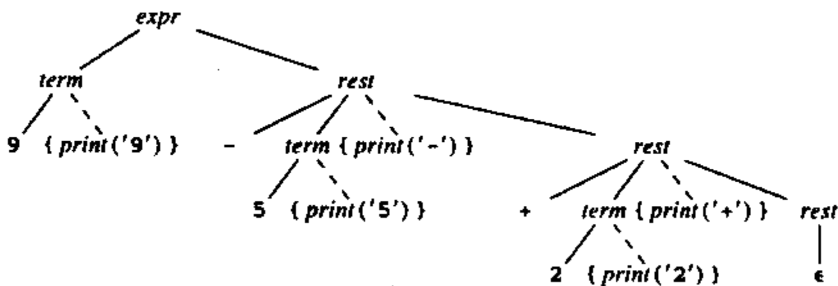
Kỹ thuật loại bỏ đệ qui trái được phác thảo trong Hình 2.18 cũng có thể áp dụng cho các luật sinh chứa hành động ngữ nghĩa. Chúng ta sẽ mở rộng phép biến đổi này trong Phần 5.5 và xem xét cả các *thuộc tính tổng hợp* (synthesized attribute). Kỹ thuật này biến đổi các luật sinh $A \rightarrow A\alpha \mid A\beta \mid \gamma$ thành

$$
\begin{aligned}
A &\rightarrow \gamma R \\
R &\rightarrow \alpha R \mid \beta R \mid \varepsilon
\end{aligned}
$$

Khi hành động ngữ nghĩa được gắn vào các luật sinh, chúng ta sẽ mang chúng vào kết quả biến đổi. Ở đây nếu đặt $A = \mathit{expr}$, $\alpha = +\ \mathit{term}\ \{\ \mathit{print}(\text{'+'})\ \}$, $\beta = -\ \mathit{term}\ \{\ \mathit{print}(\text{'-'})\ \}$, và $\gamma = \mathit{term}$, phép biến đổi ở trên sẽ sinh ra lược đồ dịch (2.14). Các luật sinh *expr* trong Hình 2.19 đã được biến đổi thành các luật sinh cho *expr* và một ký hiệu chưa tận mới là *rest* trong (2.14). Các luật sinh cho *term* được lập lại như trong Hình 2.19. Chú ý rằng văn phạm này khác với văn phạm trong Thí dụ 2.9 và sự khác biệt này cho phép thu được bản dịch như mong đợi.

$$\begin{array}{lcl} expr & \rightarrow & term\ rest \\ rest & \rightarrow & +\ term\ \{\ print('+')\ \}\ rest\ \mid\ -\ term\ \{\ print('-')\ \}\ rest\ \mid\ \varepsilon \\ term & \rightarrow & 0\ \{\ print('0')\ \} \\ term & \rightarrow & 1\ \{\ print('1')\ \} \\ & \dots & \\ term & \rightarrow & 9\ \{\ print('9')\ \} \end{array} \tag{2.14}$$

Hình 2.21 cho thấy quá trình dịch 9–5+2 nhờ văn phạm ở trên.



Hình 2.21. Dịch 9–5+2 thành 95–2+.

Thủ tục cho các chưa tận *expr*, *term* **và** *rest*

Bây giờ chúng ta cài đặt một chương trình dịch viết bằng C theo lược đồ dịch dựa cú pháp (2.14). Mấu chốt của chương trình dịch này là các đoạn mã C trong Hình 2.22 cho các hàm **expr**, **term** và **rest**. Những hàm này cài đặt các chưa tận tương ứng trong (2.14).

Hàm **match**, được trình bày sau, là một đối tác C của đoạn mã trong Hình 2.17 để đối sánh một thẻ từ với ký hiệu sải với và di chuyển đến ký tự kế tiếp qua chuỗi nguyên liệu. Bởi vì mỗi thẻ từ là một ký tự duy nhất trong ngôn ngữ của chúng ta, **match** có thể được cài đặt bằng cách so sánh và đọc các ký tự.

Đối với những bạn đọc chưa quen thuộc với ngôn ngữ lập trình C, chúng tôi thấy cần phải nêu rõ những khác biệt giữa C và các dẫn xuất của Algol như Pascal khi cần dùng những đặc trưng này của C. Một chương trình trong C gồm có một chuỗi các định nghĩa hàm, và việc thực hiện bắt đầu tại một hàm đặc biệt gọi là **main**. Các định nghĩa hàm không được lồng nhau. Các dấu ngoặc tròn bao quanh danh sách tham số bắt buộc phải có, ngay cả khi hàm không có tham số; vì thế chúng ta viết **expr()**, **term()** và **rest()**. Các hàm giao tiếp với nhau bằng cách truyền tham số "bằng giá trị" hoặc bằng các truy xuất dữ liệu có tầm vực toàn cục cho tất cả các hàm. Chẳng hạn hàm **term()** và **rest()** kiểm tra ký hiệu sải với bằng cách sử dụng định danh toàn cục **lookahead**.

```
expr()
{
    term(); rest();
}

rest()
{
    if (lookahead == '+') {
        match('+'); term(); putchar('+'); rest();
    }
    else if (lookahead == '-') {
        match('-'); term(); putchar('-'); rest();
    }
    else;
}

term();
{
    if (isdigit(lookahead)) {
        putchar(lookahead); match(lookahead);
    }
    else error();
}
```

Hình 2.22. Hàm cho các chưa tận *expr*, *rest*, và *term*.

C và Pascal sử dụng những ký hiệu sau đây cho *phép gán* (assignment) và các phép thử:

PHÉP TOÁN	C	PASCAL
phép gán	=	:=
phép thử đẳng thức	==	=
phép thử bất đẳng thức	!=	<>

Các hàm cho các chưa tận mô phỏng vế phải của các luật sinh. Thí dụ luật sinh $expr \rightarrow term\ rest$ được cài đặt bằng các lời gọi `term()` và `rest()` trong hàm `expr()`.

Một thí dụ khác, hàm `rest()` dùng luật sinh đầu tiên cho *rest* trong (2.14) nếu sải với là dấu cộng, dùng luật sinh thứ hai nếu sải với là dấu trừ còn luật sinh $rest \rightarrow \varepsilon$ làm mặc định. Luật sinh đầu tiên cho *rest* được cài đặt bằng câu lệnh **if** đầu tiên trong Hình 2.22. Nếu ký hiệu sải với là +, thì dấu cộng được đối sánh bằng lời gọi

match('+'). Sau lời gọi **term()**, thủ tục trong thư viện chuẩn của C là **putchar('+')** cài đặt hành động ngữ nghĩa bằng cách in ra ký tự +. Vì qui tắc thứ ba cho *rest* có ε ở vế phải nên mệnh đề **else** cuối cùng trong **rest()** không làm gì cả.

Mười luật sinh cho *term* sinh ra mười ký số. Trong Hình 2.22, thủ tục **isdigit** kiểm tra xem ký hiệu sải với có phải là một ký số hay không. Ký số sẽ được in và được đối sánh nếu phép thử thành công; ngược lại sẽ xảy ra lỗi. (Chú ý rằng **match** làm thay đổi ký hiệu sải với, vì thế hành động in phải xảy ra trước khi ký số được đối sánh.) Trước khi trình bày chương trình hoàn chỉnh, chúng ta sẽ sửa đổi đoạn mã trong Hình 2.22 nhằm cải thiện tốc độ hoạt động.

Tối ưu hóa chương trình dịch

Một số lời gọi đệ qui có thể được thay bằng các vòng lặp. Khi câu lệnh cuối cùng được thực hiện trong phần thân thủ tục là một lời gọi đệ qui của chính nó, lời gọi này được gọi là *đệ qui đuôi* (tail recursive). Thí dụ các lời gọi **rest()** tại mỗi dòng thứ tư và thứ bảy của hàm **rest()** là đệ qui đuôi bởi vì quyền điều khiển được trao vào cuối phần thân hàm sau mỗi lời gọi này.

Chúng ta có thể làm cho chương trình chạy nhanh hơn bằng cách thay lời gọi đệ qui đuôi bằng vòng lặp. Đối với các thủ tục không tham số, lời gọi đệ qui đuôi chỉ đơn giản được thay bằng một lệnh nhảy đến vị trí đầu tiên của thủ tục. Đoạn mã cho **rest** có thể được viết lại như sau:

```
rest()
{
L:    if (lookahead == '+') {
         match('+'); term(); putchar('+'); goto L;
      }
      else if (lookahead == '-' ) {
         match('-'); term(); putchar('-'); goto L;
      }
      else;
}
```

Với điều kiện ký hiệu sải với là dấu cộng hoặc dấu trừ, thủ tục **rest** sẽ đối sánh với dấu này, gọi **term** để in và đối sánh một ký số rồi lặp lại quá trình này. Chú ý rằng vì **match** loại bỏ dấu mỗi khi nó được gọi, chu trình này chỉ xảy ra trên một chuỗi dấu và ký số xen kẽ nhau. Nếu thay đổi này được thực hiện trong Hình 2.22, lời gọi còn lại duy nhất của **rest** là từ **expr** (xem dòng 3). Vì thế hai hàm này có thể được tích hợp lại thành một như trong Hình 2.23. Trong C, một câu lệnh *stmt* có thể được thực hiện

lập đi lập lại bằng cách viết

`while(1)` *stmt*

do 1 là điều kiện hằng đúng. Chúng ta có thể thoát ra khỏi vòng lặp nhờ lệnh **break**. Kiểu chương trình trong Hình 2.23 cho phép đưa thêm các toán tử khác vào một cách dễ dàng.

```
expr()
{
  term();
  while(1)
    if (lookahead == '+') {
      match('+'); term(); putchar('+');
    }
    else if (lookahead == '-') {
      match('-'); term(); putchar('-');
    }
    else break;
}
```

Hình 2.23. Thay thế các hàm **expr** và **rest** của Hình 2.22.

Chương trình hoàn chỉnh

Chương trình nguồn C hoàn chỉnh cho chương trình dịch của chúng ta được trình bày trong Hình 2.24. dòng đầu tiên, bắt đầu là **#include**, tải tập tin **<ctype.h>** vào, đó là một tập tin các thủ tục chuẩn có chứa đoạn mã cho vị từ **isdigit**.

Các thẻ từ, chứa các ký tự, được cung cấp bởi thủ tục thư viện chuẩn **getchar** sẽ đọc ký tự kế tiếp từ tập tin nguyên liệu. Tuy nhiên **lookahead** được khai báo là một số nguyên trên dòng 2 của Hình 2.24 để dự phòng cho các thẻ từ bổ sung không phải là những ký tự đơn sẽ được giới thiệu trong những phần sau. Bởi vì **lookahead** được khai báo bên ngoài tất cả mọi hàm, nó có tầm vực toàn cục đối với mọi hàm được định nghĩa sau dòng 2 của Hình 2.24.

Hàm **match** kiểm tra các thẻ từ; đọc thẻ từ kế tiếp nếu ký hiệu sãi với đối sánh được và gọi thủ tục **error** trong trường hợp ngược lại.

Hàm **error** sử dụng hàm thư viện chuẩn **printf** để in ra thông báo **"syntax error"** (lỗi cú pháp) rồi kết thúc chương trình bằng lời gọi **exit(1)** đến một hàm thư viện chuẩn khác.

```
#include <ctype.h> /* tải tập tin có vị từ isdigit vào */
int lookahead;

main()
{
  lookahead = getchar();
  expr(); putchar('\n'); /* thêm vào ký tự xuống hàng */
}

expr()
{
  term();
  while(1)
    if (lookahead == '+') {
      match('+'); term(); putchar('+');
    }
    else if (lookahead == '-') {
      match('-'); term(); putchar('-');
    }
    else break;
}

term()
{
  if (isdigit(lookahead)) {
    putchar(lookahead); match(lookahead);
  }
  else error();
}

match(int t)
{
  if (lookahead == t)
    lookahead = getchar();
  else error();
}

error()
{
  print("syntax error\n"); /* in ra thông báo lỗi */
  exit(1);                 /* rồi kết thúc */
}
```

Hình 2.24. Chương trình C dịch biểu thức trung vị thành dạng hậu vị.

2.6 PHÂN TÍCH TỪ VỰNG

Bây giờ chúng ta sẽ đưa thêm vào chương trình dịch của phần trước một *thể phân từ vựng* (lexical analyzer) để đọc và biến đổi nguyên liệu thành một chuỗi các *thẻ từ* (token) cho thể phân cú pháp xử lý. Từ định nghĩa văn phạm trong Phần 2.2, độc giả cần nhớ rằng câu của một ngôn ngữ gồm các chuỗi thẻ từ. Một chuỗi ký tự nguyên liệu tạo ra một thẻ từ duy nhất được gọi là một *từ tố* (lexeme). Thể phân từ vựng có thể lo xử lý các dạng biểu diễn từ tố của các thẻ từ thay cho thể phân cú pháp. Chúng ta sẽ liệt kê ra một số chức năng cần được thể phân từ vựng thực hiện.

Loại bỏ khoảng trắng và các dòng giải thích

Chương trình dịch biểu thức trong phần trước sẽ xét mỗi ký tự trong nguyên liệu, vì thế những ký tự "ngoài dự kiến" như các *ký tự trống* (blank) sẽ khiến nó thất bại. Nhiều ngôn ngữ cho phép các "khoảng trắng" (các ký tự trống, ký tự tab, ký tự newline) được xuất hiện giữa các thẻ từ. *Các dòng giải thích* (comment) cũng được thể phân cú pháp và chương trình dịch bỏ qua, vì thế chúng cũng có thể được xử lý như những khoảng trắng.

Nếu khoảng trắng được loại bỏ nhờ thể phân từ vựng, thể phân cú pháp sẽ không bao giờ phải xem xét chúng. Chọn lựa cách sửa đổi văn phạm để đưa cả khoảng trắng vào trong cú pháp thì hầu như rất khó cài đặt.

Các hằng

Mỗi khi có một ký số đơn độc xuất hiện trong một biểu thức, có lẽ sẽ hợp lý hơn khi cho phép đặt một hằng số nguyên nào đó ở vị trí của nó. Bởi vì một hằng nguyên là một dãy ký số, nó có thể dùng được bằng cách thêm các luật sinh vào văn phạm cho các biểu thức hoặc bằng cách tạo ra một thẻ từ cho các hằng như thế. Công việc gom các ký số thành các số nguyên nói chung được trao cho thể phân từ vựng bởi vì các số có thể được xử lý như những đơn vị riêng biệt trong quá trình dịch.

Gọi **num** là thẻ từ biểu thị cho một số nguyên. Khi một dãy ký số xuất hiện trong dòng nguyên liệu, thể phân từ vựng sẽ chuyển **num** cho thể phân cú pháp. Giá trị của số nguyên sẽ được chuyển theo dưới dạng một thuộc tính của thẻ từ **num**. Về mặt logic, thể phân từ vựng sẽ chuyển cả thẻ từ và thuộc tính cho thể phân cú pháp. Nếu chúng ta viết một thẻ từ và thuộc tính của nó như một bộ dữ liệu được bao giữa hai dấu < > thì nguyên liệu

```
31 + 28 + 59
```

được biến đổi thành một dãy các bộ

<**num**, 31> <+, > <**num**, 28> <+, > <**num**, 59>

Thẻ từ + không có thuộc tính. Thành phần thứ hai của các bộ, đó là các thuộc tính, không có vai trò gì trong khi phân tích cú pháp nhưng sẽ cần dùng đến trong khi dịch.

Nhận diện các định danh và từ khóa

Ngôn ngữ sử dụng các *định danh* (identifier) làm tên cho các biến, mảng, hàm và những thành phần tương tự. Một văn phạm cho một ngôn ngữ thường xử lý định danh như một thẻ từ. Thể phân cú pháp dựa trên một văn phạm như thế muốn nhận được cùng một thẻ từ, chẳng hạn là **id**, mỗi khi có một định danh xuất hiện trong nguyên liệu. Thí dụ với nguyên liệu

```
count = count + increment;                                    (2.15)
```

sẽ được thể phân từ vựng biến đổi thành dòng thẻ từ

id = id + id ; (2.16)

Khi đề cập đến việc phân tích từ vựng cho dòng nguyên liệu (2.15), chúng ta thường phân biệt giữa thẻ từ **id** với các từ tố `count` và `increment` đi kèm với những thể hiện của thẻ từ này. Chương trình dịch cần biết rằng từ tố `count` tạo ra hai thể hiện đầu tiên của **id** trong (2.16) còn từ tố `increment` tạo ra thể hiện thứ ba của **id**.

Khi gặp một từ tố tạo ra một định danh trong nguyên liệu, chúng ta cần có một cơ chế nhằm xác định xem từ tố này đã được gặp trước đó hay chưa. Như đã được đề cập trong Chương 1, chúng ta sẽ dùng một *bảng ký hiệu* (symbol table) cho công việc này. Từ tố được lưu trong bảng ký hiệu và một con trỏ chỉ đến mục ghi trong bảng trở thành một thuộc tính của thẻ từ **id**.

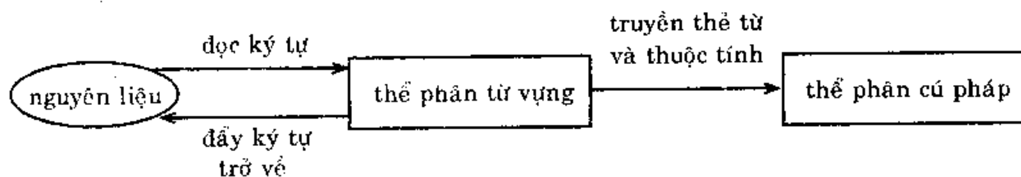
Nhiều ngôn ngữ sử dụng các chuỗi ký tự cố định như **begin**, **end**, **if**, vân vân, làm dấu chấm câu hoặc để xác định một số kết cấu. Những chuỗi ký tự này, được gọi là *từ khóa* (keyword), nói chung đều thỏa các qui tắc tạo ra định danh, vì thế chúng ta cũng cần một cơ chế để quyết định xem khi nào một từ tố tạo ra một từ khóa và khi nào nó tạo ra một định danh. Vấn đề sẽ dễ giải quyết hơn nếu các từ khóa được *dành riêng* (reserved), nghĩa là nếu chúng không được dùng làm định danh. Thế thì một chuỗi ký tự tạo ra một định danh chỉ nếu nó không phải là một từ khóa.

Vấn đề cô lập các thẻ từ cũng cần đặt ra nếu cùng một chuỗi ký tự xuất hiện trong từ tố của nhiều thẻ từ, chẳng hạn như <, <=, và <> của ngôn ngữ Pascal. Những kỹ thuật nhận diện hiệu quả những thẻ từ như thế sẽ được thảo luận trong Chương 3.

Giao diện cho thể phân từ vựng

Khi một thể phân từ vựng được đặt vào giữa thể phân cú pháp và dòng nguyên liệu, nó tương tác với cả hai phần này theo như Hình 2.25. Nó đọc các ký tự từ nguyên liệu, nhóm chúng lại thành các từ tố rồi gửi các thẻ từ được tạo bởi các từ tố cùng với giá trị thuộc tính của chúng đến những giai đoạn sau của trình biên dịch. Trong một số tình

huống, thể phân cú pháp phải đọc trước một số ký tự trước khi có thể quyết định sẽ trả về thẻ từ nào cho thể phân cú pháp. Thí dụ một thể phân cú pháp cho Pascal phải đọc tiếp nữa khi nó gặp ký tự >. Nếu ký tự kế tiếp là = thì chuỗi ký tự >= là từ tố tạo ra thẻ từ cho toán tử "lớn hơn hoặc bằng". Ngược lại thì > là từ tố tạo ra toán tử "lớn hơn", và thể phân từ vựng đã đọc một ký tự quá nhiều lần. Ký tự "dư" ra này phải được đưa trở lại nguyên liệu bởi vì nó có thể là ký tự đầu tiên của từ tố kế tiếp trong nguyên liệu.



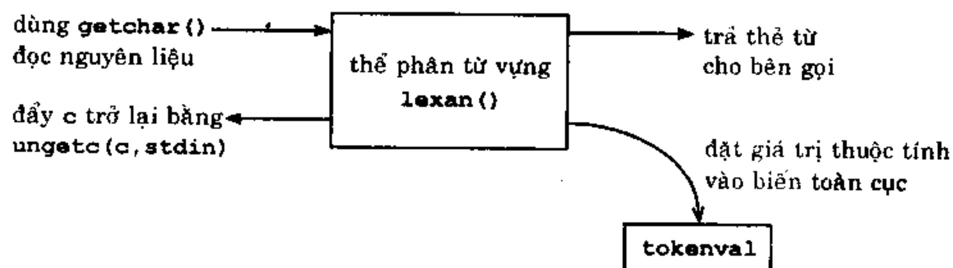
Hình 2.25. Đặt thể phân từ vựng vào giữa nguyên liệu và thể phân cú pháp.

Thể phân từ vựng và thể phân cú pháp tạo ra một cặp "*kẻ làm–người ăn*" (producer-consumer). Thể phân từ vựng tạo ra các thẻ từ và thể phân cú pháp sẽ "tiêu thụ" nó. Các thẻ từ được sinh ra có thể được giữ trong một vùng đệm cho đến khi chúng được sử dụng. Tương tác giữa hai thể phân tích này chỉ bị ràng buộc bởi kích thước vùng đệm bởi vì thể phân từ vựng sẽ không thể tiến hành khi vùng đệm đầy còn thể phân cú pháp sẽ không thể tiến khi hành vùng đệm rỗng. Thông thường vùng đệm chỉ giữ một thẻ từ. Trong trường hợp này, tương tác có thể được cài đặt đơn giản bằng cách đưa thể phân từ vựng thành một thủ tục được thể phân cú pháp gọi, trả về các thẻ từ theo như yêu cầu.

Cài đặt thao tác đọc và đẩy các ký tự trở lại thường được thực hiện bằng cách thiết lập một vùng đệm nguyên liệu. Mỗi lần sẽ đọc một khối các ký tự vào vùng đệm; sử dụng một con trỏ để theo dõi phần nguyên liệu đã được phân tích. Đẩy một ký tự trở lại được cài đặt bằng cách cho con trỏ trở lui lại. Các ký tự nguyên liệu cũng có thể cần được lưu lại cho công việc ghi nhận lỗi bởi vì cần phải chỉ ra vị trí lỗi trong đoạn chương trình. Đệm các ký tự nguyên liệu chỉ vì lý do hiệu năng. Chuyển một khối ký tự bao giờ cũng hiệu quả hơn là chuyển mỗi lần một ký tự. Các kỹ thuật đệm nguyên liệu sẽ được thảo luận trong Phần 3.2.

Một thể phân từ vựng

Bây giờ chúng ta xây dựng một thể phân từ vựng thông thường cho chương trình dịch biểu thức của Phần 2.5. Mục đích của thể phân từ vựng là cho phép các khoảng trắng và các số xuất hiện trong biểu thức. Trong phần kế tiếp chúng ta mở rộng thể phân từ vựng cho phép dùng cả các định danh.



Hình 2.26. Cài đặt các tương tác trong Hình 2.25.

Hình 2.26 gợi ý một cách cài đặt các tương tác trong Hình 2.25 của thể phân từ vựng, được viết bằng C dưới dạng hàm **lexan**. Các thủ tục **getchar** và **ungetc**, được lấy từ thư viện chuẩn **<stdio.h>** lo liệu việc đệm nguyên liệu; **lexan** đọc và đẩy các ký tự nguyên liệu trở lại bằng cách gọi thủ tục **getchar** và **ungetc**. Với **c** được khai báo là một ký tự, cặp câu lệnh

```
c = getchar(); ungetc(c, stdin);
```

để lại dòng nguyên liệu như cũ. Lời gọi **getchar** gán ký tự tiếp theo cho **c**; lời gọi **ungetc** đẩy giá trị của **c** vào lại dòng nguyên liệu chuẩn **stdin**.

Nếu ngôn ngữ cài đặt không cho phép trả về các cấu trúc dữ liệu từ các hàm thì thẻ từ và thuộc tính của nó phải được truyền riêng rẽ. Hàm **lexan** trả về một số nguyên mã hóa cho một thẻ từ. Thẻ từ cho một ký tự có thể là một số nguyên qui ước được dùng để mã hóa cho ký tự đó. Một thẻ từ như **num** có thể được mã hóa bằng một số nguyên lớn hơn mọi số nguyên được dùng để mã hóa cho các ký tự, chẳng hạn là 256. Để dễ dàng thay đổi cách mã hóa, chúng ta dùng một hằng tượng trưng NUM thay cho số nguyên mã hóa của **num**. Trong Pascal, chúng ta có thể dùng khai báo **const** để liên kết NUM với số nguyên mã hóa cho **num**; trong C, NUM có thể dùng thay cho 256 bằng câu lệnh **define**:

```
#define NUM 256
```

Hàm **lexan** trả về NUM khi một dãy ký số được phát hiện trong nguyên liệu. Biến toàn cục **tokenval** được đặt là giá trị của dãy ký số này. Vì thế nếu một ký số 7 và một ký số 6 nằm kế tiếp nhau trong nguyên liệu, **tokenval** được gán giá trị 76.

Cho phép các số có mặt trong biểu thức đòi hỏi phải thay đổi văn phạm trong Hình 2.19. Chúng ta thay các ký số riêng rẽ bằng chưa tận *factor* và đưa ra các luật sinh và hành động ngữ nghĩa như sau:

$$
\begin{array}{lll}
\textit{factor} & \rightarrow & (\ \textit{expr}\ ) \\
 & | & \textbf{num}\ \ \{\ \textit{print}(\textbf{num}.\textit{value})\ \}
\end{array}
$$

```
factor()
  {
    if (lookahead == '(') {
      match('('); expr(); match(')');
    }
    else if (lookahead == NUM) {
      printf(" %d ", tokenval); match(NUM);
    }
    else error();
  }
```

Hình 2.27. Đoạn chương trình C cho *factor* khi các toán hạng có thể là các số.

Đoạn chương trình C cho *factor* trong Hình 2.27 là một cài đặt trực tiếp của các luật sinh ở trên. Khi `lookahead` bằng với `NUM`, giá trị của thuộc tính **num**.*value* được cho bởi biến toàn cục `tokenval`. Hành động in ra giá trị này được thực hiện bởi hàm thư viện `printf`. Đối thứ nhất của `printf` là chuỗi nằm giữa các dấu nháy kép mô tả khuôn dạng được dùng khi in các đối còn lại. Khi `%d` xuất hiện trong chuỗi này, đối kế tiếp sẽ được in ra dưới dạng số thập phân. Vì thế câu lệnh `printf` trong Hình 2.27 in ra một khoảng trống theo sau là dạng thập phân của `tokenval` rồi một khoảng trống nữa sau đó.

Cài đặt của hàm `lexan` được trình bày trong Hình 2.28. Mỗi lần thân vòng `while` trên các dòng 8-28 được thực hiện, một ký tự được đọc vào `t` trên dòng 9. Nếu là ký tự blank hoặc tab (được viết là `'\t'`), thì không có thẻ từ nào được trả về cho thể phân cú pháp; chúng ta chỉ đi qua vòng `while` rồi quay lại. Nếu là ký tự newline (được ghi là `'\n'`) thì biến toàn cục `lineno` được tăng lên 1, vì thế theo dõi được chỉ số dòng trong nguyên liệu nhưng cũng không trả về một thẻ từ nào cả. Cung cấp chỉ số dòng trong một thông báo lỗi sẽ giúp định vị được lỗi.

Đoạn chương trình đọc một dãy ký số nằm trên các dòng 14-23. Vị từ `isdigit(t)` từ tập tin `<ctype.h>` được dùng trên các dòng 14 và 17 để xác định xem một ký tự kế tiếp `t` có phải là ký số hay không. Nếu đúng như thế thì giá trị nguyên của nó được tính từ biểu thức `t-'0'` trong cả hai bảng mã ASCII và EBCDIC. Với các bảng mã khác, việc chuyển đổi có thể phải dùng một cách khác. Trong Phần 2.9 chúng ta sẽ gắn thể phân từ vựng này vào chương trình dịch biểu thức của chúng ta.

2.7 KẾT HỢP VỚI BẢNG KÝ HIỆU

Một cấu trúc dữ liệu được gọi là *bảng ký hiệu* (symbol table) thường được dùng để lưu thông tin về nhiều *kết cấu* (construct) của ngôn ngữ nguồn. Các thông tin này được thu thập trong giai đoạn phân tích của trình biên dịch và được dùng trong giai đoạn tổng

hợp để sinh ra mã đích. Thí dụ trong quá trình phân tích từ vựng, các chuỗi ký tự (từ tố) tạo ra một định danh sẽ được lưu vào một mục ghi trong bảng ký hiệu. Các giai đoạn sau có thể bổ sung thêm các thông tin về kiểu của định danh, cách sử dụng nó (thí dụ do thủ tục, biến hoặc nhãn), và vị trí được lưu. Giai đoạn sinh mã sẽ dùng thông tin này để tạo ra mã phù hợp, cho phép lưu và truy xuất biến đó. Trong Phần 7.6 (Tập II) chúng ta sẽ thảo luận chi tiết về cách cài đặt và sử dụng bảng ký hiệu. Ở phần này chúng ta chỉ minh họa cách tương tác giữa thể phân từ vựng với một bảng ký hiệu.

```
(1)  #include <stdio.h>
(2)  #include <ctype.h>
(3)  int lineno = 1;
(4)  int tokenval = NONE;
(5)  int lexan()
(6)  {
(7)     int t;
(8)     while(1) {
(9)        t = getchar();
(10)       if (t == ' ' || t == '\t')
(11)          ; /*  loại bỏ blank và tab */
(12)       else if (t == '\n')
(13)          lineno = lineno + 1;
(14)       else if (isdigit(t)) {
(15)          tokenval = t - '0';
(16)          t = getchar();
(17)          while (isdigit(t)) {
(18)             tokenval = tokenval*10 + t-'0';
(19)             t = getchar();
(20)          }
(21)          ungetc(t, stdin);
(22)          return NUM;
(23)       }
(24)       else {
(25)          tokenval = NONE;
(26)          return t;
(27)       }
(28)    }
(29) }
```

Hình 2.28. Thể phân từ vựng được viết bằng C để loại bỏ khoảng trắng và gom các số.

Giao diện của bảng ký hiệu

Các thủ tục của bảng ký hiệu chủ yếu liên quan đến việc lưu và truy xuất các từ tố. Khi một từ tố được lưu, chúng ta cũng lưu thẻ từ đi kèm với từ tố đó. Các thao tác sau đây sẽ được thực hiện trên bảng ký hiệu.

`insert(s, t)`:	trả về chỉ mục của một mục ghi mới cho chuỗi `s`, thẻ từ `t`.
`lookup(s)`:	trả về chỉ mục của mục ghi dành cho chuỗi `s`, hoặc là 0 nếu không tìm thấy `s`.

Thể phân từ vựng dùng thao tác `lookup` (tìm kiếm) để xác định xem đã có một mục ghi dành cho một từ tố trong bảng ký hiệu hay chưa. Nếu mục đó chưa có thì nó dùng thao tác `insert` để tạo ra. Chúng ta sẽ thảo luận một cài đặt mà trong đó cả thể phân từ vựng và thể phân cú pháp đều biết về dạng thức của các mục ghi trong bảng ký hiệu.

Xử lý các từ khóa dành riêng

Các thủ tục ở trên có thể xử lý được các *từ khóa dành riêng* (reserved keyword). Thí dụ xét các thẻ từ **div** và **mod** với các từ tố `div` và `mod`. Chúng ta có thể khởi gán bảng ký hiệu bằng cách dùng lời gọi

```
insert("div", div);
insert("mod", mod);
```

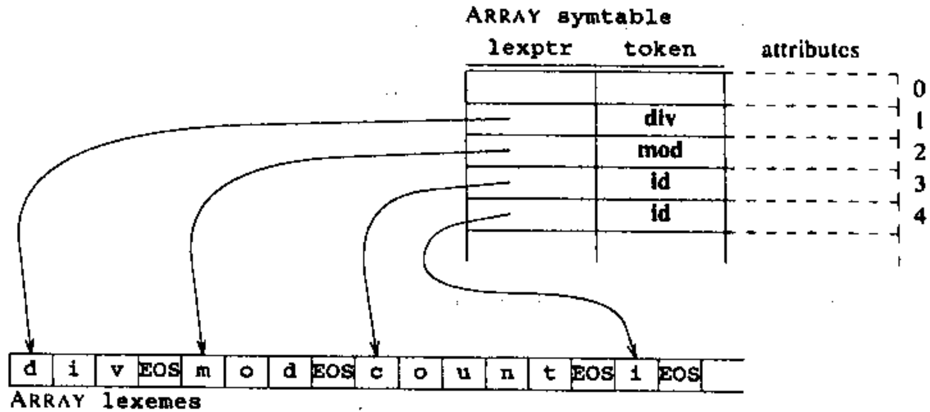
Mọi lời gọi `lookup("div")` sau đó sẽ trả về thẻ từ **div**, vì thế `div` không thể được dùng làm định danh.

Một tập các từ khóa dành riêng có thể được xử lý theo lối này qua việc khởi gán bảng ký hiệu cho phù hợp.

Một cài đặt cho bảng ký hiệu

Cấu trúc dữ liệu cài đặt cụ thể cho một bảng ký hiệu được phác thảo trong Hình 2.29. Chúng ta không muốn dành ra một lượng không gian nhất định để lưu các từ tố tạo ra định danh; một lượng không gian cố định có thể không đủ lớn để lưu các định danh rất dài và có thể làm lãng phí nhiều khi gặp một định danh ngắn, chẳng hạn như `i`. Trong Hình 2.29, chúng ta dùng một mảng riêng rẽ là `lexemes` để lưu chuỗi ký tự tạo ra một định danh. Chuỗi này kết thúc bằng một ký tự `EOS` (end-of-string, cuối chuỗi) không xuất hiện trong các định danh. Mỗi mục ghi trong mảng `symtable` cho bảng ký hiệu là một mẫu tin gồm có hai trường, trường `lexptr` chỉ đến đầu từ tố và trường `token`. Cũng có thể dùng thêm một số trường khác để lưu các giá trị thuộc tính mặc dù ở đây chúng ta không làm như thế.

Trong Hình 2.29, mục ghi thứ zero được để trống bởi vì `lookup` trả về `0` để chỉ ra rằng không có mục ghi nào dành cho chuỗi đang xét. Các mục ghi thứ nhất và thứ hai dành cho các từ khóa `div` và `mod`. Mục ghi thứ ba và thứ tư dành cho các định danh `count` và `i`.



Hình 2.29. Bảng ký hiệu và mảng để lưu các chuỗi.

Đoạn mã giả cho thể phân từ vựng được dùng để xử lý các định danh được trình bày trong Hình 2.30; một cài đặt bằng C xuất hiện trong Phần 2.9. Khoảng trắng và hằng số nguyên được xử lý bởi thể phân từ vựng bằng phương thức giống như trong Hình 2.28 của phần trước.

Khi thể phân từ vựng hiện tại của chúng ta đọc một chữ cái, nó bắt đầu lưu các chữ cái và ký số trong vùng đệm `lexbuf`. Chuỗi được thu thập vào `lexbuf` sau đó sẽ được tìm trong bảng ký hiệu bằng thao tác `lookup`. Bởi vì bảng ký hiệu được khởi gán với các mục ghi cho từ khóa `div` và `mod` như trong Hình 2.29, thao tác `lookup` sẽ thấy những mục ghi này nếu `lexbuf` chứa `div` hoặc `mod`. Nếu không có mục ghi nào cho chuỗi đang trong `lexbuf`, nghĩa là `lookup` trả về `0`, thì `lexbuf` chứa một từ tố của một định danh mới. Một mục ghi cho định danh mới sẽ được tạo ra bằng thao tác `insert`. Sau khi chèn, `p` là chỉ mục của mục ghi trong bảng ký hiệu cho chuỗi đang trong `lexbuf`. Chỉ mục này dùng để tương tác với thể phân cú pháp bằng cách đặt `tokenval` là `p`, và thẻ từ trong trường `token` của mục ghi này được trả về.

Hành động mặc nhiên là trả về số nguyên mã hóa cho ký tự làm thẻ từ. Bởi vì ở đây các thẻ từ một ký tự không có thuộc tính, `tokenval` được đặt là NONE.

```
function lexan: integer;
var  lexbuf:  array[0..100] of char;
     c:       char;
begin
   loop begin
      đọc một ký tự vào c;
      if c là một ký tự trống blank hoặc một ký tự tab then
         không thực hiện gì
      else if c là một ký tự newline then
         lineno := lineno + 1
      else if c là một ký số then begin
         đặt tokenval là giá trị của ký số này và các ký số theo sau;
         return NUM
      end
      else if c là một chữ cái then begin
         đặt c và các ký tự, ký số theo sau vào lexbuf;
         p := lookup(lexbuf);
         if p = 0 then
            p := insert(lexbuf, ID);
         tokenval := p;
         return trường token của mục ghi p
      end
      else begin  /* thẻ từ là một ký tự  */
         đặt tokenval là NONE;   /* không có thuộc tính */
         return số nguyên mã hóa của ký tự c
      end
   end
end
```

Hình 2.30. Đoạn mã giả cho một thể phân từ vựng.

2.8 MÁY CHỒNG XẾP TRỪU TƯỢNG

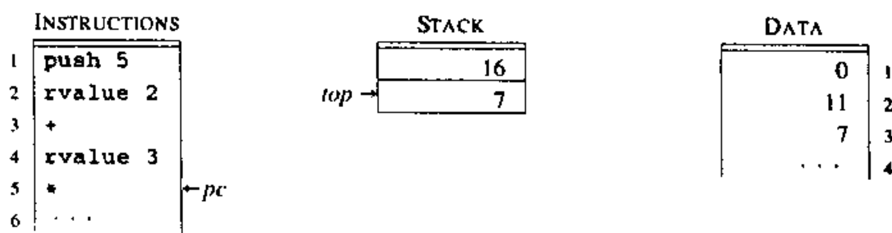
Kỳ đầu (front end) của một trình biên dịch xây dựng một dạng biểu diễn trung gian cho chương trình nguồn để từ đó *kỳ sau* (back end) sẽ tạo ra chương trình đích. Một dạng biểu diễn trung gian thông dụng là dạng mã cho một *máy chồng xếp trừu tượng* (abstract stack machine). Như đã nói ở Chương 1, phân chia một trình biên dịch thành kỳ đầu và kỳ sau cho phép dễ dàng sửa lại trình biên dịch để chạy trên một máy khác.

Trong phần này chúng ta sẽ trình bày một máy chồng xếp trừu tượng và chỉ ra

cách sinh mã chương trình cho nó. Máy này có các bộ nhớ dành riêng cho các chỉ thị và dành riêng cho dữ liệu; tất cả các phép toán số học được thực hiện trên các giá trị nằm trên một chồng xếp. Các chỉ thị rất hạn chế và được chia làm ba nhóm: nhóm chỉ thị số học trên số nguyên, nhóm thao tác chồng xếp và nhóm dòng điều khiển. Hình 2.31 minh họa cho máy này. Con trỏ *pc* chỉ ra chỉ thị đang được cho thực hiện. Ý nghĩa của các chỉ thị trên máy sẽ được thảo luận như dưới đây.

Chỉ thị số học

Máy trừu tượng phải cài đặt mỗi toán tử bằng ngôn ngữ trung gian. Một phép toán cơ bản như phép cộng hoặc trừ đều được máy trừu tượng hỗ trợ trực tiếp. Một phép toán phức tạp hơn có thể cần phải được cài đặt như một loạt chỉ thị của máy trừu tượng. Chúng ta đơn giản hóa phần mô tả máy bằng cách giả thiết rằng có một chỉ thị cho mỗi toán tử số học.



Hình 2.31. Hình ảnh của máy chồng xếp sau khi thực hiện bốn chỉ thị đầu tiên.

Mã chương trình máy trừu tượng cho một biểu thức số học sẽ mô phỏng hành động ước lượng dạng hậu vị cho biểu thức đó bằng phương pháp dùng chồng xếp. Việc ước lượng được tiến hành bằng cách xử lý dạng hậu vị từ trái sang phải, đẩy mỗi toán hạng vào chồng xếp khi bắt gặp nó. Khi gặp một toán tử *k*-ngôi, đối tận trái của nó nằm ở $(k - 1)$ vị trí bên dưới đỉnh chồng và đối tận phải nằm tại đỉnh. Hành động ước lượng áp dụng toán tử cho *k* giá trị trên cùng của chồng, lấy các toán hạng ra và đẩy kết quả vào lại chồng. Thí dụ khi ước lượng biểu thức `1 3 + 5 *`, các hành động sau đây được thực hiện.

1. Đẩy số 1 vào chồng xếp.
2. Đẩy số 3 vào chồng xếp.
3. Cộng hai phần tử trên cùng, lấy chúng ra khỏi chồng xếp rồi đẩy kết quả 4 vào.
4. Đẩy số 5 vào chồng xếp.
5. Nhân hai phần tử trên cùng, lấy chúng ra khỏi chồng xếp rồi đẩy kết quả 20 vào.

Giá trị ở đỉnh chồng xếp ở bước cuối cùng (ở đây là 20) là giá trị của toàn bộ biểu thức.

Trong ngôn ngữ trung gian, tất cả mọi giá trị đều là số nguyên, 0 tương ứng với **false** và các số nguyên khác không tương ứng với **true**. Toán tử logic **and** và **or** cần phải có cả hai đối.

L-giá trị và R-giá trị

Chúng ta cần phân biệt ý nghĩa của các định danh ở bên trái và bên phải của một phép gán. Trong mỗi phép gán sau

```
i := 5;
i := i + 1;
```

vế phải xác định một giá trị nguyên, còn vế trái xác định nơi giá trị được lưu. Tương tự nếu **p** và **q** là những con trỏ chỉ đến các ký tự và

```
p↑ := q↑;
```

vế phải **q↑** xác định một ký tự còn **p↑** xác định vị trí ký tự được lưu. Các thuật ngữ *l-giá trị* và *r-giá trị* muốn nói đến các giá trị thích hợp tương ứng ở vế trái và vế phải của một phép gán. Nghĩa là, *r-giá trị* là điều mà chúng ta thường xem là "giá trị" còn *l-giá trị* chính là các vị trí.

Thao tác chồng xếp

Bên cạnh những chỉ thị cho thao tác đẩy một hằng số nguyên vào chồng xếp và nhặt một giá trị ra khỏi đỉnh chồng, chúng ta còn có những chỉ thị truy xuất vùng nhớ dữ liệu:

push *v*	đẩy *v* vào chồng xếp
rvalue *l*	đẩy nội dung ở vị trí dữ liệu *l* vào chồng xếp
lvalue *l*	đẩy địa chỉ của vị trí dữ liệu *l* vào chồng xếp
pop	xóa bỏ giá trị ở đỉnh chồng xếp (nhặt ra khỏi chồng)
:=	*r*-giá trị trên đỉnh chồng được đặt vào *l*-giá trị bên dưới nó và cả hai đều được nhặt ra khỏi chồng
copy	đẩy một bản sao của giá trị ở đỉnh vào chồng

Dịch các biểu thức

Đoạn mã chương trình dùng để ước lượng một biểu thức trên một máy chồng xếp có liên quan mật thiết với ký pháp hậu vị cho biểu thức đó. Theo định nghĩa, dạng hậu vị của biểu thức $E + F$ là sự ghép nối dạng hậu vị của E, dạng hậu vị của F và dấu +. Tương tự đoạn mã máy chồng xếp để ước lượng $E + F$ là ghép nối của đoạn mã ước lượng E, đoạn mã ước lượng F và chỉ thị cộng các giá trị của chúng lại. Phiên dịch các biểu thức thành mã máy chồng xếp vì thế có thể được thực hiện bằng cách kết hợp các

chương trình dịch trong các Phần 2.6 và 2.7 lại.

Ở đây chúng ta tạo ra đoạn mã chương trình cho các biểu thức trong đó các vị trí dữ liệu đã được đánh địa chỉ bằng các ký hiệu. (Việc cấp phát vị trí dữ liệu cho các định danh sẽ được thảo luận trong Chương 7). Biểu thức `a+b` được dịch thành:

```
rvalue a
rvalue b
+
```

và đọc là: đẩy các nội dung ở các vị trí dữ liệu của `a` và `b` vào chồng xếp; rồi nhặt hai giá trị trên cùng của chồng xếp ra, cộng chúng lại rồi đẩy kết quả vào chồng xếp.

Dịch các phép gán thành mã máy chồng xếp được thực hiện như sau: *l*-giá trị của định danh cần gán sẽ được đẩy vào chồng, biểu thức được ước lượng và *r*-giá trị của nó được gán cho định danh. Thí dụ phép gán

```
day := (1461*y) div 4 + (153*m + 2) div 5 + d          (2.17)
```

được dịch thành đoạn mã trong Hình 2.32.

```
lvalue day          push 2
push 1461           +
rvalue y            push 5
*                   div
push 4              +
div                 rvalue d
push 153            +
rvalue m            :=
*
```

Hình 2.32. Dịch phép gán `day := (1461*y) div 4 + (153*m + 2) div 5 + d`.

Những ghi nhận này có thể được diễn tả một cách hình thức như sau. Mỗi chưa tận có một thuộc tính *t* cho biết bản dịch của nó. Thuộc tính *lexeme* của **id** cho biết dạng biểu diễn chuỗi của định danh.

stmt → **id** := *expr*
{ *stmt.t* := `'lvalue'` || **id**.*lexeme* || *expr.t* || `':='` }

Dòng điều khiển

Máy chồng xếp thực hiện các chỉ thị theo đúng thứ tự liệt kê trừ khi được yêu cầu thực hiện khác đi bằng câu lệnh nhảy có điều kiện hoặc không điều kiện. Có một số tùy chọn dùng để mô tả các đích nhảy:

1. Toán hạng làm chỉ thị cho biết vị trí đích.
2. Toán hạng làm chỉ thị mô tả khoảng cách tương đối cần nhảy theo chiều tới (dương) hoặc lui (âm).
3. Đích được mô tả bằng các ký hiệu tượng trưng; nghĩa là máy có cho phép dùng các nhãn.

Với hai tùy chọn đầu tiên, chúng ta có thêm khả năng lấy toán hạng từ đỉnh chồng xếp.

Chúng ta chọn tùy chọn thứ ba cho máy trừu tượng đang xét bởi vì nó rất dễ sinh ra các lệnh nhảy. Hơn nữa địa chỉ tượng trưng (dùng ký hiệu) không cần phải thay đổi nếu sau khi đã sinh ra mã cho máy trừu tượng, chúng ta cần phải sửa đổi lại đoạn mã, chèn thêm hay xóa bớt các chỉ thị.

Chỉ thị cho dòng điều khiển của máy chồng xếp là:

`label` *l*	đích của các lệnh nhảy đến *l*; không có tác dụng nào khác
`goto` *l*	chỉ thị tiếp theo được lấy từ câu lệnh có `label` *l*
`gofalse` *l*	lấy giá trị ở đỉnh ra; nhảy đến *l* nếu nó là zero
`gotrue` *l*	lấy giá trị ở đỉnh ra; nhảy đến *l* nếu nó khác zero
`halt`	ngừng thực hiện

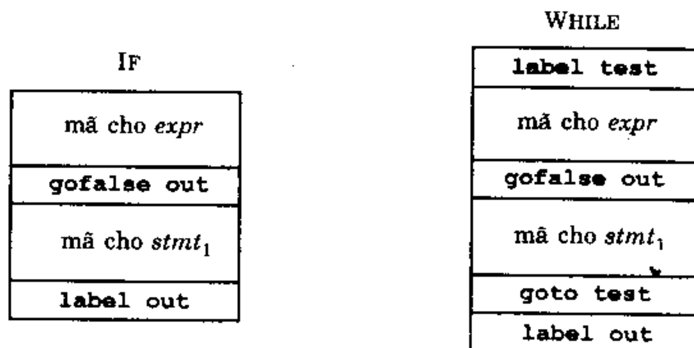
Dịch các câu lệnh

Sơ đồ bố trí trong Hình 2.33 phác thảo đoạn mã máy trừu tượng cho các câu lệnh **while** và câu lệnh điều kiện **if**. Thảo luận dưới đây tập trung vào việc tạo nhãn.

Xét sơ đồ đoạn mã cho câu lệnh **if** trong Hình 2.33. Chỉ có một chỉ thị `label out` trong bản dịch của chương trình nguồn; bằng không sẽ không biết dòng điều khiển sẽ chuyển đến đâu từ một câu lệnh `goto out`. Vì thế chúng ta cần một cơ chế nào đó để thay `out` trong sơ đồ bằng một nhãn duy nhất mỗi lần dịch một câu lệnh **if**.

Giả sử *newlabel* là một thủ tục trả về một nhãn mới mỗi khi nó được gọi. Trong hành động ngữ nghĩa sau đây, nhãn được trả về bởi một lời gọi đến *newlabel* được ghi lại bằng cách dùng một biến cục bộ *out*:

$$stmt \rightarrow \textbf{if}\ expr\ \textbf{then}\ stmt_1 \quad \begin{array}{l} \{\ out := newlable; \\ stmt.t := expr.t \ \| \\ \texttt{'gofalse'}\ out \ \| \\ \quad stmt_1.t \ \| \\ \quad \texttt{'label'}\ out \ \} \end{array} \tag{2.18}$$



Hình 2.33. Sơ đồ đoạn mã cho các câu lệnh điều kiện và câu lệnh **while**.

Đưa ra một bản dịch

Chương trình dịch biểu thức trong Phần 2.5 dùng các lệnh in để tạo dần dần bản dịch cho một biểu thức. Các lệnh in tương tự có thể được dùng để đưa ra bản dịch cho các câu lệnh. Thay vì các câu lệnh in, chúng ta dùng thủ tục *emit* nhằm che dấu các chi tiết in. Thí dụ *emit* có thể phải lo liệu xem mỗi chỉ thị máy trừu tượng có cần ở trên một hàng riêng biệt hay không. Sử dụng thủ tục *emit*, chúng ta có thể viết lại (2.18) như dưới đây

$$
\begin{array}{lll}
stmt & \to & \textbf{if} \\
 & & expr \quad \{\ out := newlabel;\ emit(\texttt{'gofalse'}, out);\ \} \\
 & & \textbf{then} \\
 & & stmt_1 \quad \{\ emit(\texttt{'label'}, out);\ \}
\end{array}
$$

Khi các hành động ngữ nghĩa xuất hiện bên trong một luật sinh, chúng ta xét các phần tử ở vế phải của luật sinh theo thứ tự từ trái sang phải. Đối với luật sinh ở trên, thứ tự các hành động như sau: các hành động trong khi phân tích cú pháp cho *expr* được thực hiện, *out* được đặt là nhãn do *newlabel* trả về và chỉ thị `gofalse` được đưa ra, các hành động khi phân tích $stmt_1$ được thực hiện, và cuối cùng chỉ thị `label` được đưa ra. Giả sử rằng các hành động khi phân tích cú pháp cho *expr* và $stmt_1$ đưa ra đoạn chương trình cho những chưa tận này, luật sinh ở trên cài đặt sơ đồ đoạn chương trình của Hình 2.33.

Đoạn mã giả dịch phép gán và các câu lệnh điều kiện được trình bày trong Hình 2.34. Bởi vì biến *out* là cục bộ đối với thủ tục *stmt*, giá trị của nó không bị ảnh hưởng bởi các lời gọi đến các thủ tục *expr* và *stmt*. Việc sinh ra các nhãn cần phải giải thích kỹ hơn. Giả sử rằng các nhãn trong bản dịch có dạng `L1`, `L2`, . . . Đoạn mã giả thao tác những nhãn này bằng cách dùng số nguyên theo sau `L`. Vì thế *out* được khai báo là số nguyên, *newlabel* trả về một số nguyên và nó trở thành giá trị của *out*, rồi *emit* phải được viết để in ra một nhãn khi cho biết một số nguyên.

```
procedure stmt;
    var test, out: integer; /* cho các nhãn */
begin
    if lookahead = id then begin
        emit('lvalue', tokenval); match(id); match(':='); expr
    end
    else if lookahead = 'if' then begin
        match('if');
        expr;
        out := newlabel;
        emit('gofalse', out);
        match('then');
        stmt;
        emit('label', out)
    end
        /* đoạn mã cho các lệnh còn lại để ở đây */
    else error;
end
```

Hình 2.34. Đoạn mã giả để dịch các câu lệnh.

Sơ đồ đoạn mã cho các câu lệnh **while** trong Hình 2.33 có thể được chuyển đổi thành chương trình theo cách thức tương tự. Bản dịch của một dãy lệnh chỉ là sự ghép nối các bản dịch trong dãy và được để lại cho độc giả.

Bản dịch của các kết cấu một-đầu-vào một-đầu-ra thì tương tự như của câu lệnh **while**. Chúng ta minh họa bằng cách xem xét dòng điều khiển trong các biểu thức.

Thí dụ 2.10. Thể phân từ vựng trong Phần 2.7 chứa một câu lệnh điều kiện dạng:

if t = **blank** **or** t = **tab** **then** . . .

Nếu t là một ký tự trống thì rõ ràng không cần phải thử xem t có phải là một ký tự tab nữa bởi vì đẳng thức thứ nhất đã khẳng định điều kiện là đúng. Biểu thức

$expr_1$ **or** $expr_2$

vì thế có thể được cài đặt là

if $expr_1$ **then true else** $expr_2$

Độc giả có thể xác nhận lại rằng đoạn mã sau đây cài đặt toán tử **or**:

```
đoạn mã cho expr1
copy                    /* chép lại giá trị của expr1 */
gotrue out
pop                     /* lấy giá trị của expr1 ra */
đoạn mã cho expr2
label out
```

Cần nhớ rằng các chỉ thị **gotrue** và **gofalse** lấy giá trị ở đỉnh chồng xếp nhằm đơn giản hóa việc sinh mã cho các câu lệnh điều kiện và **while**. Bằng cách chép lại giá trị của $expr_1$, chúng ta bảo đảm rằng giá trị trên đỉnh chồng sẽ đúng (true) nếu chỉ thị **gotrue** dẫn đến một lệnh nhảy. □

2.9 KẾT NỐI CÁC KỸ THUẬT

Trong chương này chúng ta đã trình bày một số kỹ thuật dựa cú pháp để xây dựng kỳ đầu của trình biên dịch. Để tóm tắt lại, trong phần này chúng ta sẽ kết nối chúng vào một chương trình C có chức năng như một chương trình dịch *trung vị–hậu vị* cho một ngôn ngữ gồm các dãy biểu thức kết thúc bằng các dấu chấm phẩy. Các biểu thức gồm có các số, định danh, các toán tử +, –, *, /, **div** và **mod**. Thành phẩm của chương trình dịch là dạng biểu diễn hậu vị cho mỗi biểu thức. Chương trình dịch này là một mở rộng của các chương trình đã được phát triển trong các Phần 2.5 – 2.7. Liệt kê toàn bộ chương trình C được đưa ra ở cuối phần này.

Mô tả chương trình dịch

Chương trình dịch được thiết kế bằng cách dùng lược đồ dịch dựa cú pháp trong Hình 2.35. Thẻ từ **id** biểu diễn một dãy không rỗng gồm các chữ cái và ký số bắt đầu bằng một chữ cái, **num** là một dãy ký số, và **eof** là *ký tự cuối tập tin* (end-of-file). Các thẻ từ được phân cách bằng các dãy ký tự blank, tab và newline (gọi chung là các khoảng trắng, white space). Thuộc tính *lexeme* của thẻ từ **id** chứa chuỗi ký tự tạo ra thẻ từ; thuộc tính *value* của thẻ từ **num** chứa số nguyên được biểu diễn bởi **num**.

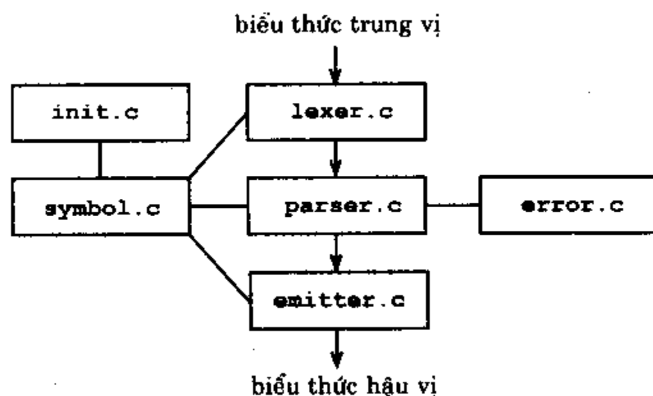
Đoạn mã cho chương trình dịch được sắp đặt vào bảy mô đun, mỗi mô đun được lưu trong một tập tin riêng. Điểm bắt đầu thực thi chương trình nằm trong mô đun `main.c` gồm có một lời gọi đến `init()` để khởi gán, theo sau là một lời gọi đến `parse()` để dịch. Sáu mô đun còn lại được trình bày trong Hình 2.36. Chúng ta cũng dùng một tập tin tiêu đề toàn cục `global.h` chứa các định nghĩa chung cho nhiều mô đun; câu lệnh đầu tiên trong mỗi mô đun là

```
#include "global.h"
```

khiến cho tập tin tiêu đề này được đưa vào trong mô đun. Trước khi trình bày đoạn mã cho chương trình dịch, chúng ta mô tả sơ lược mỗi mô đun và cách xây dựng chúng.

start	→	*list* **eof**	
list	→	*expr* ; *list*	
	|	ε	
expr	→	*expr* + *term*	{ *print*('+') }
	|	*expr* − *term*	{ *print*('−') }
	|	*term*	
term	→	*term* * *factor*	{ *print*('*') }
	|	*term* / *factor*	{ *print*('/') }
	|	*term* `div` *factor*	{ *print*('DIV') }
	|	*term* `mod` *factor*	{ *print*('MOD') }
	|	*factor*	
factor	→	(*expr*)	
	|	**id**	{ *print*(**id**.*lexeme*) }
	|	**num**	{ *print*(**num**.*value*) }

Hình 2.35. Đặc tả chương trình dịch trung vị–hậu vị.



Hình 2.36. Các mô đun cho chương trình dịch trung vị–hậu vị.

Mô đun phân tích từ vựng `lexer.c`

Thể phân từ vựng là một thủ tục có tên `lexan()` được gọi bởi thể phân cú pháp khi cần tìm các thẻ từ. Được cài đặt theo đoạn mã giả trong Hình 2.30, thủ tục này đọc nguyên liệu, mỗi lần một ký tự và trả về thẻ từ vừa tìm ra cho thể phân cú pháp. Giá trị của thuộc tính đi kèm với thẻ từ được gán cho biến toàn cục `tokenval`.

Thể phân cú pháp hy vọng sẽ nhận được các thẻ từ sau đây

```
+ - * / DIV MOD (   ) ID NUM DONE
```

Ở đây `ID` biểu diễn cho một định danh, `NUM` là một số, và `DONE` là ký tự cuối tập tin. Khoảng trắng âm thầm được loại bỏ bởi thể phân từ vựng. Bảng cho trong Hình 2.37 trình bày thẻ từ và giá trị được sinh ra bởi thể phân từ vựng cho mỗi từ tố của chương trình nguồn.

TỪ TỐ	THẺ TỪ	GIÁ TRỊ THUỘC TÍNH
khoảng trắng		
dãy ký số	NUM	Giá trị số của dãy
div	DIV	
mod	MOD	
các dãy một chữ cái rồi các chữ cái và ký số	ID	chỉ mục vào **symtable**
ký tự cuối tập tin	DONE	
một ký tự bất kỳ	ký tự đó	NONE

Hình 2.37. Mô tả các thẻ từ.

Thể phân từ vựng sử dụng thủ tục `lookup` trên bảng ký hiệu để xác định xem một từ tố cho định danh đã từng gặp trước đó hay chưa và thủ tục `insert` sẽ lưu một từ tố mới vào bảng ký hiệu. Nó cũng tăng biến toàn cục `lineno` mỗi khi gặp ký tự newline.

Mô đun phân tích cú pháp `parser.c`

Thể phân cú pháp được xây dựng bằng các kỹ thuật của Phần 2.5. Trước tiên chúng ta loại bỏ đệ qui trái ra khỏi lược đồ dịch của Hình 2.35 để văn phạm nền tảng có thể được phân tích nhờ thể phân cú pháp đệ qui xuống. Lược đồ đã biến đổi được trình bày trong Hình 2.38.

Sau đó chúng ta xây dựng các hàm cho các chưa tận *expr*, *term*, và *factor* như đã làm trong Hình 2.24. Hàm `parse()` cài đặt ký hiệu khởi đầu của văn phạm; nó gọi `lexan` mỗi khi nó cần một thẻ từ mới. Thể phân cú pháp sử dụng hàm `emit` để sinh ra kết quả và hàm `error` để ghi nhận một lỗi cú pháp.

Mô đun kết xuất `emitter.c`

Mô đun này chỉ có một hàm `emit(t, tval)` sinh ra kết quả cho thẻ từ `t` với giá trị thuộc tính `tval`.

Mô đun cho bảng ký hiệu `symbol.c` và `init.c`

Mô đun `symbol.c` cài đặt cấu trúc dữ liệu đã được trình bày trong Hình 2.29 của Phần 2.7. Các mục ghi trong mảng `symtable` là các cặp gồm có một con trỏ chỉ đến mảng `lexemes` và một số nguyên biểu thị cho thẻ từ được lưu ở đó. Thao tác `insert(s, t)` trả về chỉ mục của mục ghi trong `symtable` cho từ tố `s` đã tạo ra thẻ từ `t`. Hàm

`lookup(s)` trả về chỉ mục của mục ghi trong `symtable` cho từ số `s` hoặc trả về `0` nếu `s` không có ở đó.

Mô đun `init.c` được dùng để khởi gán các từ khóa vào bảng ký hiệu. Biểu diễn từ tố và thẻ từ cho tất cả các từ khóa được lưu trong mảng `keywords` cùng kiểu với mảng `symtable`. Hàm `init()` đi lần lượt qua mảng `keywords`, sử dụng hàm `insert` để đặt các từ khóa vào bảng ký hiệu. Cách sắp đặt này cho phép chúng ta thay đổi dạng biểu diễn của các thẻ từ cho các từ khóa một cách hết sức thuận tiện.

```
start       → list eof
list        → expr ; list
            | ε
expr        → term moreterms
moreterms   → + term { print('+') } moreterms
            | - term { print('-') } moreterms
            | ε
term        → factor morefactors
morefactors → * factor { print('*') } morefactors
            | / factor { print('/') } morefactors
            | div factor { print('DIV') } morefactors
            | mod factor { print('MOD') } morefactors
            | ε
factor      → ( expr )
            | id    { print(id.lexeme) }
            | num   { print(num.value) }
```

Hình 2.38. Lược đồ dịch dựa cú pháp sau khi đã loại bỏ đệ qui trái.

Mô đun lỗi `error.c`

Mô đun này quản lý các ghi nhận lỗi và hết sức cần thiết. Khi gặp một lỗi cú pháp, trình biên dịch in ra một thông báo cho biết rằng một lỗi đã xảy ra trên dòng nguyên liệu hiện hành rồi ngừng. Một kỹ thuật khắc phục lỗi tốt hơn có thể sẽ nhảy qua dấu chấm phẩy kế tiếp và tiếp tục phân tích; độc giả rất nên tự thực hiện các sửa đổi như thế cho chương trình dịch này. Các kỹ thuật khắc phục lỗi phức tạp hơn được trình bày trong Chương 4.

Tạo ra trình biên dịch

Đoạn mã cho các mô đun nằm trong bảy tập tin: `lexer.c`, `parser.c`, `emitter.c`, `symbol.c`, `init.c`, `error.c` và `main.c`. Tập tin `main.c` chứa thủ tục `main` trong chương trình C và nó gọi `init()`, rồi `parse()`, và hoàn tất thành công bằng `exit(0)`.

Với hệ điều hành UNIX, trình biên dịch này có thể được tạo ra bằng cách gõ lệnh

```
cc lexer.c parser.c emitter.c symbol.c init.c error.c main.c
```

hoặc biên dịch từng tập tin riêng rẽ bằng cách dùng

`cc -c` *filename*`.c`

rồi nối các tập tin kết quả *filename*`.o`:

```
cc lexer.o parser.o emitter.o symbol.o init.o error.o main.o
```

Lệnh `cc` tạo ra một tập tin `a.out` chứa chương trình dịch. Sau đó chương trình dịch có thể được cho chạy bằng cách gõ `a.out` theo sau là các biểu thức cần dịch, thí dụ

```
2+3*5;
12 div 5 mod 2;
```

hoặc bất kỳ biểu thức nào chúng ta thích. Hãy thử xem.[6]

Chương trình nguồn

Dưới đây là chương trình nguồn C cài đặt chương trình dịch này. Trước tiên là tập tin tiêu đề `global.h`, sau đó là bảy tập tin nguồn. Để cho dễ đọc, chương trình được viết theo kiểu C cơ bản.

```
/****    global.h    ****************************************/

#include <stdio.h>    /* tải các thủ tục xuất nhập */
#include <ctype.h>    /* tải các thủ tục kiểm tra ký tự */

#define BSIZE 128     /* buffer size kích thước vùng đệm */
#define NONE  -1
#define EOS   '\0'

#define NUM   256
#define DIV   257
#define MOD   258
#define ID    259
#define DONE  260

int tokenval;         /* giá trị của thuộc tính thẻ từ  */
int lineno;
```

[6] Đây là cách dịch một chương trình C trong hệ điều hành UNIX và các tập tin nguồn theo đúng bản gốc. Để chạy được với Turbo C hoặc Microsoft C, độc giả có thể phải sửa đổi tập tin nguồn và cách dịch. Chúng tôi không sửa lại bản gốc và dành cho độc giả tự tìm ra cách sửa đổi. (ND)

```
struct entry {          /* khuôn dạng cho mục ghi trong bảng ký hiệu */
     char *lexptr;
     int  token;
};

struct entry symtable[];    /* bảng ký hiệu */


/****     lexer.c     *****************************************/

#include "global.h"

char lexbuf[BSIZE];
int  lineno = 1;
int  tokenval = NONE;

int  lexan()            /* thể phân từ vựng */
{
     int t;
     while(1) {
       t = getchar();
       if (t == ' ' || t == '\t')
        ;          /* xóa các khoảng trắng */
       else if (t == '\n')
         lineno = lineno + 1;
       else if (isdigit(t)) { /* t là một ký số */
         ungetc(t, stdin);
         scanf("%d", &tokenval);
         return NUM;
       }
       else if (isalpha(t)) { /* t là một chữ cái */
         int p, b = 0;
         while (isalnum(t)) { /* t thuộc loại chữ-số */
           lexbuf[b] = t;
           t = getchar();
           b = b + 1;
           if (b >= BSIZE)
             error("compiler error");
         }
         lexbuf[b] = EOS;
         if (t != EOF)
           ungetc(t, stdin);
```

```
            p = lookup(lexbuf);
            if (p == 0)
               p = insert(lexbuf, ID);
            tokenval = p;
            return symtable[p].token;
         }
         else if (t == EOF)
            return DONE;
         else {
            tokenval = NONE;
            return t;
         }
      }
}

/****    parser.c    ****************************************/

#include "global.h"

int lookahead;

parse()              /* phân tích cú pháp và dịch danh sách biểu thức */
{
      lookahead = lexan();
      while (lookahead != DONE) {
         expr(); match(';');
      }
}

expr()
{
      int t;
      term();
      while(1)
         switch(lookahead) {
            case '+': case '-':
               t = lookahead;
               match(lookahead); term(); emit(t, NONE);
               continue;
            default:
               return;
         }
}
```

```
term()
{
      int t;
      factor();
      while(1)
         switch(lookahead) {
            case '*': case '/': case DIV: case MOD:
               t = lookahead;
               match(lookahead); factor(); emit(t, NONE);
               continue;
            default:
               return;
         }
}

factor()
{
      switch(lookahead) {
         case '(':
            match('('); expr(); match(')'); break;
         case NUM:
            emit(NUM, tokenval); match(NUM); break;
         case ID:
            emit(ID, tokenval); match(ID); break;
         default:
            error("syntax error");
      }
}

match(t)
      int t;
{
      if (lookahead == t)
         lookahead = lexan();
      else error("syntax error");
}
```

```
/****    emitter.c    ****************************************/
#include "global.h"
emit(t, tval)          /* tạo ra kết quả */
     int t, tval;
{
     switch(t) {
        case '+': case '-': case '*': case '/':
          printf("%c\n", t); break;
        case DIV:
          printf("DIV\n"); break;
        case MOD:
          printf("MOD\n"); break;
        case NUM:
          printf("%d\n", tval); break;
        case ID:
          printf("%s\n", symtable[tval].lexptr); break;
        default:
          printf("token %d, tokenval %d\n", t, tval);
     }
}

/****    symbol.c    ****************************************/
#include "global.h"
#define STRMAX 999    /* kích thước mảng lexemes */
#define SYMMAX 100    /* kích thước bảng symtable */
char lexemes[STRMAX];
int  lastchar = -1;   /* vị trí được dùng cuối cùng trong lexemes */
struct entry symtable[SYMMAX];
int  lastentry = 0;   /* vị trí được dùng cuối cùng trong symtable */
int  lookup(s)        /* trả về vị trí của mục ghi cho s */
     char s[];
{
     int p;
     for (p = lastentry; p > 0; p = p - 1)
        if (strcmp(symtable[p].lexptr, s) == 0)
          return p;
     return 0;
}
```

```
int  insert(s, tok)   /* trả về vị trí của mục ghi cho  s */
     char s[];
     int  tok;
{
      int len;
      len = strlen(s);/* strlen tính chiều dài của  s */
      if (lastentry + 1 >= SYMMAX)
        error("symbol table full");
      if (lastchar + len + 1 >= STRMAX)
        error("lexemes array full");
      lastentry = lastentry + 1;
      symtable[lastentry].token = tok;
      symtable[lastentry].lexptr = &lexemes[lastchar + 1];
      lastchar = lastchar + len + 1;
      strcpy(symtable[lastentry].lexptr, s);
      return lastentry;
}


/****   init.c   ******************************************/

#include "global.h"

struct entry keywords[] = {
      "div", DIV,
      "mod", MOD,
      0,   0
};

init()              /* đưa các từ khóa vào symtable */
{
      struct entry *p;
      for (p = keywords; p->token; p++)
        insert(p->lexptr, p->token);
}
```

```
int  insert(s, tok)   /* trả về vị trí của mục ghi cho s */
     char s[];
     int  tok;
{
     int len;
     len = strlen(s);/* strlen tính chiều dài của s */
     if (lastentry + 1 >= SYMMAX)
       error("symbol table full");
     if (lastchar + len + 1 >= STRMAX)
       error("lexemes array full");
     lastentry = lastentry + 1;
     symtable[lastentry].token = tok;
     symtable[lastentry].lexptr = &lexemes[lastchar + 1];
     lastchar = lastchar + len + 1;
     strcpy(symtable[lastentry].lexptr, s);
     return lastentry;
}


/****   init.c   ********************************************/

#include "global.h"

struct entry keywords[] = {
     "div", DIV,
     "mod", MOD,
     0,   0
};

init()               /* đưa các từ khóa vào symtable */
{
     struct entry *p;
     for (p = keywords; p->token; p++)
       insert(p->lexptr, p->token);
}
```

```
/****   error.c   ******************************************/
#include "global.h"

error(m)               /* sinh ra tất cả các thông báo lỗi */
    char *m;
{
    fprintf(stderr, "line %d: %s\n", lineno, m);
    exit(1);           /* kết thúc không thành công */
}


/****  main.c   *******************************************/
#include "global.h"

main()
{
    init();
    parse();
    exit(0);           /* kết thúc thành công */
}
/**********************************************************/
```

BÀI TẬP

2.1 Xét văn phạm phi ngữ cảnh

$$S \rightarrow S\,S\,+ \mid S\,S\,* \mid \mathtt{a}$$

a) Trình bày cách tạo ra chuỗi **aa+a*** từ văn phạm trên.

b) Xây dựng một cây phân tích cú pháp cho chuỗi này.

c) Ngôn ngữ nào được sinh ra bởi văn phạm trên? Biện minh cho câu trả lời.

2.2 Ngôn ngữ nào được sinh ra bởi các văn phạm sau đây? Trong mỗi trường hợp hãy biện minh cho câu trả lời.

a) $S \rightarrow 0\,S\,1 \mid 0\,1$

b) $S \rightarrow +\,S\,S \mid -\,S\,S \mid \mathtt{a}$

c) $S \rightarrow S\,(\,S\,)\,S \mid \varepsilon$

d) $S \rightarrow \mathbf{a}\,S\,\mathbf{b}\,S \mid \mathbf{b}\,S\,\mathbf{a}\,S \mid \varepsilon$

e) $S \rightarrow \mathtt{a} \mid S\,+\,S \mid S\,S \mid S\,* \mid (\,S\,)$

2.3 Văn phạm nào trong Bài tập 2.2 là đa nghĩa?

2.4 Xây dựng các văn phạm phi ngữ cảnh đơn nghĩa cho mỗi ngôn ngữ sau đây. Trong mỗi trường hợp hãy chứng tỏ rằng văn phạm được xây dựng là chính xác.

a) Các biểu thức số học ở dạng hậu vị.

b) Các danh sách định danh có tính kết hợp trái được phân cách bởi dấu phẩy.

c) Các danh sách định danh có tính kết hợp phải được phân cách bởi dấu phẩy.

d) Các biểu thức số học của số nguyên và định danh với bốn toán tử hai ngôi +, -, *, /.

e) Thêm toán tử cộng và trừ đơn ngôi vào các toán tử số học của (d).

***2.5** a) Chứng tỏ rằng mọi chuỗi nhị phân được sinh bởi văn phạm sau đây đều có giá trị chia hết cho 3. *Hướng dẫn*: Dùng qui nạp trên số nút trong cây phân tích cú pháp.

num → 1 1 | 1 0 0 1 | *num* 0 | *num num*

b) Văn phạm này có sinh ra tất cả mọi chuỗi nhị phân có giá trị chia hết cho 3 hay không?

2.6 Xây dựng một văn phạm phi ngữ cảnh cho các số La mã.

2.7 Xây dựng một lược đồ dịch dựa cú pháp để dịch các biểu thức số học từ ký pháp trung vị sang ký pháp tiền vị, nghĩa là một toán tử sẽ xuất hiện trước các toán hạng của nó; thí dụ **-xy** là ký pháp tiền vị cho **x-y**. Vẽ các cây phân tích cú pháp chú giải cho các nguyên liệu 9-5+2 và 9-5*2.

2.8 Xây dựng một lược đồ dịch dựa cú pháp để dịch các biểu thức từ dạng hậu vị sang dạng trung vị. Vẽ các cây phân tích cú pháp cho các nguyên liệu 95-2* và 952*-.

2.9 Xây dựng một lược đồ dịch dựa cú pháp để dịch các số nguyên thành các số La mã.

2.10 Xây dựng một lược đồ dịch dựa cú pháp dịch các số La mã thành các số nguyên.

2.11 Xây dựng các thể phân cú pháp đệ qui xuống cho các văn phạm trong Bài tập 2.2 (a), (b) và (c).

2.12 Xây dựng một chương trình dịch dựa cú pháp xác nhận rằng các dấu ngoặc trong một chuỗi nguyên liệu đúng là cân bằng.

2.13 Các qui tắc sau đây định nghĩa bản dịch một từ tiếng Anh sang tiếng Latin giản lược:

a) Nếu một từ bắt đầu bằng một chuỗi phụ âm không rỗng, di chuyển chuỗi phụ âm đi đầu ra sau từ đó rồi gắn thêm hậu tố `AY`; thí dụ `pig` trở thành `igpay`.

b) Nếu một từ bắt đầu bằng một nguyên âm, thêm hậu tố `YAY`; thí dụ `owl` trở thành `owlyay`.

c) `U` có một `Q` theo sau là một nguyên âm.

d) `Y` ở đầu một từ là một nguyên âm nếu sau nó không phải là một nguyên âm.

e) Các từ chỉ có một chữ cái không bị thay đổi.

Xây dựng một lược đồ dịch dựa cú pháp cho tiếng Latin giản lược.

2.14 Trong ngôn ngữ lập trình C, câu lệnh `for` có dạng

for ($expr_1$; $expr_2$; $expr_3$ **)** *stmt*

Biểu thức thứ nhất được thực hiện trước khi vào vòng lặp; thường thì nó được dùng để khởi gán chỉ số vòng. Biểu thức thứ hai là một phép thử được thực hiện trước mỗi lần lặp của vòng; vòng sẽ được thoát ra nếu biểu thức này trở thành 0. Bản thân vòng lặp gồm có câu lệnh { *stmt* $expr_3$; }. Biểu thức thứ ba được thực hiện vào cuối mỗi lần lặp; điển hình nó được dùng để làm tăng chỉ số vòng. Ý nghĩa của câu lệnh `for` tương tự như

$expr_1$; **while (** $expr_2$ **)** { *stmt* $expr_3$; }

Xây dựng một lược đồ dịch dựa cú pháp để dịch các câu lệnh `for` thành mã máy chồng xếp.

*** 2.15** Xét câu lệnh `for` sau đây.

for i **:= 1 step** $10 - j$ **until** $10 * j$ **do** $j := j + 1$

Ba định nghĩa về ngữ nghĩa có thể gán cho câu lệnh này. Một ý nghĩa có thể yêu cầu rằng giới hạn $10 * j$ và trị tăng $10 - j$ phải được ước lượng một lần trước vòng lặp giống như trong PL/I. Thí dụ nếu $j = 5$ trước vòng lặp thì chúng ta sẽ chạy qua vòng 10 lần rồi thoát. Ý nghĩa thứ hai hoàn toàn khác hẳn, đó là chúng ta bắt buộc phải ước lượng giới hạn và trị tăng mỗi lần qua vòng lặp. Thí dụ nếu $j = 5$ trước vòng lặp, vòng lặp đó sẽ không bao giờ kết thúc. Ý nghĩa thứ ba được cho giống như trong các ngôn ngữ như Algol. Khi trị tăng trở thành âm, phép thử khiến kết thúc vòng lặp sẽ là $i < 10 * j$ chứ không phải $i > 10 * j$. Với mỗi ngữ nghĩa này, hãy xây dựng một lược đồ dịch để dịch những vòng `for` thành mã máy chồng xếp.

2.16 Xét đoạn văn phạm sau đây cho các câu lệnh **if-then** và **if-then-else**:

stmt → **if** *expr* **then** *stmt*
  | **if** *expr* **then** *stmt* **else** *stmt*
  | **other**

trong đó **other** đại diện cho những câu lệnh khác của ngôn ngữ.

a) Chứng tỏ rằng văn phạm này là đa nghĩa.

b) Xây dựng một văn phạm đơn nghĩa tương đương với mỗi **else** được đối sánh với **then** gần nhất trước nó và chưa được đối sánh.

c) Xây dựng một lược đồ dịch dựa cú pháp dựa trên văn phạm này để dịch các câu lệnh điều kiện thành mã máy chồng xếp.

***2.17** Xây dựng một lược đồ dịch dựa cú pháp để dịch các biểu thức số học dạng trung vị thành dạng trung vị không có dấu ngoặc thừa. Trình bày cây phân tích cú pháp chú giải cho nguyên liệu `(((1+2)*(3*4))+5)`.

BÀI TẬP LẬP TRÌNH

P2.1 Cài đặt một chương trình dịch từ số nguyên thành các số La mã dựa trên lược đồ dịch dựa cú pháp đã xây dựng trong Bài tập 2.9.

P2.2 Sửa lại chương trình dịch trong Phần 2.9 cho sinh ra đoạn mã cho máy chồng xếp trừu tượng của Phần 2.8.

P2.3 Sửa lại mô đun khắc phục lỗi của chương trình dịch trong Phần 2.9 để nhảy đến biểu thức kế tiếp khi gặp một lỗi.

P2.4 Mở rộng chương trình dịch trong Phần 2.9 để xử lý tất cả mọi biểu thức trong Pascal.

P2.5 Mở rộng trình biên dịch của Phần 2.9 để dịch sang các câu lệnh của máy chồng xếp các câu lệnh được sinh ra bởi văn phạm sau đây:

stmt → **id** := *expr*
  | **if** *expr* **then** *stmt*
  | **while** *expr* **do** *stmt*
  | **begin** *opt_stmts* **end**
opt_stmts → *stmt_list* | ε
stmt_list → *stmt_list* ; *stmt* | *stmt*

***P2.6** Xây dựng một tập các biểu thức thử nghiệm cho trình biên dịch trong Phần 2.9 để mỗi luật sinh được dùng ít nhất một lần khi dẫn xuất một biểu thức thử nghiệm nào đó. Xây dựng một chương trình kiểm tra có thể được dùng làm

một công cụ kiểm tra trình biên dịch tổng quát. Sử dụng chương trình kiểm tra để chạy trình biên dịch trên những biểu thức thử nghiệm.

P2.7 Xây dựng một tập các câu lệnh thử nghiệm cho trình biên dịch của Bài tập P2.5 để mỗi luật sinh được dùng ít nhất một lần khi tạo ra một câu lệnh thử nghiệm. Sử dụng chương trình kiểm tra của Bài tập P2.6 để chạy trình biên dịch trên những biểu thức thử nghiệm.

GHI CHÚ VỀ TÀI LIỆU THAM KHẢO

Chương giới thiệu mở đầu này đã lướt qua một số đề tài sẽ được phân tích chi tiết hơn trong phần còn lại của cuốn sách. Các chỉ dẫn tham khảo xuất hiện trong các chương đó có nhiều chất liệu hơn.

Văn phạm phi ngữ cảnh đã được Chomsky [1956] đưa ra như một thành phần nghiên cứu về ngôn ngữ tự nhiên. Việc sử dụng chúng để đặc tả cú pháp của ngôn ngữ lập trình đã xuất hiện độc lập nhau. Trong khi xây dựng bản dự thảo của Algol 60, John Backus "đã vội vã áp dụng [các luật sinh của Emil Post]" (Wexelblat [1981], trang 162). Ký pháp thu được là một biến thể của văn phạm phi ngữ cảnh. Nhà nghiên cứu Panini đã nghĩ ra một ký pháp ngữ pháp tương đương để mô tả các qui tắc của văn phạm Sanskrit khoảng những năm 400 đến 200 trước công nguyên (Ingerman [1967]).

Đề xuất nên đọc BNF là Backus-Naur Form mà lúc đầu là chữ viết tắt của Backus Normal Form để ghi nhận các đóng góp của Naur như một người đã biên tập cho bản dự thảo Algol 60 (Naur [1963]) đã được đưa ra trong một bức thư của Knuth [1964].

Định nghĩa dựa cú pháp là một dạng định nghĩa qui nạp, trong đó phép qui nạp được thực hiện trên cấu trúc cú pháp. Như thế từ lâu chúng đã được dùng một cách không hình thức trong toán học. Ứng dụng của chúng cho các ngôn ngữ lập trình đã xảy ra cùng với việc sử dụng một văn phạm để xây dựng báo cáo Algol 60. Ngay sau đó, Irons [1961] đã xây dựng một trình biên dịch dựa cú pháp.

Phân tích cú pháp đệ qui xuống đã được dùng từ những năm đầu của thập niên 60. Bauer [1976] xem phương pháp này là của Lucas [1961]. Hoare [1962b, trang 128] mô tả một trình biên dịch Algol được tổ chức như một "tập các thủ tục, mỗi thủ tục có khả năng xử lý một trong các đơn vị cú pháp của báo cáo Algol." Foster [1968] thảo luận phương pháp loại bỏ đệ qui trái từ các luật sinh chứa các hành động ngữ nghĩa và không làm ảnh hưởng đến các giá trị thuộc tính.

McCarthy [1963] đã cho rằng dịch một ngôn ngữ cần dựa trên cú pháp trừu tượng. Cũng trong bài báo này, McCarthy [1963, trang 24] để "độc giả tự thuyết phục" rằng việc trình bày theo lối đệ qui đuôi cho hàm giai thừa thì tương đương với một chương trình lặp.

Ích lợi của việc phân chia trình biên dịch thành kỳ đầu và kỳ sau đã được khám

phá bởi Strong el al. [1958] trong một báo cáo hội nghị. Báo cáo này đặt tên UNCOL (từ chữ universal computer oriented language) cho *ngôn ngữ trung gian phổ quát* (universal imtermediate language). Khái niệm này cho đến nay vẫn còn là một điều lý tưởng.

Có một cách rất hay để học các kỹ thuật cài đặt là đọc các đoạn mã chương trình của những trình biên dịch đã có. Không may là những phần này thường không được công bố. Randell and Russell [1964] trình bày một bản mô tả dễ hiểu của trình biên dịch Algol sơ khởi. Đoạn mã của trình biên dịch cũng có thể xem trong McKeeman, Horning và Wortman [1970]. Barron [1981] là một tập hợp các bài báo cài đặt Pascal, bao gồm các ghi chép được phổ biến kèm với trình biên dịch Pascal P (Nori et al., 1981]), chi tiết sinh mã (Ammann [1977]) và đoạn mã chương trình của một cài đặt Pascal S, là một tập con của Pascal được thiết kế bởi Wirth [1981] dành cho sinh viên. Knuth [1985] đưa ra một mô tả chi tiết và rõ ràng về chương trình dịch T_EX.

Kernighan and Pike [1984] mô tả chi tiết cách xây dựng một chương trình máy tính bỏ túi qua một lược đồ dịch dựa cú pháp có sử dụng các công cụ xây dựng trình biên dịch có sẵn trên hệ điều hành UNIX. Phương trình (2.17) được lấy từ Tantzen [1963].

CHƯƠNG 3

Phân Tích Từ Vựng

Chương này sẽ đề cập đến những kỹ thuật đặc tả và cài đặt thể phân từ vựng. Có một cách đơn giản để xây dựng một thể phân từ vựng là tạo ra một sơ đồ minh họa cấu trúc các thẻ từ của ngôn ngữ nguồn rồi dịch sơ đồ thành một chương trình tìm kiếm các thẻ từ. Nhiều thể phân từ vựng hiệu quả có thể được tạo ra theo cách này.

Kỹ thuật được dùng cài đặt thể phân từ vựng cũng có thể được áp dụng cho các lãnh vực khác, chẳng hạn trong các ngôn ngữ vấn tin và các hệ thống truy xuất thông tin. Trong mỗi ứng dụng, bài toán cơ bản là đặc tả và thiết kế các chương trình thực hiện các hành động được kích hoạt bởi các *mẫu* (pattern) trong các chuỗi. Bởi vì vấn đề *lập trình theo mẫu* (pattern-directed programming) có nhiều công dụng nên chúng ta sẽ giới thiệu một ngôn ngữ *mẫu-hành động* có tên là Lex để đặc tả thể phân từ vựng. Trong ngôn ngữ này, các mẫu được đặc tả bằng các *biểu thức chính qui* (regular expression) và một trình biên dịch cho Lex có thể tạo ra một *thể nhận dạng automat hữu hạn* (finite automata recognizer) hiệu quả cho các biểu thức chính qui này.

Nhiều ngôn ngữ sử dụng biểu thức chính qui để mô tả các mẫu. Chẳng hạn ngôn ngữ AWK sử dụng biểu thức chính qui để chọn các dòng nguyên liệu cần xử lý, hệ thống Shell của UNIX cho phép người sử dụng tham chiếu một tập các tên tập tin bằng cách viết một biểu thức chính qui. Chẳng hạn lệnh `rm *.o` xóa mọi tập tin có tên kết thúc bằng "`.o`".[1]

Một công cụ phần mềm tự động xây dựng thể phân từ vựng cho phép nhiều người với những hiểu biết khác nhau có thể áp dụng kỹ thuật so mẫu (đối sánh mẫu) vào những lãnh vực chuyên môn của họ. Thí dụ Jarvis [1976] đã sử dụng một *bộ sinh thể phân từ vựng* (lexical-analyzer generator) để tạo ra một chương trình nhận dạng những sai sót trong các bo mạch in. Các bo mạch này sẽ được quét rồi được biến đổi thành chuỗi các đoạn thẳng ở những góc khác nhau. "Thể phân từ vựng" sẽ tìm các

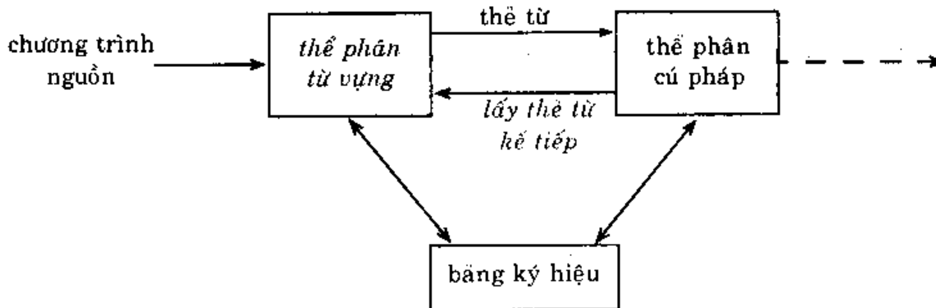
---

[1] Biểu thức `*.o` là một biến thể của ký pháp thông thường cho các biểu thức chính qui. Bài tập 3.10 và 3.14 đề cập đến các biến thể thường gặp của ký pháp biểu thức chính qui.

mẫu tương ứng với các khiếm khuyết trong chuỗi đoạn thẳng. Ưu điểm chủ yếu của bộ sinh thể phân từ vựng là nó có thể sử dụng được những thuật toán so mẫu tốt nhất và vì thế tạo ra được những thể phân từ vựng hoạt động rất hiệu quả cho những người không phải là những chuyên gia về các kỹ thuật so mẫu.

3.1 VAI TRÒ CỦA THỂ PHÂN TỪ VỰNG

Phân tích từ vựng là giai đoạn đầu tiên của quá trình biên dịch. Nhiệm vụ chủ yếu của nó là đọc các ký tự nhập (nguyên liệu) rồi tạo ra một chuỗi thẻ từ cho thể phân cú pháp sử dụng trong giai đoạn phân tích cú pháp. Tương tác này, được tóm tắt qua sơ đồ trong Hình 3.1, thường được cài đặt bằng cách cho thể phân từ vựng làm một thủ tục con hoặc một đồng thủ tục với thể phân tích cú pháp. Khi nhận được yêu cầu "lấy thẻ từ tiếp theo" từ thể phân cú pháp, thể phân từ vựng sẽ đọc các ký tự cho đến khi nó nhận diện ra được một thẻ từ.



Hình 3.1. Tương tác của thể phân từ vựng với thể phân cú pháp.

Bởi vì thể phân từ vựng là thành phần đọc chương trình nguồn của trình biên dịch, nó thường thực hiện thêm một số tác vụ khác ở mức giao diện người sử dụng. Một tác vụ là lược bỏ các *lời giải thích* (comment) và các khoảng trắng (ký tự trống, ký tự tab, và ký tự newline). Một tác vụ khác là liên kết các thông báo lỗi của trình biên dịch với chương trình nguồn. Chẳng hạn thể phân từ vựng có thể theo dõi số lượng các *ký tự xuống hàng* (newline character), nhờ vậy có thể liên kết chỉ số của một hàng với một thông báo lỗi. Trong một số trình biên dịch, thể phân từ vựng chịu trách nhiệm tạo ra một bản sao chương trình nguồn có kèm theo các thông báo lỗi được đánh dấu vào trong đó. Nếu ngôn ngữ nguồn có hỗ trợ một số chức năng của bộ tiền xử lý macro thì những chức năng này thường được cài đặt khi thực hiện phân tích từ vựng.

Đôi khi thể phân từ vựng được chia làm hai giai đoạn nối tiếp nhau. Giai đoạn đầu "quét" nguyên liệu và giai đoạn sau mới là giai đoạn phân tích từ vựng. Chương trình quét nguyên liệu chịu trách nhiệm thực hiện một số tác vụ đơn giản còn bản thân thể phân từ vựng thực hiện các tác vụ phức tạp hơn. Chẳng hạn trình biên dịch Fortran có thể dùng một chương trình quét để loại bỏ các ký tự trống ra khỏi nguyên liệu.

Các vấn đề của giai đoạn phân tích từ vựng

Có rất nhiều lý do để chia giai đoạn phân tích thành hai giai đoạn riêng rẽ: phân tích từ vựng và phân tích cú pháp.

1. Làm cho việc thiết kế đơn giản hơn là lý do quan trọng nhất. Tách phân tích từ vựng ra khỏi phân tích cú pháp cho phép chúng ta đơn giản hóa từng giai đoạn. Chẳng hạn một thể phân cú pháp phải lo xử lý các lời giải thích và khoảng trắng sẽ phức tạp hơn rất nhiều so với một thể phân cú pháp không phải xử lý các lời giải thích và các khoảng trắng vì chúng đã được thể phân từ vựng loại bỏ trước rồi. Nếu chúng ta dự định thiết kế một ngôn ngữ mới, việc tách các qui ước từ vựng ra khỏi qui ước cú pháp có thể dẫn đến một thiết kế dễ hiểu hơn.

2. Hiệu quả của trình biên dịch được cải thiện. Một thể phân từ vựng riêng rẽ cho phép chúng ta xây dựng được một chương trình xử lý chuyên dụng và hiệu quả hơn cho tác vụ này. Một phần thời gian khá lớn được dành để đọc chương trình nguồn và để tách chúng thành các thẻ từ. Các kỹ thuật đệm đặc dụng dành để đọc các ký tự và xử lý thẻ từ có thể làm tăng đáng kể hiệu năng của trình biên dịch.

3. Tính đa tương thích (mang đi dễ dàng) của trình biên dịch cũng được cải thiện. Đặc tính của bộ ký tự nhập và những dị biệt của từng loại thiết bị có thể được giới hạn trong thể phân từ vựng. Dạng biểu diễn của các ký hiệu đặc biệt hoặc là những ký hiệu không chuẩn, chẳng hạn như ký hiệu ↑ trong Pascal có thể được cô lập trong thể phân từ vựng.

Có những công cụ chuyên dụng được thiết kế nhằm tự động hóa việc xây dựng các thể phân từ vựng và cú pháp khi chúng được tách riêng. Chúng ta sẽ gặp một số thí dụ về những công cụ như thế trong cuốn sách này.

Thẻ từ, mẫu từ và từ tố

Khi nói đến việc phân tích từ vựng, chúng ta sẽ sử dụng các thuật ngữ *thẻ từ* (token), *mẫu từ* (pattern) và *từ tố* (lexeme) với nghĩa cụ thể. Một số thí dụ về cách dùng những thuật ngữ này được trình bày trong Hình 3.2. Tổng quát, có một tập chuỗi ký tự trong *nguyên liệu* (input) có thể sinh ra cùng một thẻ từ. Tập các chuỗi này được mô tả bằng một qui tắc gọi là *mẫu từ* đi kèm với thẻ từ đó. Mẫu từ này được gọi là "so khớp" (đối sánh được) với mỗi chuỗi trong tập. Một từ tố là một chuỗi các ký tự trong chương trình nguồn đối sánh được với mẫu của một thẻ từ. Chẳng hạn trong câu lệnh Pascal

```
const pi = 3.1416;
```

chuỗi con "`pi`" là một từ tố cho thẻ từ "identifier".

Chúng ta xem các thẻ từ như là các *ký hiệu tận* (terminal symbol) hay nói gọn là *tận* (terminal) trong văn phạm của ngôn ngữ nguồn và in đậm tên để biểu thị thẻ từ. Các từ tố khớp với mẫu của thẻ từ đó biểu thị chuỗi ký tự trong chương trình nguồn có thể được xem như một *đơn vị từ vựng* (lexical unit).

THẺ TỪ	TỪ TỐ MINH HỌA	MÔ TẢ KHÔNG HÌNH THỨC CÁC MẪU
const	`const`	`const`
if	`if`	`if`
relation	<, <=, =, <>, >, >=	< hoặc <= hoặc = hoặc <> hoặc >= hoặc >
id	`pi`, `count`, `D2`	chữ cái theo sau là những chữ cái hoặc ký số
num	`3.1416`, `0`, `6.02E23`	một hằng số bất kỳ
literal	`"core dumped"`	mọi chữ cái nằm giữa " và " ngoại trừ "

Hình 3.2. Các thí dụ về thẻ từ.

Trong hầu hết các ngôn ngữ lập trình, các kết cấu sau đây được xử lý như các thẻ từ: *từ khóa* (keyword), *toán tử* (operator), *hằng* (constant), *chuỗi trực kiện* (literal) và các *dấu chấm câu* (punctuation) như *dấu ngoặc đơn* (parentheses), *dấu phẩy* (comma) và *dấu chấm phẩy* (semicolon). Trong thí dụ trên, khi chuỗi ký tự `pi` xuất hiện trong chương trình nguồn, một thẻ từ biểu thị cho một định danh được trả về cho thể phân cú pháp. Trả về một thẻ từ thường được cài đặt bằng cách truyền một số nguyên tương ứng với thẻ từ. Số nguyên này trong Hình 3.2 được biểu diễn bằng chữ in đậm **id**.

Một mẫu từ là một qui tắc mô tả tập từ tố có thể biểu diễn một thẻ từ cụ thể trong các chương trình nguồn. Mẫu cho thẻ từ **const** trong Hình 3.2 chỉ là chuỗi `const` tương ứng với một từ khóa. Mẫu cho thẻ từ **relation** là tập tất cả sáu toán tử quan hệ của Pascal. Để mô tả chính xác các mẫu cho các thẻ từ phức tạp như **id** (do chữ identifier, nghĩa là định danh) và **num** (number, các số), chúng ta dùng ký pháp biểu thức chính qui sẽ được phát triển trong Phần 3.3.

Một số qui ước của ngôn ngữ gây nhiều khó khăn cho việc phân tích từ vựng. Các ngôn ngữ như Fortran đòi hỏi một số kết cấu phải nằm ở những vị trí cố định trên các dòng. Vì thế việc canh lề cho một từ tố có thể cần thiết khi xác định tính đúng đắn của một chương trình nguồn. Xu hướng thiết kế các ngôn ngữ hiện đại là dùng nguyên liệu phi định dạng, cho phép các kết cấu nằm tại một vị trí bất kỳ trên các dòng chương trình, vì vậy điều kiện này hiện không còn quan trọng nữa.

Xử lý các ký tự trống có nhiều khác biệt tùy theo từng ngôn ngữ. Trong một số ngôn ngữ như Fortran hoặc Algol 68, các khoảng trống không có ý nghĩa gì trừ khi

chúng nằm trong *chuỗi trực kiện* (literal). Chúng có thể được thêm vào tùy ý cho chương trình dễ đọc. Các qui ước liên quan đến các khoảng trống có thể làm phức tạp thêm cho công việc xác định các thẻ từ.

Một thí dụ minh họa tính chất khó khăn khi nhận dạng các thẻ từ là câu lệnh DO của Fortran. Trong câu lệnh

```
DO 5 I = 1.25
```

Chúng ta không thể nói gì cho đến khi thấy được dấu chấm thập phân, nhận ra rằng DO không phải là từ khóa, nhưng là thành phần của định danh DO5I. Ngược lại trong câu lệnh

```
DO 5 I = 1,25
```

Chúng ta có bảy thẻ từ, tương ứng với từ khóa DO, nhãn lệnh 5, định danh I, toán tử =, hằng 1, dấu phẩy, và hằng 25. Ở đây chúng ta không dám khẳng định gì cho đến khi chúng ta nhìn thấy dấu phẩy, cho thấy DO là từ khóa. Để né tránh tình huống không chắc chắn này, Fortran 77 cho phép có dấu phẩy tùy ý giữa nhãn và chỉ mục của câu lệnh DO. Việc sử dụng dấu phẩy được khuyến khích bởi vì nó làm cho câu lệnh DO rõ ràng hơn và dễ đọc hơn.

Trong nhiều ngôn ngữ, một số chuỗi ký tự được *dành riêng* (reserved); điều này muốn nói là ý nghĩa của chúng được định nghĩa trước và người sử dụng không thể thay đổi được. Nếu các từ khóa không được dành riêng thì thể phân từ vựng phải phân biệt giữa một từ khóa và một định danh do người sử dụng định nghĩa. Trong PL/I, từ khóa không được dành riêng; vì thế các qui tắc phân biệt các từ khóa với các định danh hết sức phức tạp, như được minh họa qua câu lệnh PL/I sau:

```
IF THEN THEN THEN = ELSE; ELSE ELSE = THEN;
```

Thuộc tính của thẻ từ

Khi có nhiều mẫu từ cùng khớp được với một từ tố, thể phân từ vựng phải cung cấp thêm thông tin về từ tố đã khớp cho các pha biên dịch sau đó. Chẳng hạn mẫu **num** khớp với cả hai chuỗi 0 và 1, nhưng thể sinh mã phải biết cụ thể là chuỗi nào đã khớp.

Thể phân từ vựng đưa thông tin về các thẻ từ vào các thuộc tính đi kèm của chúng. Các thẻ từ có ảnh hưởng đến các quyết định phân tích cú pháp; các thuộc tính ảnh hưởng đến việc phiên dịch các thẻ từ. Vấn đề thực hành là, một thẻ từ thường chỉ có một thuộc tính — đó là một con trỏ chỉ đến một mục ghi trong bảng ký hiệu có chứa thông tin về thẻ từ; con trỏ trở thành thuộc tính của thẻ từ. Để dễ chẩn đoán lỗi, chúng ta có thể quan tâm đến cả từ tố của một định danh lẫn chỉ số dòng có lỗi được phát hiện ra lần đầu tiên. Cả hai thông tin này đều có thể được lưu vào mục ghi dành cho định danh trong bảng.

Thí dụ 3.1. Thẻ từ và giá trị thuộc tính đi kèm của câu lệnh Fortran

```
E = M * C ** 2
```

được viết như một dãy các cặp:

<**id**, con trỏ chỉ đến mục ghi cho `E` trong bảng ký hiệu >
<**assign_op**, >
<**id**, con trỏ chỉ đến mục ghi cho `M` trong bảng ký hiệu>
<**mult_op**, >
<**id**, con trỏ chỉ đến mục ghi cho `C` trong bảng ký hiệu>
<**exp_op**, >
<**num**, giá trị nguyên 2>

Chú ý rằng một số cặp không cần giá trị thuộc tính; thành phần đầu tiên là đủ để nhận dạng từ tố. Trong thí dụ nhỏ ở trên, thẻ từ **num** đã được cho một giá trị nguyên. Trình biên dịch có thể lưu chuỗi ký tự đã tạo ra một số vào bảng ký hiệu và để thuộc tính của thẻ từ **num** là một con trỏ chỉ đến mục ghi đó của bảng. □

Lỗi từ vựng

Có rất ít lỗi có thể phát hiện ra ở mức độ từ vựng bởi vì thể phân từ vựng có một hình ảnh rất cục bộ về chương trình nguồn. Nếu chuỗi `fi` được gặp trong một chương trình C vào lần đầu tiên trong ngữ cảnh

```
fi (a == f(x) ) . . .
```

thể phân từ vựng không thể xác định được rằng `fi` là từ khóa `if` bị viết sai hay đây là một định danh hàm chưa được định nghĩa. Vì `fi` là một định danh hợp lệ, thể phân từ vựng sẽ trả về thẻ từ cho định danh và để một pha biên dịch khác xử lý nếu có lỗi.

Nhưng giả sử xảy ra một tình huống mà thể phân từ vựng không thể tiếp tục được bởi vì không có mẫu từ nào cho các thẻ từ khớp được với một *tiền tố* (prefix) của phần nguyên liệu còn lại. Rất có thể chiến lược khắc phục đơn giản nhất là "*thể thức hoảng sợ*" (panic mode). Chúng ta sẽ xóa các ký tự tiếp theo ra khỏi phần nguyên liệu còn lại cho đến khi thể phân từ vựng có thể tìm ra được một thẻ từ hoàn chỉnh. Kỹ thuật khắc phục này đôi khi gây nhầm lẫn cho thể phân cú pháp nhưng trong môi trường xử lý tương tác thì có thể dùng được.

Các hành động khắc phục lỗi khác có thể là:

1. xóa một ký tự dư
2. chèn một ký tự thiếu
3. thay một ký tự sai bằng một ký tự đúng
4. đổi chỗ hai ký tự kế cận

Biến đổi để khắc phục lỗi như trên có thể được thử với hy vọng sẽ sửa được nguyên liệu. Chiến lược đơn giản nhất đó muốn thử xem có thể biến đổi một tiền tố của phần nguyên liệu còn lại thành một từ tố hợp lệ chỉ bằng một biến đổi duy nhất. Nó giả thiết rằng phần lớn lỗi từ vựng là kết quả của một biến đổi duy nhất, là điều thường xảy ra trong thực hành.

Một cách tìm các lỗi trong một chương trình là tính số lượng biến đổi ít nhất cần để biến một chương trình lỗi thành một chương trình chỉnh dạng về mặt cú pháp. Chúng ta nói rằng một chương trình lỗi có *k* lỗi nếu chuỗi biến đổi ngắn nhất chuyển được nó thành một chương trình hợp lệ có chiều dài là *k*. Chỉnh lỗi với số biến đổi ít nhất là một chuẩn mực hợp lý về lý thuyết, nhưng nói chung thường không được dùng trong thực hành bởi vì chi phí quá cao khi cài đặt nó. Tuy nhiên một số ít trình biên dịch thử nghiệm đã sử dụng tiêu chuẩn này để hiệu chỉnh cục bộ.

3.2 ĐỆM NGUYÊN LIỆU

Trong phần này chúng ta để cập đến một số vấn đề hiệu quả, có liên quan đến việc đệm nguyên liệu. Trước tiên chúng ta nói đến *lược đồ đệm đôi* (two-buffer input scheme) rất có ích khi cần xem trước nguyên liệu để nhận diện các thẻ từ. Sau đó chúng ta sẽ giới thiệu một số kỹ thuật có ích làm tăng tốc độ của thể phân từ vựng, như sử dụng *khóa cầm canh* (sentinel) để đánh dấu vị trí kết thúc vùng đệm.

Có ba phương pháp cài đặt tổng quát cho thể phân từ vựng.

1. Sử dụng một chương trình để tạo ra thể phân từ vựng (bộ sinh thể phân từ vựng) từ phần đặc tả bằng biểu thức chính qui, chẳng hạn như trình biên dịch Lex sẽ được thảo luận trong Phần 3.5. Trong trường hợp này, chính bộ sinh thể phân từ vựng sẽ cung cấp các thủ tục để đọc và đệm nguyên liệu.
2. Viết một thể phân từ vựng bằng một ngôn ngữ lập trình hệ thống thông thường và sử dụng các tiện ích xuất nhập của ngôn ngữ đó để đọc nguyên liệu.
3. Viết một thể phân từ vựng bằng hợp ngữ và tự lo quản lý việc đọc nguyên liệu.

Ba lựa chọn này được liệt kê theo thứ tự tăng dần về mức độ khó khăn khi cài đặt. Không may là, các phương pháp khó cài đặt thường tạo ra các thể phân tích chạy nhanh hơn. Bởi vì thể phân từ vựng là giai đoạn biên dịch duy nhất có đọc chương trình nguồn từng ký tự một, có thể sẽ mất nhiều thời gian trong giai đoạn phân tích này, dù rằng các giai đoạn sau phức tạp hơn. Vì vậy, tốc độ của thể phân từ vựng là điều cần phải được xem xét khi thiết kế trình biên dịch. Mặc dù phần lớn của chương này dành cho cách tiếp cận thứ nhất, là thiết kế và sử dụng bộ sinh tự động, chúng ta cũng xem xét một số kỹ thuật rất có ích khi thiết kế thủ công. Phần 3.4 sẽ thảo luận về các *sơ đồ chuyển vị* (transition diagram), là một khái niệm có ích trong việc tổ chức một thể phân từ vựng được thiết kế thủ công.

Cặp vùng đệm

Đối với nhiều ngôn ngữ nguồn, có nhiều khi thể phân từ vựng phải đọc thêm một số ký tự trong nguyên liệu vượt quá từ tố cho một mẫu trước khi có thể thông báo rằng đã có một đối sánh xảy ra. Thể phân từ vựng trong Chương 2 dùng hàm `ungetc` để đẩy trả lại các ký tự sải với này cho dòng nguyên liệu. Vì có thể mất nhiều thời gian để di chuyển các ký tự, chúng ta cần dùng đến một kỹ thuật đệm đặc biệt nhằm giảm bớt chi phí cần để xử lý một ký tự nguyên liệu. Chúng ta có sẵn một số lược đồ đệm nhưng vì các kỹ thuật này thường hay phụ thuộc vào các tham số hệ thống, chúng ta chỉ phác thảo các nguyên tắc qua một lớp lược đồ sau.

Vùng đệm của chúng ta được chia thành hai nửa, mỗi nửa chứa được N ký tự như trong Hình 3.3. Thông thường N là số ký tự trên một khối đĩa, thí dụ là 1024 hoặc 4096.

Hình 3.3. Một vùng đệm nguyên liệu hai nửa.

Chúng ta đọc mỗi lần N ký tự vào mỗi nửa của vùng đệm bằng một *lệnh đọc* (read) của hệ thống chứ không phải kích hoạt lệnh đọc cho mỗi ký tự. Nếu trong nguyên liệu còn ít hơn N ký tự thì một ký tự đặc biệt **eof** được đọc vào sau các ký tự như trong Hình 3.3. Nghĩa là **eof** đánh dấu cuối tập tin nguồn và nó khác với mọi ký tự trong nguyên liệu.

Chúng ta cũng duy trì hai con trỏ chỉ đến vùng đệm. Chuỗi ký tự giữa hai con trỏ là từ tố hiện hành. Khởi đầu cả hai con trỏ đều chỉ về ký tự đầu tiên của từ tố tiếp theo được tìm thấy. Một con trỏ, được gọi là *con trỏ tới*, quét tới trước cho đến khi thấy một đối sánh với mẫu. Một khi đã xác định được từ tố kế tiếp, con trỏ tới được đặt trỏ tới ký tự ở đầu phải của từ tố. Sau khi đã xử lý từ tố, cả hai con trỏ đều được chỉ tới ký tự nằm ngay sau từ tố. Với lược đồ này, các dòng giải thích và khoảng trắng đều có thể được xử lý như các mẫu không sinh ra thẻ từ nào cả.

Khi con trỏ tới chuẩn bị vượt qua điểm giữa vùng đệm, nửa bên phải sẽ được làm đầy bằng N ký tự mới. Khi con trỏ tới chuẩn bị vượt qua đầu phải của vùng đệm, nửa trái sẽ được làm đầy bằng N ký tự mới và con trỏ tới sẽ được đưa trở lại vị trí bắt đầu của vùng đệm.

Lược đồ đệm này thường hoạt động rất tốt nhưng khi đó thì số lượng sải với bị

giới hạn, và sải với bị hạn chế này có thể làm cho nó không thể nhận diện được các thẻ từ trong những tình huống con trỏ tới phải vượt qua một khoảng cách lớn hơn chiều dài của vùng đệm. Thí dụ nếu chúng ta gặp

```
DECLARE ( ARG1, ARG2, . . . , AGRn )
```

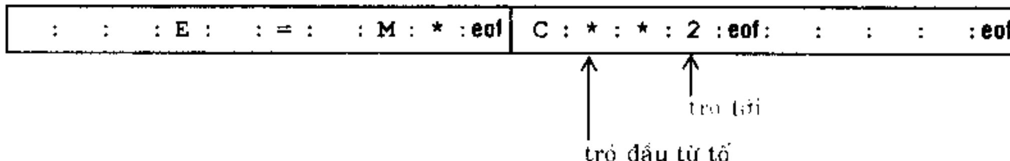
trong một chương trình PL/I, chúng ta không thể xác định được DECLARE là một từ khóa hay là tên mảng cho đến khi chúng nhìn thấy ký tự theo sau dấu ngoặc đóng. Trong mỗi trường hợp này, từ tố sẽ kết thúc tại ký tự E thứ hai, nhưng lượng sải với cần phải biết lại tỷ lệ với số lượng các đối mà về nguyên tắc thì không bị hạn chế.

if *forward* ở cuối nửa thứ nhất **then begin**
  đọc tiếp nguyên liệu vào nửa thứ hai;
  forward := *forward* + 1
end
else if *forward* ở cuối nửa thứ hai **then begin**
  đọc tiếp nguyên liệu vào nửa thứ nhất;
  di chuyển *forward* đến vị trí đầu tiên của nửa thứ nhất
end
else *forward* := *forward* + 1;

Hình 3.4. Đoạn mã để dịch con trỏ tới trước.

Khóa cầm canh

Nếu dùng lược đồ của Hình 3.3 giống như đã trình bày, mỗi khi di chuyển con trỏ tới, chúng ta phải kiểm tra rằng chúng ta đã không di chuyển qua khỏi mỗi nửa của vùng đệm; nếu có như thế thì chúng ta phải đọc tiếp nguyên liệu vào nửa kia. Nghĩa là đoạn chương trình dùng để di chuyển con trỏ tới trước phải kiểm tra giống như trong Hình 3.4.



Hình 3.5. Khóa cầm canh tại mỗi nửa vùng đệm.

Ngoại trừ ở các vị trí cuối của mỗi nửa, đoạn mã trong Hình 3.4 đòi hỏi phải có hai lần kiểm tra cho mỗi di chuyển của con trỏ tới. Chúng ta có thể giảm bớt hai lần kiểm tra này xuống còn một nếu chúng ta đặt một *khóa cầm canh* (sentinel) tại cuối mỗi

nửa. Khóa cầm canh là một ký tự đặc biệt không là thành phần của chương trình nguồn. Một chọn lựa tự nhiên là dùng ký tự **eof**; Hình 3.5 trình bày vùng đệm giống như Hình 3.3 nhưng có thêm khóa cầm canh.

Với cách bố trí như Hình 3.5, chúng ta có thể dùng đoạn mã của Hình 3.6 để di chuyển con trỏ tới (và kiểm tra vị trí cuối tập tin nguồn). Phần lớn thời gian chương trình chỉ thực hiện một kiểm tra để xem *forward* có trỏ tới một ký tự **eof** hay không. Chỉ khi chúng ta đi đến cuối của một nửa vùng đệm hoặc đến cuối tập tin chúng ta mới cần phải thực hiện các kiểm tra khác. Vì *N* ký tự nguyên liệu sẽ được gặp giữa các dấu **eof**, số lần kiểm tra tính trung bình cho mỗi ký tự nguyên liệu gần như là 1.

```
forward := forward + 1;
if forward↑ = eof then begin
    if forward ở cuối của nửa thứ nhất then begin
        đọc tiếp nguyên liệu vào nửa thứ hai;
        forward := forward + 1
    end
    else if forward ở cuối của nửa thứ hai then begin
        đọc tiếp nguyên liệu vào nửa thứ nhất;
        di chuyển forward đến vị trí đầu tiên của nửa thứ nhất
    end
    else /* eof nằm ở trong vùng đệm cho biết đã đến cuối nguyên liệu */
        kết thúc phân tích từ vựng
end
```

Hình 3.6. Đoạn mã có dùng khóa cầm canh.

Chúng ta cũng cần phải quyết định xem làm cách nào để xử lý ký tự đã được quét bởi con trỏ tới; nó đánh dấu cuối một thẻ từ, hoặc nó cho biết đang tìm kiếm một từ khóa hay điều gì nữa? Một cách để xây dựng các kiểm tra này là dùng câu lệnh **case** nếu ngôn ngữ cài đặt có câu lệnh này. Thế thì hành động kiểm tra

```
if forward↑ = eof
```

có thể được cài đặt như một nhánh trong **case**.

3.3 ĐẶC TẢ CÁC THẺ TỪ

Biểu thức chính qui là một ký pháp quan trọng để đặc tả các mẫu. Mỗi mẫu sẽ đối sánh được với một tập các chuỗi, vì thế có thể dùng biểu thức chính qui làm tên cho các tập chuỗi. Phần 3.5 sẽ mở rộng ký pháp này thành một ngôn ngữ dựa theo mẫu để phân tích từ vựng.

Chuỗi và Ngôn ngữ

Thuật ngữ *bộ chữ cái* (alphabet) hay *bộ ký tự* biểu thị một tập hữu hạn các ký hiệu. Các thí dụ điển hình cho các ký hiệu là các chữ cái và *ký tự* (character).[2] Tập {**0, 1**} là *bộ ký tự nhị phân* (binary alphabet). ASCII và EBCDIC là hai bộ chữ cái cho máy tính.

Một *chuỗi* (string) trên một bộ chữ cái là một dãy hữu hạn các ký hiệu được lấy từ bộ chữ cái đó. Trong lý thuyết ngôn ngữ, các thuật ngữ *câu* (sentence) và *từ* (word) thường được xem là đồng nghĩa với thuật ngữ "chuỗi". Chiều dài của một chuỗi s, thường ghi là $|s|$, là số lần xuất hiện của các ký hiệu trong s. Thí dụ **banana** có chiều dài 6. *Chuỗi rỗng* (empty string), được ký hiệu là ε, là một chuỗi đặc biệt có chiều dài 0. Một số thuật ngữ thông dụng đi kèm với chuỗi được tóm tắt trong Hình 3.7.

THUẬT NGỮ	ĐỊNH NGHĨA
Tiền tố của *s*	Một chuỗi thu được bằng cách loại bỏ zero hoặc nhiều ký hiệu ở phần sau của *s*; thí dụ **ban** là một tiền tố của **banana**.
Hậu tố của *s*	Một chuỗi được tạo ra bằng cách xóa zero hoặc nhiều ký hiệu ở phần trước của *s*; thí dụ **nana** là hậu tố của **banana**.
Chuỗi con của *s*	Một chuỗi thu được bằng cách xóa một tiền tố và một hậu tố ra khỏi *s*; thí dụ **nan** là một chuỗi con của **banana**. Mỗi tiền tố và mỗi hậu tố của *s* là một chuỗi con của *s*, nhưng không phải mọi chuỗi con của *s* đều là tiền tố hoặc hậu tố của *s*. Với mỗi chuỗi *s*, cả *s* và ε đều là tiền tố, hậu tố và chuỗi con của *s*.
Tiền tố, hậu tố hoặc *chuỗi con thực sự* của *s*	Một chuỗi không rỗng *x* tương ứng là tiền tố, hậu tố hoặc chuỗi con của *s* sao cho $s \neq x$.
Dãy con của *s*	Một chuỗi được tạo ra bằng cách xóa zero hoặc nhiều ký hiệu không nhất thiết là liên tục ra khỏi *s*; thí dụ **baaa** là dãy con của **banana**.

Hình 3.7. Các thuật ngữ cho các bộ phận của chuỗi.

Thuật ngữ *ngôn ngữ* (language) biểu thị một tập các chuỗi trên một bộ chữ cái cố định nào đó. Định nghĩa này rõ ràng là quá rộng. Các ngôn ngữ trừu tượng như $\varnothing$, là tập rỗng, hoặc $\{\varepsilon\}$, tập chỉ chứa một chuỗi rỗng, cũng là ngôn ngữ theo định nghĩa này. Tập các chương trình Pascal đúng cú pháp và tập tất cả các câu tiếng Anh đúng văn

[2] Các ký tự nói chung là các ký hiệu có thể nhập vào từ bàn phím (chữ cái, ký số và một số ký hiệu đặc biệt) và đôi khi cũng muốn nói đến các ký hiệu trong các bộ chữ cái của các ngôn ngữ tự nhiên. (ND)

phạm cũng là ngôn ngữ, mặc dù hai tập này rất khó mô tả. Cũng chú ý rằng định nghĩa này không gán bất kỳ một ý nghĩa nào cho các chuỗi trong một ngôn ngữ. Các phương pháp gán nghĩa cho các chuỗi được thảo luận trong Chương 5.

Nếu x và y là các chuỗi thì *ghép nối chuỗi* (concatenation) x và y, được ghi là xy, là chuỗi được tạo ra bằng cách gắn y vào sau x. Chẳng hạn nếu x = **dog** và y = **house** thì xy = **doghouse**. Chuỗi rỗng là phần tử trung hòa đối với phép ghép nối chuỗi. Nghĩa là $s\varepsilon = \varepsilon s = s$.

Nếu xem ghép nối chuỗi như một "*tích*" (product) thì chúng ta có thể định nghĩa chuỗi theo kiểu lũy thừa như sau. Định nghĩa s^0 là ε, và với $i > 0$, định nghĩa s^i là $s^{i-1}s$. Bởi vì εs là chính s, $s^1 = s$. Thế thì $s^2 = ss$, $s^3 = sss$, vân vân.

Các phép toán trên ngôn ngữ

Có rất nhiều phép toán quan trọng có thể áp dụng cho ngôn ngữ. Đối với phân tích từ vựng, chúng ta chủ yếu quan tâm đến *phép hợp* (union), *phép ghép nối chuỗi* (concatenation), và *bao đóng* (closure) như được định nghĩa trong Hình 3.8. Chúng ta cũng có thể tổng quát hóa toán tử "lấy lũy thừa" cho ngôn ngữ bằng cách định nghĩa L^0 là $\{\varepsilon\}$, và L^i là $L^{i-1}L$. Vì thế L^i là L được ghép với chính nó $i - 1$ lần.

Thí dụ 3.2. Gọi L là tập {A, B, . . . , Z, a, b, . . . , z} và D là tập {0, 1, . . . , 9}. Chúng ta có thể xem L và D bằng hai cách. L có thể được xem là bộ chữ cái chứa tập các chữ hoa và chữ thường còn D là tập chứa mười ký số. Một cách khác, vì một ký hiệu có thể được xem là một chuỗi có chiều dài là 1, các tập L và D đều là những ngôn ngữ hữu hạn. Dưới đây là một số thí dụ về các ngôn ngữ được tạo ra từ L và D bằng cách áp dụng các toán tử được định nghĩa trong Hình 3.8.

1. $L \cup D$ là tập các chữ cái và ký số.
2. LD là tập các chuỗi chứa một chữ cái và theo sau là một ký số.
3. L^4 là tập tất cả các chuỗi bốn chữ cái.
4. L^* là tập tất cả các chuỗi chữ cái, kể cả chuỗi rỗng ε.
5. $L(L \cup D)^*$ là tập tất cả các chuỗi chữ cái và ký số bắt đầu bằng một chữ cái.
6. D^+ là tập tất cả các chuỗi có một hoặc nhiều ký số. □

Biểu thức chính qui

Trong Pascal, một *định danh* (identifier) là một chữ cái và có zero hoặc nhiều chữ cái hoặc ký số theo sau; nghĩa là, một định danh là một phần tử của tập được định nghĩa bằng mục (5) của Thí dụ 3.2. Trong phần này, chúng ta sẽ trình bày một ký pháp có tên gọi là *biểu thức chính qui* (regular expression), cho phép chúng ta định nghĩa chính xác các tập như thế. Với ký pháp này, chúng ta có thể định nghĩa các định danh

của Pascal là

letter (letter | digit)*

Vạch đứng ở đây có nghĩa là "hoặc", các dấu ngoặc được dùng để nhóm các biểu thức con lại, dấu sao có nghĩa là "zero hoặc nhiều thể hiện của" biểu thức trong ngoặc, và viết **letter** cạnh phần còn lại của biểu thức có ý nghĩa là một ghép nối chuỗi.

PHÉP TOÁN	ĐỊNH NGHĨA
hợp của L và M được viết là $L \cup M$	$L \cup M = \{ s \mid s$ thuộc L hoặc s thuộc $M \}$
ghép nối của L và M được viết là LM	$LM = \{ st \mid s$ thuộc L và t thuộc $M \}$
bao đóng Kleene của L được viết là L^*	$L^* = \bigcup_{i=0}^{\infty} L^i$ L^* biểu thị "zero hoặc nhiều ghép nối của" L.
bao đóng dương của L được viết là L^+	$L^+ = \bigcup_{i=1}^{\infty} L^i$ L^+ biểu thị "một hoặc nhiều ghép nối của" L.

Hình 3.8. Định nghĩa các phép toán trên ngôn ngữ.

Một biểu thức chính qui được xây dựng từ những biểu thức đơn giản hơn bằng một tập qui tắc định nghĩa. Mỗi biểu thức chính qui r biểu thị cho một ngôn ngữ $L(r)$. Qui tắc định nghĩa sẽ mô tả cách thức tạo ra $L(r)$ bằng nhiều cách tổ hợp các ngôn ngữ được biểu thị bằng các biểu thức con của r.

Dưới đây là những qui tắc định nghĩa *các biểu thức chính qui trên bộ chữ cái* Σ. Kèm với mỗi qui tắc là một đặc tả của ngôn ngữ được biểu thị bởi biểu thức chính qui đang được định nghĩa.

1. ε là một biểu thức chính qui biểu thị cho $\{\varepsilon\}$, nghĩa là tập chứa chuỗi rỗng.
2. Nếu a là một ký hiệu trong Σ thì a là một biểu thức chính qui biểu thị cho $\{a\}$, nghĩa là tập chứa chuỗi a. Mặc dù chúng ta sử dụng cùng ký pháp cho cả ba loại, về lý thuyết, biểu thức chính qui a khác với chuỗi a hoặc ký hiệu a. Từ ngữ cảnh chúng ta sẽ nhận ra a là biểu thức chính qui, là chuỗi hay là ký hiệu.
3. Giả sử r và s là những biểu thức chính qui biểu thị cho các ngôn ngữ $L(r)$ và $L(s)$. Thế thì,

   a) $(r) \mid (s)$ là biểu thức chính qui biểu thị cho $L(r) \cup L(s)$.

   b) $(r)(s)$ là biểu thức chính qui biểu thị cho $L(r)L(s)$.

c) $(r)^*$ là biểu thức chính qui biểu thị cho $(L(r))^*$.

d) (r) là biểu thức chính qui biểu thị cho $L(r)$.[3]

Ngôn ngữ được biểu thị bằng một biểu thức chính qui gọi là *tập chính qui* (regular set).

Đặc tả của một biểu thức chính qui là một thí dụ về loại định nghĩa đệ qui. Qui tắc (1) và (2) tạo ra cơ sở của định nghĩa; chúng ta sử dụng thuật ngữ *ký hiệu cơ sở* để nói đến ε hoặc một ký hiệu thuộc tập Σ xuất hiện trong một biểu thức chính qui. Qui tắc (3) cung cấp bước qui nạp.

Chúng ta có thể tránh sử dụng các dấu ngoặc không cần thiết trong biểu thức chính qui nếu thừa nhận qui ước sau:

1. Toán tử đơn ngôi * có thứ bậc cao nhất và có tính kết hợp trái.
2. Toán tử ghép nối có thứ bậc thứ hai và có tính kết hợp trái.
3. | có thứ bậc thấp nhất và có tính kết hợp trái.

Theo qui ước này, $(a)|((b)^*(c))$ là tương đương với $a|b^*c$. Cả hai biểu thức đều biểu thị cho tập các chuỗi chỉ có a hoặc có zero hoặc nhiều b và theo sau là một c.

Thí dụ 3.3. Gọi $\Sigma = \{a, b\}$

1. Biểu thức chính qui $a|b$ biểu thị tập $\{a, b\}$.
2. Biểu thức chính qui $(a|b)(a|b)$ biểu thị tập $\{aa, ab, ba, bb\}$, là tập gồm tất cả các chuỗi a và b có chiều dài 2. Một biểu thức chính qui khác cho tập này là $aa \mid ab \mid ba \mid bb$.
3. Biểu thức chính qui a^* biểu thị tập tất cả các chuỗi có zero hoặc nhiều a, nghĩa là $\{\varepsilon, a, aa, aaa, \ldots\}$.
4. Biểu thức chính qui $(a|b)^*$ biểu thị tập tất cả các chuỗi có zero hoặc nhiều thể hiện của a hoặc b, nghĩa là tập các chuỗi a và b. Một biểu thức chính qui khác cho tập này là $(a^*b^*)^*$.
5. Biểu thức chính qui $a|a^*b$ biểu thị tập chứa chuỗi a và tất cả các chuỗi gồm zero hoặc nhiều a với một b theo sau. □

Nếu hai biểu thức chính qui r và s biểu thị cho cùng một ngôn ngữ, chúng ta nói r và s là *tương đương* và viết $r = s$. Thí dụ $(a|b) = (b|a)$.

Biểu thức chính qui cũng tuân theo một số luật đại số và như thế có thể dùng các luật này để biến đổi các biểu thức thành những dạng tương đương. Hình 3.9 trình bày một số luật đại số cho các biểu thức chính qui r, s và t.

[3] Qui tắc này nói rằng các cặp dấu ngoặc bổ sung có thể được đặt quanh các biểu thức chính qui nếu chúng ta muốn thế.

TIÊN ĐỀ	MÔ TẢ
$r \mid s = s \mid r$	$\mid$ có tính giao hoán
$r \mid (s \mid t) = (r \mid s) \mid t$	$\mid$ có tính kết hợp
$(rs)t = r(st)$	phép ghép nối có tính kết hợp
$r(s \mid t) = rs \mid rt$ $(s \mid t)r = sr \mid tr$	phép ghép nối phân phối trên $\mid$
$\varepsilon r = r$ $r\varepsilon = r$	ε là phần tử trung hòa đối với phép ghép nối
$r^* = (r \mid \varepsilon)^*$	quan hệ giữa * và ε
$r^{**} = r^*$	* có tính lũy đẳng

Hình 3.9. Các tính chất đại số của biểu thức chính qui.

Định nghĩa chính qui

Để thuận lợi về mặt ký pháp, chúng ta sẽ gán tên cho các biểu thức chính qui và định nghĩa các biểu thức chính qui bằng cách dùng tên này giống như chúng là những ký hiệu. Nếu Σ là bộ chữ cái chứa các ký hiệu cơ bản thì một *định nghĩa chính qui* (regular definition) là một dãy định nghĩa có dạng

$$d_1 \rightarrow r_1$$
$$d_2 \rightarrow r_2$$
$$\cdots$$
$$d_n \rightarrow r_n$$

với mỗi d_i là một tên riêng biệt, và mỗi r_i là một biểu thức chính qui trên các ký hiệu thuộc $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$, nghĩa là các ký hiệu cơ bản và các tên đã được định nghĩa trước đó. Bằng việc hạn chế mỗi r_i trong các ký hiệu của Σ và những tên đã định nghĩa trước đó, chúng ta có thể xây dựng một biểu thức chính qui trên Σ cho mọi r_i bằng cách thay thế lập đi lập lại tên biểu thức bằng các biểu thức được chúng biểu thị. Nếu r_i đã dùng d_j với $j \geq i$ thì r_i có thể được định nghĩa đệ qui, và quá trình sẽ không chấm dứt.

Để phân biệt tên với ký hiệu, tên sẽ được in đậm trong các định nghĩa chính qui.

Thí dụ 3.4. Như chúng ta đã nói, tập các định danh trong Pascal là tập các chuỗi chữ cái và ký số bắt đầu bằng một chữ cái. Dưới đây là định nghĩa chính qui cho tập này.

letter → A | B | . . . | Z | a | b | . . . | z
digit → 0 | 1 | . . . | 9
id → **letter** (**letter** | **digit**)* □

Thí dụ 3.5. Các số không dấu trong Pascal là những chuỗi như 5280, 39.37, 6.336E4 hoặc 1.984E-4. Định nghĩa chính qui sau đây đưa ra một đặc tả chính xác cho lớp các chuỗi này:

```
           digit  →  0 | 1 | . . . | 9
          digits  →  digit digit*
optional_fraction →  . digits | ε
optional_exponent →  ( E ( + | - | ε ) digits ) | ε
             num  →  digits optional_fraction optional_exponent
```

Định nghĩa này nói rằng một **optional_fraction** gồm một chấm thập phân có một hoặc nhiều ký số theo sau hoặc nó bị thiếu (chuỗi rỗng). Một **optional_exponent**, nếu có, là một E và sau đó là một dấu + hoặc - (tùy chọn) rồi sau đó có thể là một hoặc nhiều ký số. Chú ý rằng ít nhất phải có một ký số sau dấu chấm, vì thế **num** không đối sánh được với 1 nhưng đối sánh được với 1.0. □

Ký pháp viết tắt

Một số kết quả xuất hiện khá thường xuyên trong các biểu thức chính qui, do vậy để cho thuận tiện chúng ta đưa ra một số qui ước viết tắt cho chúng.

1. *Một hoặc nhiều thể hiện.* Toán tử hậu vị đơn ngôi $^+$ có nghĩa là "một hoặc nhiều thể hiện của". Nếu r là một biểu thức chính qui biểu thị cho ngôn ngữ $L(r)$ thì $(r)^+$ là một biểu thức chính qui biểu thị cho ngôn ngữ $(L(r))^+$. Vì thế biểu thức chính qui a^+ biểu thị tập tất cả các chuỗi có một hoặc nhiều a. Toán tử $^+$ có cùng thứ bậc và tính kết hợp như toán tử *. Hai đồng nhất thức đại số $r^* = r^+ \mid \varepsilon$ và $r^+ = rr^*$ được gọi tương ứng là các toán tử bao đóng Kleene và bao đóng dương.

2. *Zero hoặc một thể hiện.* Toán tử hậu vị đơn ngôi ? có nghĩa là "zero hoặc một thể hiện của." Ký pháp $r?$ là dạng tắt của $r \mid \varepsilon$. Nếu r là một biểu thức chính qui thì $(r)?$ là một biểu thức biểu thị cho ngôn ngữ $L(r) \cup \{\varepsilon\}$. Thí dụ, sử dụng các toán tử $^+$ và ?, chúng ta có thể viết lại định nghĩa chính qui cho **num** trong Thí dụ 3.5 là

```
           digit  →  0 | 1 | . . . | 9
          digits  →  digit+
optional_fraction →  ( . digits )?
optional_exponent →  ( E ( + | - )? digits)?
             num  →  digits optional_fraction optional_exponent
```

3. *Lớp ký tự* (character class). Ký pháp [abc] trong đó a, b và c là các ký hiệu trong bộ chữ cái biểu thị cho biểu thức chính qui a | b | c. Một lớp ký tự được viết tắt như [a-z] biểu thị cho biểu thức chính qui a | b | . . . | z. Sử dụng các lớp ký tự, chúng ta có thể mô tả các định danh như các chuỗi được tạo ra bởi biểu thức chính qui

```
[A-Za-z][A-Za-z0-9]*
```

Các tập không chính qui

Một số ngôn ngữ không mô tả được bằng biểu thức chính qui. Để minh họa khả năng mô tả hạn chế của các biểu thức chính qui, ở đây chúng tôi đưa ra một số thí dụ về các kết cấu ngôn ngữ lập trình không mô tả được bằng biểu thức chính qui. Chứng minh những phán đoán này có thể tìm thấy trong các tài liệu tham khảo.

Biểu thức chính qui không mô tả được các kết cấu cân hoặc kết cấu lồng. Thí dụ tập tất cả các chuỗi ngoặc cân không mô tả được bằng một biểu thức chính qui. Thế nhưng tập này có thể được đặc tả qua văn phạm phi ngữ cảnh.

Các chuỗi lặp cũng không mô tả được bằng biểu thức chính qui. Tập

$\{wcw \mid w$ là một chuỗi chứa các a và $b\}$

không biểu diễn được bằng một biểu thức chính qui và cũng không biểu diễn được bằng văn phạm phi ngữ cảnh.

Biểu thức chính qui chỉ biểu thị được một số cố định các lần lặp hoặc một số lần lặp không xác định của một kết cấu đã cho. Hai số lượng tùy ý không thể so sánh được để xem chúng có như nhau hay không. Vì thế chúng ta không thể mô tả được các chuỗi Hollerith có dạng $n\text{H}a_1a_2 \ldots a_n$ từ các phiên bản đầu của Fortran bằng một biểu thức chính qui vì số lượng các ký tự theo sau H phải khớp với số thập phân trước H.

3.4 NHẬN DẠNG CÁC THẺ TỪ

Trong phần trước chúng ta đã xét đến bài toán đặc tả các thẻ từ. Trong phần này, chúng ta tập trung cho câu hỏi làm thế nào để nhận ra chúng. Trong suốt phần này, chúng ta sẽ dùng ngôn ngữ được tạo ra bởi văn phạm dưới đây làm thí dụ minh họa.

Thí dụ 3.6. Xét đoạn văn phạm sau:

stmt	→	**if** *expr* **then** *stmt*
	|	**if** *expr* **then** *stmt* **else** *stmt*
	|	ε
expr	→	*term* **relop** *term*
	|	*term*
term	→	**id**
	|	**num**

trong đó các tận là **if**, **then**, **else**, **relop**, **id** và **num** sinh ra các tập chuỗi được cho bởi các định nghĩa chính qui sau:

if → `if`
then → `then`
else → `else`
relop → < | <= | = | <> | > | >=
id → **letter** (**letter** | **digit**)*
num → **digit**$^+$ (. **digit**$^+$)? (E(+ | −)? **digit**$^+$)?

trong đó **letter** và **digit** được định nghĩa như trước kia.

Với đoạn ngôn ngữ này, thể phân từ vựng sẽ nhận diện các từ khóa `if`, `then`, `else` cũng như các từ tố biểu thị bởi **relop**, **id** và **num**. Để cho vấn đề đơn giản, chúng ta giả thiết rằng các từ khóa được dành riêng, nghĩa là chúng không được dùng làm định danh. Giống như trong Thí dụ 3.5, **num** biểu thị cho số nguyên không dấu và số thực trong Pascal.

Ngoài ra chúng ta giả sử các từ tố được phân cách bởi khoảng trắng, đó là các chuỗi không rỗng chứa các ký hiệu blank, tab và newline. Thể phân từ vựng của chúng ta sẽ gỡ bỏ các khoảng trắng bằng cách so sánh một chuỗi với định nghĩa chính qui **ws** dưới đây

delim → **blank** | **tab** | **newline**
ws → **delim**$^+$

Nếu một đối sánh cho **ws** được tìm ra thì thể phân từ vựng không trả một thẻ từ nào về cho thể phân cú pháp. Đúng ra nó sẽ tiếp tục tìm một thẻ từ theo sau khoảng trắng và trả thẻ từ đó về cho thể phân cú pháp.

Mục đích của chúng ta là xây dựng một thể phân từ vựng có thể định vị được từ tố cho thẻ từ kế tiếp trong vùng đệm nguyên liệu và tạo ra thành phẩm là một cặp chứa thẻ từ thích hợp và giá trị thuộc tính của nó bằng cách dùng bảng dịch được cho trong Hình 3.10. Giá trị thuộc tính của các toán tử quan hệ được cho bởi các hằng tượng trưng LT, LE, EQ, NE, GT, GE.[4]

Sơ đồ chuyển vị

Được xem là một bước trung gian trong việc xây dựng một thể phân từ vựng, trước tiên chúng ta tạo ra một sơ đồ đặc biệt được gọi là *sơ đồ chuyển vị* (transition diagram). Sơ đồ chuyển vị mô tả các hành động xảy ra khi một thể phân từ vựng được gọi bởi thể phân cú pháp yêu cầu lấy thẻ từ kế tiếp, như được minh họa trong Hình 3.1. Giả sử vùng đệm nguyên liệu giống như Hình 3.3 và *con trỏ đầu từ tố* chỉ đến ký tự nằm sau từ tố cuối cùng được tìm thấy. Chúng ta dùng một sơ đồ chuyển vị để theo dõi thông

---

[4] Chúng là các chữ tắt của less than (<), less than or equal to (<=), equal to (=), not equal to (<>), greater than (>), greater than or equal to (>=). (ND)

tin về các ký tự đã gặp khi con trỏ tới quét qua nguyên liệu. Chúng ta thực hiện bằng cách di chuyển qua từng vị trí trong sơ đồ khi các ký tự được đọc.

BIỂU THỨC CHÍNH QUI	THẺ TỪ	GIÁ TRỊ THUỘC TÍNH
ws	–	–
if	**if**	–
then	**then**	–
else	**else**	–
id	**id**	con trỏ chỉ đến mục ghi trong bảng
num	**num**	con trỏ chỉ đến mục ghi trong bảng
<	**relop**	**LT**
<=	**relop**	**LE**
=	**relop**	**EQ**
<>	**relop**	**NE**
>	**relop**	**GT**
>=	**relop**	**GE**

Hình 3.10. Các mẫu biểu thức chính qui cho các thẻ từ.

Các vị trí trong sơ đồ chuyển vị là các vòng tròn và được gọi là *trạng thái* (state). Các trạng thái được nối lại với nhau bằng các mũi tên và được gọi là *cạnh* (edge). Các cạnh đi ra khỏi trạng thái *s* có các nhãn chỉ ra các ký tự nguyên liệu có thể xuất hiện tiếp theo sau khi sơ đồ chuyển vị đã đến được trạng thái *s*. Nhãn **other** muốn nói đến một ký tự không được bất kỳ cạnh nào trong số những cạnh đi khỏi *s* biểu thị.

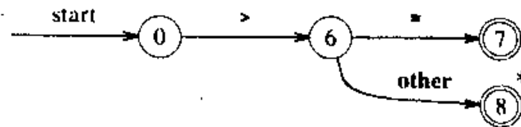
Chúng ta giả sử sơ đồ chuyển vị của phần này là *tất định* (deterministic);[5] nghĩa là không có ký hiệu nào khớp với nhãn của hai cạnh đi ra khỏi một trạng thái. Bắt đầu từ Phần 3.5, chúng ta sẽ giảm bớt điều kiện này, và tạo đơn giản hơn cho nhà thiết kế thể phân từ vựng và với những công cụ thích hợp thì không có khó khăn hơn cho người cài đặt.

Một trạng thái có nhãn *start*; đây là trạng thái khởi đầu của sơ đồ chuyển vị, là nơi giữ quyền điều khiển khi chúng ta bắt đầu nhận dạng thẻ từ. Một số trạng thái khác có thể có những hành động được thực hiện khi dòng điều khiển đến được trạng thái đó. Khi đi vào một trạng thái, chúng ta đọc ký tự kế tiếp. Nếu có một cạnh đi từ trạng thái hiện hành có nhãn so khớp được với ký tự này, chúng ta chuyển đến trạng thái được cạnh chỉ đến. Ngược lại chúng ta thất bại.

Hình 3.11 trình bày một sơ đồ chuyển vị cho các mẫu >= và >. Sơ đồ này hoạt động như sau. Trạng thái khởi đầu là 0. Trong trạng thái 0, chúng ta đọc ký tự tiếp

[5] Trong cuốn sách này, chúng tôi dùng hai thuật ngữ *tất định* và *đơn định* để dịch deterministic và hai thuật ngữ *không tất định*, *không đơn định* và *đa định* để dịch nondeterministic. (ND)

theo. Cạnh có nhãn > từ 0 được đi theo để đến trạng thái 6 nếu ký tự nguyên liệu là >. Ngược lại chúng ta thất bại không thể nhận diện > lẫn >=.



Hình 3.11. Sơ đồ chuyển vị cho >=.

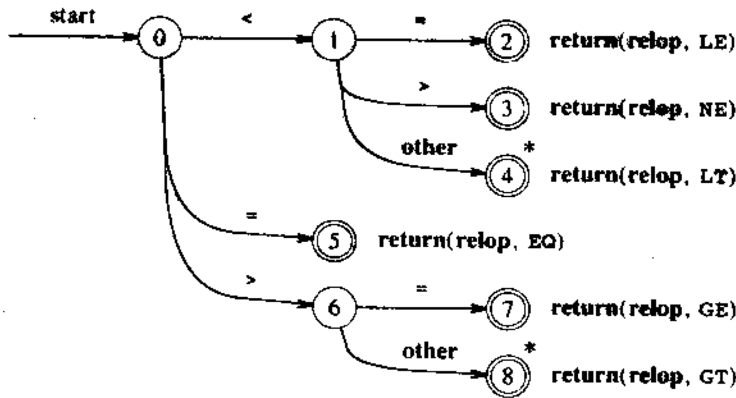
Khi đến được trạng thái 6, chúng ta đọc ký tự tiếp theo. Cạnh có nhãn = từ trạng thái 6 được đi theo để đến trạng thái 7 nếu ký tự này là =. Bằng không cạnh có nhãn **other** chỉ ra rằng chúng ta phải đi đến trạng thái 8. Vòng tròn kép ở trạng thái 7 chỉ ra rằng đây là *trạng thái kiểm nhận* (accepting state), nơi tìm ra thẻ từ >=.

Để ý rằng ký tự > và một ký tự khác nữa sẽ được đọc khi chúng ta đi theo loạt các cạnh từ trạng thái khởi đầu đến trạng thái kiểm nhận 8. Bởi vì ký tự dôi ra này không phải là thành phần của toán tử quan hệ >, chúng ta phải thu con trỏ tới trở lại một ký tự. Chúng ta dùng ký hiệu * để chỉ những trạng thái cần phải dịch lui con trỏ lại.

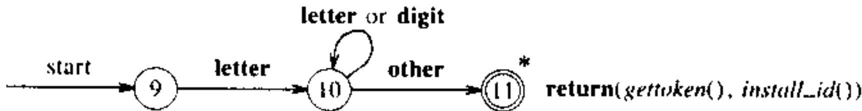
Nói chung thường có nhiều sơ đồ chuyển vị, mỗi sơ đồ đặc tả một nhóm thẻ từ. Nếu xảy ra thất bại khi chúng ta đang đi theo một sơ đồ chuyển vị thì chúng ta dịch lui con trỏ tới về nơi nó đã ở trong trạng thái khởi đầu của sơ đồ này rồi kích hoạt sơ đồ chuyển vị tiếp theo. Do con trỏ đầu từ tố và con trỏ tới đều chỉ đến cùng một vị trí trong trạng thái khởi đầu của sơ đồ, con trỏ tới sẽ được dịch lui lại để chỉ đến vị trí được con trỏ đầu từ tố chỉ tới. Nếu xảy ra thất bại trong tất cả mọi sơ đồ chuyển vị thì một lỗi từ vựng đã được phát hiện và chúng ta khởi động một thủ tục khắc phục lỗi.

Thí dụ 3.7. Một sơ đồ chuyển vị cho thẻ từ **relop** được trình bày trong Hình 3.12. Chú ý rằng Hình 3.11 là một phần của sơ đồ phức tạp này. □

Thí dụ 3.8. Bởi vì từ khóa là các dãy chữ cái, chúng là những ngoại lệ đối với qui tắc định danh là dãy chữ cái và ký số bắt đầu bằng một ký tự. Thay vì mã hóa các ngoại lệ này vào trong một sơ đồ chuyển vị, chúng ta dùng kỹ xảo là xử lý từ khóa như các định danh đặc biệt giống như trong Phần 2.7. Khi đến được trạng thái kiểm nhận trong Hình 3.13, chúng ta thực hiện một đoạn chương trình để xác định xem từ tố dẫn đến trạng thái kiểm nhận là một từ khóa hay là một định danh.



Hình 3.12. Sơ đồ chuyển vị cho các toán tử quan hệ.



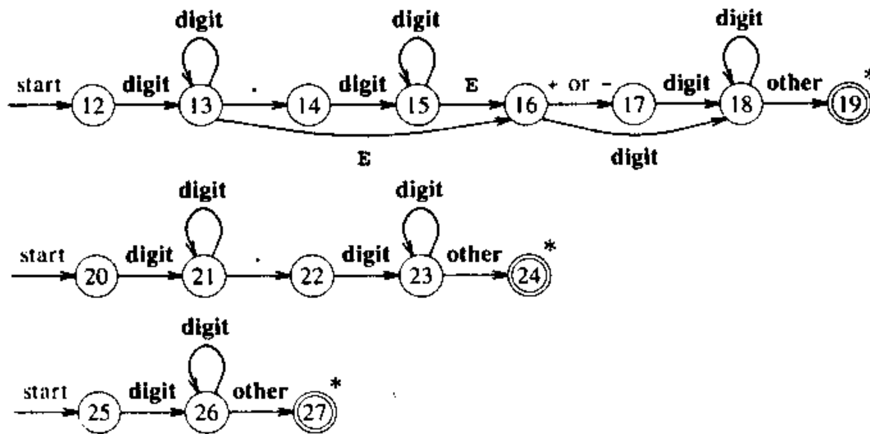
Hình 3.13. Sơ đồ chuyển vị cho định danh và từ khóa.

Một kỹ thuật đơn giản để tách từ khóa ra khỏi định danh là khởi gán bảng ký hiệu lưu thông tin về định danh một cách thích hợp. Đối với các thẻ từ của Hình 3.10 chúng ta cần nhập các chuỗi `if`, `then`, và `else` vào bảng ký hiệu trước khi đọc các ký tự trong nguyên liệu. Chúng ta cũng ghi chú trong bảng ký hiệu để trả về thẻ từ đó khi một trong những chuỗi này được nhận ra. Câu lệnh trả về (return) nằm cạnh trạng thái nhận trong Hình 3.13 sử dụng *gettoken*() và *install_id*() tương ứng để nhận thẻ từ và giá trị thuộc tính được trả về. Thủ tục *install_id*() có quyền truy xuất đến vùng đệm nơi từ tố cho định danh đã được định vị. Bảng ký hiệu sẽ được kiểm tra và nếu từ tố tìm thấy ở đó được ghi chú là từ khóa, *install_id*() trả về trị 0. Nếu từ tố được tìm ra và là một biến chương trình, *install_id*() trả về một con trỏ chỉ đến một mục ghi trong bảng ký hiệu. Nếu từ tố không tìm thấy trong bảng, nó được đưa vào làm một biến và một con trỏ chỉ đến mục ghi vừa được tạo ra sẽ được trả về.

Tương tự, thủ tục *gettoken*() tìm kiếm từ tố trong bảng ký hiệu. Nếu từ tố là một từ khóa, thẻ từ tương ứng được trả về; bằng không thẻ từ **id** được trả về.

Chú ý rằng sơ đồ chuyển vị không thay đổi nếu đưa thêm các từ khóa vào; chúng ta chỉ khởi gán bảng ký hiệu với các chuỗi và thẻ từ của các từ khóa mới này. □

Kỹ thuật đặt các từ khóa trong bảng ký hiệu rất cần thiết nếu thể phân từ vựng được xây dựng theo lối thủ công. Nếu không thực hiện như thế, số lượng các trạng thái trong một thể phân từ vựng cho một ngôn ngữ lập trình điển hình dễ lên đến cả trăm trạng thái, còn với kỹ xảo này thì có lẽ chỉ ngót nghét một trăm trạng thái.



Hình 3.14. Sơ đồ chuyển vị cho các số không dấu trong Pascal.

Thí dụ 3.9. Một số vấn đề sẽ nảy sinh khi chúng ta xây dựng thể nhận dạng cho các số không dấu được cho bởi định nghĩa chính qui

$$\textbf{num} \rightarrow \textbf{digit}^{+} \; (\; . \; \textbf{digit}^{+} \;) ? \; (\; \mathtt{E} \; (\; + \; | \; - \;) ? \; \textbf{digit}^{+}) ?$$

Để ý rằng định nghĩa này có dạng **digits fraction? exponent?** với **fraction** và **exponent** là tùy chọn.

Từ tố cho một thẻ từ đã cho phải là từ tố dài nhất có thể được. Thí dụ thể phân từ vựng không được dừng sau khi gặp 12 hoặc ngay cả 12.3 khi nguyên liệu là 12.3E4. Bắt đầu tại các trạng thái 25, 20 và 12 trong Hình 3.14, chúng ta sẽ đến được các trạng thái kiểm nhận sau khi gặp các số tương ứng là 12, 12.3, 12.3E4, miễn là sau 12.3E4 là một ký hiệu không phải ký số. Các sơ đồ chuyển vị với các trạng thái khởi đầu 25, 20 và 12 tương ứng dành cho **digits**, **digits fraction** và **digits fraction? exponent**, vì thế các trạng thái khởi đầu phải được thử theo thứ tự ngược lại là 12, 20, 25.

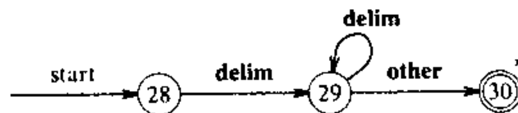
Hành động khi đến được một trong các trạng thái kiểm nhận 19, 24 hoặc 27 là gọi thủ tục *install_num* để nhập từ tố vào trong bảng các số và trả một con trỏ chỉ đến *mục ghi vừa được tạo ra. Thể phân từ vựng trả thẻ từ* ***num*** *cùng với con trỏ này làm* giá trị từ vựng. □

Thông tin về ngôn ngữ không nằm trong các định nghĩa chính qui của thẻ từ có thể được dùng để tìm lỗi trong nguyên liệu. Thí dụ với nguyên liệu **1.<x**, chúng ta thất bại trong các trạng thái 14 và 22 trong Hình 3.14 với ký tự tiếp theo là <. Thay vì trả về con số 1, chúng ta có thể muốn ghi nhận một lỗi và tiếp tục tiến hành, xem như nguyên liệu là **1.0<x**. Những hiểu biết như thế cũng có thể được dùng để đơn giản hóa các sơ đồ chuyển vị bởi vì xử lý lỗi có thể được dùng để tái hoạt lại từ một số tình huống mà nếu không thì sẽ dẫn đến thất bại.

Có nhiều cách để tránh các đối sánh dư thừa trong các sơ đồ chuyển vị của Hình 3.14. Một cách là viết lại các sơ đồ chuyển vị bằng cách tổ hợp chúng thành một, một công việc nói chung không phải là tầm thường. Một cách khác là thay đổi lối đáp ứng với thất bại trong quá trình đi qua một sơ đồ. Một phương pháp sẽ được phân tích trong chương này sẽ cho phép chúng ta vượt qua nhiều trạng thái kiểm nhận; chúng ta quay trở lại trạng thái kiểm nhận cuối cùng đã đi qua khi thất bại xảy ra.

Thí dụ 3.10. Một dãy sơ đồ chuyển vị cho tất cả các thẻ từ của Thí dụ 3.6 sẽ có được nếu chúng ta kết hợp các sơ đồ của các Hình 3.12, 3.13 và 3.14. Các trạng thái được đánh số thấp phải được thử trước các trạng thái được đánh số cao.

Vấn đề duy nhất còn lại là khoảng trắng. Việc xử lý **ws**, biểu thị cho các *khoảng trắng* (white space), có khác so với việc xử lý các mẫu đã được thảo luận ở trên bởi vì không có gì để trả về cho thể phân cú pháp khi tìm thấy các khoảng trắng trong nguyên liệu. Một sơ đồ dịch tự nhận dạng **ws** là



Chúng ta không trả gì về khi đạt đến trạng thái nhận; chúng ta chỉ đơn giản trở lại trạng thái khởi đầu của sơ đồ chuyển vị đầu tiên để tìm một mẫu khác.

Mỗi khi có thể, tốt hơn chúng ta nên tìm những thẻ từ thường gặp trước khi tìm những thẻ từ ít gặp vì chúng ta chỉ đi đến một sơ đồ chuyển vị sau khi thất bại trên các sơ đồ trước đó. Vì khoảng trắng có lẽ sẽ rất thường gặp, việc đặt sơ đồ cho khoảng trắng gần đầu sẽ tốt hơn là kiểm tra khoảng trắng vào lúc cuối. □

Cài đặt sơ đồ chuyển vị

Dãy các sơ đồ chuyển vị có thể được chuyển thành một chương trình tìm kiếm thẻ từ được đặc tả bằng các sơ đồ. Chúng ta chấp nhận một phương pháp tiếp cận có hệ

thống để có thể hoạt động được trên tất các sơ đồ chuyển vị và xây dựng các chương trình với kích thước tỷ lệ với số trạng thái và cạnh trong sơ đồ.

Mỗi trạng thái tương ứng với một đoạn mã. Nếu có các cạnh đi ra khỏi một trạng thái thì đoạn mã của nó đọc một ký tự và chọn một cạnh để đi tiếp nếu có thể. Hàm **nextchar()** được dùng để đọc ký tự tiếp theo từ vùng đệm nguyên liệu, di chuyển con trỏ tới tại mỗi lời gọi và trả về ký tự được đọc.[6] Nếu có một cạnh với nhãn là ký tự được đọc, hoặc là lớp ký tự chứa ký tự được đọc thì quyền điều khiển sẽ được trao cho đoạn mã của trạng thái được chỉ đến bởi cạnh đó. Nếu không có một cạnh như thế, và trạng thái hiện hành không phải là trạng thái cho biết đã tìm ra một thẻ từ thì thủ tục **fail()** được kích hoạt để dịch lui con trỏ tới về vị trí của con trỏ đầu và khởi hoạt tìm kiếm thẻ từ được đặc tả bằng sơ đồ chuyển vị kế tiếp. Nếu không có sơ đồ nào khác để thử, **fail()** sẽ gọi một thủ tục khắc phục lỗi.

Để trả về các thẻ từ, chúng ta dùng một biến toàn cục **lexical_value**. Nó được gán cho các con trỏ được các hàm **install_id()** và **install_num()** trả về, tương ứng khi tìm ra một định danh hoặc một số. Lớp thẻ từ được trả về bởi thủ tục chính của thể phân từ vựng có tên là **nexttoken()**.

```
int  state = 0, start = 0;
int  lexical_value; /* để “trả về” thành phần thứ hai của thẻ từ */

int  fail()
{
  forward = token_beginning;
  switch (start) {
    case 0:  start = 9; break;
    case 9:  start = 12; break;
    case 12: start = 20; break;
    case 20: start = 25; break;
    case 25: recover(); break;
    default:/* lỗi trình biên dịch */
  }
  return start;
}
```

Hình 3.15. Đoạn chương trình C tìm trạng thái khởi đầu kế tiếp.

[6] Một cài đặt hiệu quả hơn sẽ dùng một macro in-line ở vị trí của hàm **nextchar()**.

```
token nexttoken()
{ while(1) {
      switch (state)  {
      case 0: c = nextchar();  /* c là ký hiệu sải với */
        if (c == blank || c == tab || c == newline) {
          state = 0;
          lexeme_beginning++;    /* dịch con trỏ đến đầu từ tố */
        }
        else if (c == '<') state = 1;
        else if (c == '=') state = 5;
        else if (c == '>') state = 6;
        else state = fail(); break;
        . . . /* các trường hợp 1-8 ở đây */

      case 9: c = nextchar();
        if (isletter(c)) state = 10;
        else state = fail(); break;
      case 10: c = nextchar();
        if (isletter(c)) state = 10;
        else if (isdigit(c)) state = 10;
        else state = 11; break;
      case 11: retract(1); install_id();
        return (gettoken());
        . . . /* các trường hợp 12-24 ở đây */

      case 25: c = nextchar();
        if (isdigit(c)) state = 26;
        else state = fail(); break;
      case 26: c = nextchar();
        if (isdigit(c)) state = 26;
        else state = 27; break;
      case 27: retract(1); install_num();
        return (NUM);
      }
  }
}
```

Hình 3.16. Chương trình C cho thể phân từ vựng.

Chúng ta dùng câu lệnh **case** để tìm trạng thái khởi đầu của sơ đồ chuyển vị kế tiếp. Trong bản cài đặt C ở Hình 3.15, hai biến **state** và **start** dùng để theo dõi trạng thái hiện tại và trạng thái khởi đầu của sơ đồ chuyển vị hiện hành. Chỉ số của

trạng thái trong đoạn mã dựa theo các sơ đồ chuyển vị của các Hình 3.12 đến 3.14.

Các cạnh trong sơ đồ chuyển vị được thể hiện bằng cách chọn lập đi lập lại đoạn mã cho một trạng thái và thực hiện nó để xác định trạng thái kế tiếp như trong Hình 3.16. Chúng tôi trình bày đoạn mã cho trạng thái 0, theo như đã sửa đổi trong Thí dụ 3.10 để xử lý khoảng trắng, và đoạn mã cho hai sơ đồ chuyển vị từ Hình 3.13 và Hình 3.14. Chú ý rằng kết cấu của C

`while` (1) *stmt*

lập lại *stmt* vô hạn, nghĩa là cho đến khi gặp lệnh `return`.

Bởi vì C không cho phép trả về đồng thời cả thẻ từ lẫn giá trị thuộc tính của nó, `install_id()` và `install_num()` đã đặt giá trị thuộc tính vào một biến toàn cục một cách hợp lý, tương ứng với một mục ghi trong bảng cho **id** hoặc **num** đang xét.

Nếu ngôn ngữ cài đặt không có câu lệnh `case`, chúng ta có thể tạo ra một mảng cho mỗi trạng thái, được chỉ mục bằng các ký tự. Nếu *state*1 là một mảng như thế thì *state*1[*c*] là một con trỏ chỉ đến phần chương trình phải được thực hiện khi ký tự sải với là *c*. Phần chương trình này thường chấm dứt bằng lệnh `goto` nhảy đến phần chương trình cho trạng thái kế tiếp. Mảng cho trạng thái *s* được tham chiếu đến như bảng trung chuyển cho *s*.

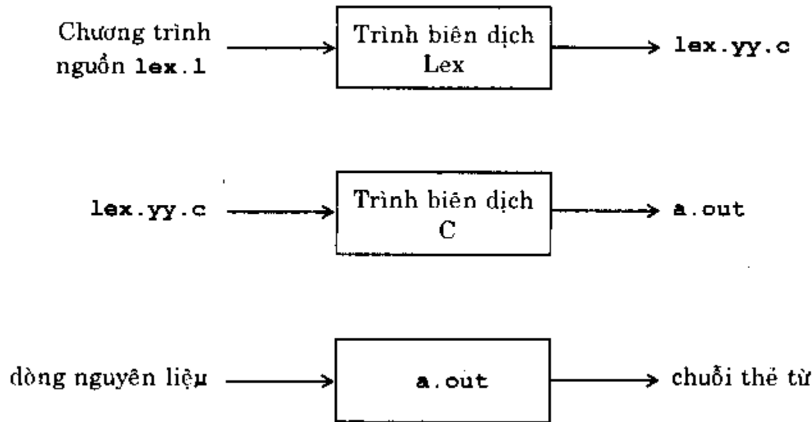
3.5 MỘT NGÔN NGỮ ĐẶC TẢ THỂ PHÂN TỪ VỰNG

Nhiều công cụ đã được thiết kế để tạo ra thể phân từ vựng từ những ký pháp đặc dụng dựa trên biểu thức chính qui. Chúng ta đã thấy việc sử dụng biểu thức chính qui để đặc tả các mẫu thẻ từ. Trước khi xét các thuật toán biên dịch biểu thức chính qui thành các chương trình đối sánh mẫu, chúng ta đưa ra một thí dụ về một công cụ có thể dùng một thuật toán như thế.

Trong phần này, chúng ta mô tả một công cụ đặc biệt có tên là Lex, đã được dùng rộng rãi để đặc tả các thể phân từ vựng cho rất nhiều ngôn ngữ. Chúng ta gọi công cụ này là *trình biên dịch Lex* (Lex compiler) và gọi đặc tả nguyên liệu của nó là *ngôn ngữ Lex* (Lex language). Thảo luận về một công cụ có sẵn cho phép chúng ta trình bày cách tổ hợp phần đặc tả mẫu bằng biểu thức chính qui với các hành động, thí dụ hành động tạo các mục ghi trong một bảng ký hiệu, một hành động mà thể phân từ vựng có thể được yêu cầu thực hiện. Các đặc tả tựa-Lex có thể được dùng ngay cả khi không có sẵn một trình biên dịch Lex; những đặc tả này có thể được chuyển thể thủ công thành một chương trình với các kỹ thuật sơ đồ chuyển vị của phần trước.

Lex nói chung được dùng theo cách thức được mô tả trong Hình 3.17. Trước tiên một đặc tả cho một thể phân từ vựng được chuẩn bị bằng cách tạo ra một chương trình `lex.l` bằng ngôn ngữ Lex. Sau đó `lex.l` được cho "chạy" qua một trình biên dịch Lex,

sinh ra một chương trình C `lex.yy.c`. Chương trình `lex.yy.c` chứa một biểu diễn dạng bảng của một sơ đồ chuyển vị được xây dựng từ các biểu thức chính qui của `lex.l` cùng với những thủ tục chuẩn có dùng bảng này để nhận dạng các từ tố. Các hành động đi kèm với biểu thức chính qui trong `lex.l` là những đoạn chương trình C và được mang trực tiếp vào `lex.yy.c`. Cuối cùng `lex.yy.c` được cho chạy qua một trình biên dịch C, sinh ra một chương trình đối tượng `a.out`, đó là thể phân từ vựng biến đổi dòng nguyên liệu thành một chuỗi thẻ từ.



Hình 3.17. Tạo ra một thể phân từ vựng bằng Lex.

Đặc tả Lex

Một chương trình Lex gồm có ba phần:

```
phần khai báo
%%
các qui tắc dịch
%%
các thủ tục phụ trợ
```

Phần khai báo chứa các *khai báo* (declaration) cho biến, các *hằng đại diện* (manifest constant) và định nghĩa chính qui. (Hằng đại diện là định danh được khai báo biểu thị cho một hằng). Định nghĩa chính qui là những câu lệnh tương tự như trong Phần 3.3 và được dùng làm thành phần của các biểu thức chính qui có trong các qui tắc dịch.

Các qui tắc dịch của một chương trình Lex là những câu lệnh có dạng

p_1 $\{action_1\}$
p_2 $\{action_2\}$
... ...
p_n $\{action_n\}$

trong đó mỗi p_i là một biểu thức chính qui và mỗi $action_i$ là một đoạn chương trình mô tả hành động mà thể phân từ vựng cần thực hiện khi mẫu p_i đối sánh được với một từ tố. Trong Lex, các hành động được viết bằng C; tuy nhiên nói chung chúng có thể được viết bằng một ngôn ngữ bất kỳ.

Phần thứ ba chứa tất cả mọi thủ tục phụ trợ cần thiết cho các hành động. Một cách chọn lựa khác là biên dịch riêng rẽ những thủ tục này và tải vào cùng với thể phân từ vựng.

Một thể phân từ vựng được tạo ra bởi Lex hoạt động hiệp đồng với thể phân cú pháp theo phương cách sau đây. Khi được kích hoạt bởi thể phân cú pháp, thể phân từ vựng bắt đầu đọc phần nguyên liệu còn lại, mỗi lần một ký tự cho đến khi nó tìm thấy tiền tố dài nhất của nguyên liệu đối sánh được với một trong những biểu thức chính qui p_i. Sau đó nó thực hiện $action_i$. Điển hình, $action_i$ sẽ trả quyền điều khiển về cho thể phân cú pháp. Tuy nhiên nếu không, thể phân từ vựng tiếp tục tìm thêm các từ tố cho đến khi có một hành động khiến quyền điều khiển được trao lại cho thể phân cú pháp. Hành động tìm kiếm được lặp lại cho các từ tố cho đến khi một lệnh trở về tường minh cho phép thể phân từ vựng xử lý khoảng trắng và các lời giải thích một cách thuận tiện.

Thể phân từ vựng sẽ trả về một đại lượng duy nhất là thẻ từ cho thể phân cú pháp. Để chuyển giá trị thuộc tính chứa thông tin về từ tố, chúng ta có thể dùng biến toàn cục `yylval`.

Thí dụ 3.11. Hình 3.18 là một chương trình Lex nhận dạng thẻ từ của Hình 3.10 và trả về thẻ từ được tìm thấy. Một vài nhận xét về đoạn mã sẽ cho chúng ta thấy được nhiều đặc tính quan trọng của Lex.

Trong phần khai báo, chúng ta thấy (một vị trí cho) khai báo các hằng đại diện được dùng bởi các qui tắc dịch.[7] Những khai báo này được bao quanh bởi các dấu ngoặc đặc biệt `%{` và `%}`. Những gì xuất hiện giữa các dấu ngoặc này được sao chép trực tiếp vào thể phân từ vựng **`lex.yy.c`** và không được xử lý như thành phần của các định nghĩa chính qui hoặc các qui tắc dịch. Cách xử lý giống y như thế cũng được dành cho các thủ tục phụ trợ trong phần thứ ba. Trong Hình 3.18 có hai thủ tục, `install_id` và `install_num` được các qui tắc dịch sử dụng; các thủ tục này sẽ được sao chép nguyên văn vào `lex.yy.c`.

[7] Chương trình `lex.yy.c` thường được dùng như một thủ tục con của thể phân cú pháp sinh ra bởi chương trình Yacc, là một bộ sinh thể phân cú pháp sẽ được mô tả trong Chương 4. Trong trường hợp này, khai báo của các hằng đại diện sẽ được thể phân cú pháp cung cấp khi nó được biên dịch cùng với chương trình `lex.yy.c`.

```
%{
     /* định nghĩa của các hằng đại diện
        LT, LE, EQ, NE, GT, GE,
        IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* định nghĩa chính qui */
delim      [ \t\n]
ws         {delim}+
letter     [A-Za-z]
digit      [0-9]
id         {letter}({letter}|{digit})*
number     {digit}+(\.{digit}+)?(E[+\-]?{digit}+)?

%%

{ws}       {/* không có hành động nào và không có lệnh trở về */}
if         {return(IF);}
then       {return(THEN);}
else       {return(ELSE);}
{id}       {yylval = install_id(); return(ID);}
{number}   {yylval = install_num(); return(NUMBER);}
"<"        {yylval = LT; return(RELOP);}
"<="       {yylval = LE; return(RELOP);}
"="        {yylval = EQ; return(RELOP);}
"<>"       {yylval = NE; return(RELOP);}
">"        {yylval = GT; return(RELOP);}
">="       {yylval = GE; return(RELOP);}

%%

install_id() {
   /* thủ tục đặt từ tố vào bảng ký hiệu và trả về
   một con trỏ chỉ đến đó. Ký tự đầu tiên của từ tố được trỏ đến
   bởi yytext và chiều dài của nó là yyleng */
}

install_num() {

   /* một thủ tục tương tự khi từ tố là một số */

}
```

Hình 3.18. Chương trình Lex cho các thẻ từ của Hình 3.10.

Phần định nghĩa cũng chứa một số định nghĩa chính qui. Mỗi định nghĩa như thế gồm có một tên và một biểu thức chính qui được biểu thị bởi tên đó. Thí dụ tên đầu tiên được định nghĩa là **delim**; nó đại diện cho lớp ký tự `[ \t\n]`, nghĩa là một trong ba ký hiệu blank, tab (biểu thị bởi `\t`) hoặc newline (biểu thị bởi `\n`). Định nghĩa thứ hai là của khoảng trắng, được biểu thị bởi tên **ws**. Khoảng trắng là một chuỗi gồm một hoặc nhiều *ký tự phân cách* (delimiter) đã được định nghĩa bằng **delim**. Chú ý rằng trong Lex, từ **delim** phải được bao quanh bởi các dấu ngoặc để phân biệt nó với một mẫu chứa năm chữ cái **delim**.

Trong định nghĩa của **letter**, chúng ta thấy cách sử dụng một lớp ký tự. Cách ghi tắt **[A-Za-z]** có nghĩa là một trong những chữ hoa từ **A** đến **Z** hoặc chữ thường từ **a** đến **z**. Định nghĩa thứ năm, là **id**, sử dụng các dấu ngoặc tròn. Nó thuộc loại *meta ký hiệu* (metasymbol)[8] trong ngôn ngữ Lex mà ý nghĩa tự nhiên là một *tác tử nhóm* (grouper). Tương tự, vạch đứng cũng là một meta ký hiệu biểu diễn phép hợp.

Định nghĩa chính qui cuối cùng dành cho **number**, ở đó chúng ta nhận xét thêm một số điểm nữa. Chúng ta thấy cách dùng **?** làm meta ký hiệu với ý nghĩa thông thường là "zero hoặc nhiều xuất hiện của". Chúng ta cũng chú ý dấu gạch ngược \ được dùng như một điểm thoát, cho phép một ký tự là meta ký hiệu của Lex mang được ý nghĩa tự nhiên của nó. Đặc biệt dấu chấm thập phân trong định nghĩa của **number** được biểu diễn bằng `\.` vì một chấm bản thân nó đã biểu diễn cho một lớp ký tự gồm tất cả các ký tự trừ ký tự newline trong Lex cũng như trong nhiều chương trình hệ thống UNIX có dùng các biểu thức chính qui. Trong lớp ký tự `[+\-]`, chúng ta đặt một dấu gạch ngược trước dấu trừ bởi vì dấu trừ khi đại diện cho chính nó có thể bị nhầm với việc sử dụng nó để biểu thị một khoảng thay đổi như `[A-Z]`.[9]

Có một cách khác làm cho các ký tự mang nghĩa tự nhiên của chúng, ngay cả nếu chúng là các meta ký hiệu trong Lex: bao quanh chúng bằng dấu nháy kép. Chúng ta đã trình bày một thí dụ về qui ước này trong phần các qui tắc dịch, ở đó sáu toán tử quan hệ được bao quanh bởi các dấu nháy kép.[10]

Bây giờ chúng ta hãy xét qui tắc dịch trong phần đi sau dấu **%%** đầu tiên. Qui tắc đầu nói rằng nếu chúng ta gặp **ws**, nghĩa là một chuỗi các ký tự blank, tab và newline

---

[8] Meta- là một tiếp đầu ngữ (gốc Hy lạp) biểu thị một khái niệm khái quát hơn hoặc một ngành nghiên cứu tổng quát hơn, thường được dịch là "siêu" hoặc để nguyên là meta. Chúng tôi chọn cách sau để khỏi nhầm lẫn với hai tiếp đầu ngữ thường được dịch là "siêu", đó là hyper- và super- mà đương nhiên ý nghĩa khác với tiếp đầu ngữ meta-. (ND)

[9] Thực sự Lex xử lý chính xác lớp ký tự `[+-]` mà không cần dùng dấu gạch ngược bởi vì dấu trừ đứng cuối không thể biểu thị cho một khoảng thay đổi.

[10] Chúng ta làm như thế vì < và > là các meta ký hiệu của Lex; chúng bao quanh tên các "trạng thái", cho phép Lex thay đổi trạng thái khi gặp một số thẻ từ, chẳng hạn các lời chú giải hoặc chuỗi ký tự được đóng ngoặc kép mà chúng phải được xử lý khác hẳn so với các đoạn văn bản thông thường. Chúng ta không cần phải bao quanh dấu bằng nhưng điều đó cũng không là bắt buộc.

dài nhất thì không làm gì cả. Cụ thể chúng ta không trở về thể phân cú pháp. Cần nhớ rằng cấu trúc của thể phân từ vựng khiến nó luôn cố gắng nhận dạng các thẻ từ cho đến khi hành động đi kèm với một thẻ từ được tìm ra khiến nó thực hiện câu lệnh trở về.

Qui tắc thứ hai nói rằng nếu gặp các chữ cái `if` thì trả về thẻ từ `IF`, là một hằng đại diện biểu diễn một số nguyên được thể phân cú pháp hiểu ngầm là thẻ từ **if**. Tương tự, hai qui tắc tiếp theo xử lý các từ khóa `then` và `else`.

Trong qui tắc cho **id**, chúng ta thấy hai câu lệnh trong hành động đi kèm. Trước tiên, biến `yylval` được đặt là giá trị được thủ tục `install_id` trả về; định nghĩa của thủ tục đó nằm trong phần thứ ba. `yylval` là một biến mà định nghĩa của nó xuất hiện trong tập thành phẩm `lex.yy.c` và thể phân cú pháp có thể sử dụng. Mục đích là cho `yylval` giữ giá trị từ vựng được trả về bởi vì câu lệnh thứ hai của hành động, là `return(ID)`, chỉ có thể trả về mã biểu thị cho lớp thẻ từ.

Chúng ta không trình bày chi tiết đoạn mã chương trình của `install_id`. Tuy nhiên chúng ta có thể giả thiết rằng nó tìm trong bảng ký hiệu một từ tố đối sánh được với mẫu **id**. Lex chuẩn bị sẵn từ tố này cho các thủ tục xuất hiện trong phần thứ ba qua hai biến `yytext` và `yyleng`. Biến `yytext` tương ứng với biến đã được gọi là *lexeme_beginning*, nghĩa là một con trỏ chỉ đến ký tự đầu tiên của từ tố; `yyleng` là một số nguyên cho biết chiều dài của từ tố. Thí dụ nếu `install_id` không tìm thấy một định danh trong bảng ký hiệu, nó có thể tạo ra một mục ghi mới cho định danh đó. `yyleng` ký tự của nguyên liệu, bắt đầu từ `yytext`, có thể được sao chép vào một mảng ký tự và được phân cách bởi một dấu end-of-string (kết thúc chuỗi) giống như trong Phần 2.7. Mục ghi mới có thể chỉ đến nơi bắt đầu của bản sao này.

Các số được xử lý tương tự bởi qui tắc tiếp theo, và với sáu qui tắc cuối cùng, `yylval` được dùng để trả về một mã cho mỗi toán tử quan hệ được tìm thấy, còn giá trị trả về thực sự sẽ là mã cho thẻ từ **relop** trong mỗi trường hợp.

Giả sử thể phân từ vựng có nguồn gốc từ chương trình của Hình 3.18 được cho một nguyên liệu chứa hai ký hiệu tab, các chữ cái `if` và một ký hiệu blank. Hai ký tự tab là tiền tố khởi đầu dài nhất của nguyên liệu khớp được với mẫu **ws**. Hành động của **ws** là không làm gì cả, vì thế thể phân từ vựng di chuyển con trỏ *lexeme-beginning* là `yytext` đến `i` và bắt đầu tìm một thẻ từ khác.

Từ tố tiếp theo đối sánh được là `if`. Để ý rằng các mẫu `if` và `{id}` đều đối sánh được với từ tố này và không có mẫu nào đối sánh được với một chuỗi dài hơn. Vì mẫu cho từ khóa `if` đi trước mẫu cho các định danh trong danh sách của Hình 3.18, xung đột được giải quyết theo từ khóa này. Nói chung, *chiến lược giải quyết tính đa nghĩa* (ambiguity-resolving strategy) sẽ dễ dàng khi dành riêng các từ khóa bằng cách liệt kê chúng trước các mẫu cho định danh.

Một thí dụ khác, giả sử **<=** là hai ký tự đầu tiên được đọc. Mặc dù mẫu **<** đối sánh

được với ký tự thứ nhất, nó không phải là mẫu dài nhất đối sánh được với một tiền tố của nguyên liệu. Vì thế chiến lược của Lex trong việc chọn tiền tố dài nhất đối sánh được với một mẫu tạo dễ dàng cho việc giải quyết xung đột giữa < và <= bằng một phương pháp được mong đợi — đó là chọn <= làm thẻ từ tiếp theo. □

Toán tử sải với

Như chúng ta đã thấy trong Phần 3.1, thể phân từ vựng của một số kết cấu của ngôn ngữ lập trình cần phải "xem trước" một số ký tự vượt quá điểm kết thúc của một từ tố trước khi có thể xác định chắc chắn một thẻ từ. Chúng ta nhớ lại thí dụ về Fortran với cặp lệnh

```
DO 5 I = 1.25
DO 5 I = 1,25
```

Trong Fortran, các *khoảng trống* (blank) không có tác dụng gì bên ngoài các phần giải thích và các chuỗi Hollerith, do vậy giả sử rằng mọi khoảng trống có thể loại được đều đã được lược bỏ trước khi thể phân từ vựng bắt đầu hoạt động. Vì thế khi chuyển cho thể phân từ vựng, các câu lệnh trên sẽ trở thành

```
DO5I=1.25
DO5I=1,25
```

Trong câu lệnh thứ nhất, chúng ta không thể nói được gì cho đến khi gặp dấu chấm thập phân, cho biết rằng chuỗi DO là thành phần của định danh DO5I. Trong câu lệnh thứ hai, bản thân DO là một từ khóa.

Trong Lex, chúng ta có thể viết một mẫu dưới dạng r_1/r_2, trong đó r_1 và r_2 là các biểu thức chính qui, mang nghĩa là đối sánh được một chuỗi với r_1 nhưng chỉ nếu theo sau nó là một chuỗi r_2. Biểu thức chính qui r_2 sau *toán tử sải với* (lookahead operator) / chỉ ra ngữ cảnh bên phải của một đối sánh; nó chỉ được dùng để hạn chế một đối sánh, không phải là thành phần của đối sánh. Thí dụ một đặc tả Lex để nhận dạng từ khóa DO trong ngữ cảnh ở trên là

```
DO/({letter} | {digit})* = ({letter} | {digit})*,
```

Với đặc tả này, thể phân từ vựng sẽ xem trong vùng đệm nguyên liệu để tìm một chuỗi chữ cái và ký số có một dấu bằng theo sau, kế đến là các chữ cái và ký số rồi đến một dấu phẩy để bảo đảm rằng không có một câu lệnh gán. Thế thì chỉ các ký tự D và O đi trước toán tử sải với / mới là thành phần của từ tố đã đối sánh được. Sau khi đối sánh thành công, `yytext` chỉ đến D và `yyleng` = 2. Chú ý rằng mẫu sải với đơn giản này cho phép nhận dạng được DO khi theo sau nó là dãy ký tự lộn xộn như **Z4=6Q**, nhưng sẽ không bao giờ xem DO là thành phần của một định danh.

Thí dụ 3.12. Toán tử sái với có thể được dùng để giải quyết một vấn đề khó khăn khác khi phân tích từ vựng của ngôn ngữ Fortran: phân biệt các từ khóa với định danh. Thí dụ nguyên liệu

```
IF(I, J) = 3
```

là một câu lệnh gán rất hoàn chỉnh trong Fortran, không phải là câu lệnh `if`. Một cách để xác định từ khóa `IF` bằng cách sử dụng Lex là định nghĩa các ngữ cảnh có thể có ở bên phải bằng toán tử sải với. Dạng đơn giản của câu lệnh `if` là

IF (*điều kiện*) *câu lệnh*

Fortran 77 đưa ra một dạng câu lệnh if khác

IF (*điều kiện*) THEN
    *khối*_then
ELSE
    *khối*_else
END IF

Chúng ta chú ý rằng mỗi câu lệnh Fortran chưa được gán nhãn đều bắt đầu bằng một chữ cái và mỗi dấu ngoặc mở được dùng để nhóm toán hạng phải có một ký hiệu toán tử theo sau như =, +, hoặc dấu phẩy, một dấu ngoặc đóng khác hay cuối câu lệnh. Một dấu ngoặc đóng như thế không thể có một chữ cái theo sau. Trong tình huống này, để chắc chắn rằng `IF` là một từ khóa chứ không phải tên mảng, chúng ta có thể quét tới trước, tìm một dấu ngoặc đóng có một chữ cái theo sau trước khi gặp một ký tự newline. Mẫu cho từ khóa `IF` có thể được viết là

```
IF / \( .* \) {letter}
```

Dấu chấm thay cho "mọi ký tự trừ newline" và dấu gạch ngược phía trước các dấu ngoặc báo cho Lex biết phải xử lý chúng theo nghĩa tự nhiên. không phải là meta ký hiệu được dùng để nhóm trong các biểu thức chính qui (xem Bài tập 3.10). □

Một cách khác để giải quyết vấn đề được đặt ra bởi câu lệnh `if` trong Fortran là, sau khi gặp `IF(`, cần xác định xem `IF` đã được khai báo là một mảng hay chưa. Chúng ta quét toàn bộ mẫu ở trên chỉ khi nó đã được khai báo. Các kiểm tra như thế sẽ khiến cho việc cài đặt tự động hóa một thể phân từ vựng từ đặc tả Lex khó khăn hơn và chạy với thời gian lâu hơn bởi vì chương trình cần phải thường xuyên thực hiện những kiểm tra này, mô phỏng một sơ đồ chuyển vị để xác định xem những kiểm tra như thế có cần phải được thực hiện hay không. Chú ý rằng việc nhận dạng thẻ từ của Fortran là một công việc không có khuôn mẫu, và do đó viết một thể phân từ vựng cho Fortran bằng một ngôn ngữ lập trình truyền thống thường dễ hơn là dùng một bộ sinh thể phân từ vựng tự động.

3.6 AUTOMAT HỮU HẠN

Thể nhận dạng (regconizer) cho một ngôn ngữ là một chương trình nhận một chuỗi x làm nguyên liệu, trả lời "yes" nếu x là một câu của ngôn ngữ và trả lời "no" nếu không phải. Chúng ta dịch một biểu thức chính qui thành một thể nhận dạng bằng cách xây dựng một sơ đồ chuyển vị tổng quát hóa được gọi là *automat hữu hạn* (finite automata). Một automat hữu hạn có thể thuộc loại *đơn định* (deterministic) hoặc *đa định* (nondeterministic),[11] trong đó "đa định" muốn nói là có thể có nhiều chuyển vị đi ra khỏi một trạng thái trên cùng một ký hiệu nguyên liệu.

Cả automat đơn định lẫn đa định đều có khả năng nhận dạng một cách chính xác các tập chính qui. Vì vậy chúng đều có thể nhận dạng đúng đắn những gì mà biểu thức chính qui có thể biểu thị. Tuy nhiên cũng có những được mất giữa thời gian-không gian; trong khi automat hữu hạn đơn định có thể cho ra những thể nhận dạng nhanh hơn automat đa định, một automat đơn định lại có thể có kích thước lớn hơn nhiều so với một automat đa định tương đương. Trong phần tiếp theo, chúng ta sẽ trình bày các phương pháp biến đổi biểu thức chính qui thành cả hai loại automat. Biến đổi sang automat đa định thì trực tiếp hơn và vì thế chúng ta sẽ thảo luận về nó trước.

Những thí dụ trong phần này và phần tiếp theo sẽ giải quyết chủ yếu với ngôn ngữ được biểu thị bằng biểu thức chính qui $(a|b)^*abb$, là tập tất cả các chuỗi chữ cái a và b kết thúc bằng chuỗi abb. Các ngôn ngữ tương tự cũng hay gặp trong thực hành. Thí dụ một biểu thức chính qui biểu thị tên của tất cả mọi tập tin kết thúc là `.o` có dạng `(.|o|`c`)*.o`, với c biểu diễn một ký tự bất kỳ không phải dấu chấm hoặc chữ `o`. Một thí dụ khác, sau dấu mở `/*`, các lời giải thích trong C bao gồm các chuỗi ký tự kết thúc bằng `*/` với một yêu cầu là không có một tiền tố thực sự nào kết thúc bằng `*/`.

Automat Hữu hạn Đa định

Một *automat hữu hạn đa định* (nondeterministic finite automaton, viết tắt là NFA)[12] là một mô hình toán học gồm có

1. Một tập *trạng thái* S
2. Một tập ký hiệu nguyên liệu Σ (bộ ký tự nguyên liệu hay bộ chữ cái)
3. Một hàm chuyển vị *move* ánh xạ cặp *trạng thái–ký hiệu* thành các tập trạng thái
4. Một trạng thái s_0 được phân biệt là *trạng thái khởi đầu* (khởi trạng)
5. Một tập trạng thái F được xem là *trạng thái kiểm nhận* (accepting state) hay *trạng thái cuối* hay *kết trạng* (final state)

Một NFA có thể được biểu diễn một cách hình ảnh bằng đồ thị có hướng và có nhãn,

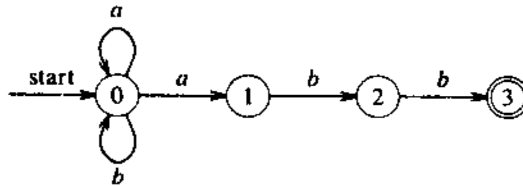
---

[11] Chúng tôi cung dịch là tất định và không tất định. (ND)

[12] Chúng tôi sẽ dùng chữ viết tắt NFA và DFA thay cho automat hữu hạn đa định và đơn định. (ND)

được gọi là *đồ thị chuyển vị* (transition graph), trong đó các nút là các trạng thái và các cạnh có nhãn biểu diễn *hàm chuyển vị* (transition function). Đồ thị này trông giống như một sơ đồ chuyển vị nhưng có thể có hai hoặc nhiều chuyển vị có cùng một ký tự làm nhãn và các cạnh có thể được gán nhãn bằng ký tự ε hoặc bằng các ký hiệu nguyên liệu.

Đồ thị chuyển vị cho NFA nhận dạng ngôn ngữ $(a|b)^*abb$ được trình bày trong Hình 3.19. Tập trạng thái của NFA là {0, 1, 2, 3} và bộ chữ cái là $\{a, b\}$. Trạng thái 0 trong Hình 3.19 được phân biệt và dùng làm khởi trạng còn trạng thái kiểm nhận 3 được chỉ ra bằng một vòng đôi.



Hình 3.19. Một automat hữu hạn đa định.

Khi mô tả một NFA, chúng ta dùng dạng đồ thị chuyển vị. Trong một máy tính, hàm chuyển vị của một NFA có thể được cài đặt bằng nhiều cách khác nhau như chúng ta sẽ thấy sau này. Cài đặt dễ nhất là dùng *bảng chuyển vị* (transition table), trong đó có một hàng dành cho một trạng thái và một cột dành cho một ký hiệu nguyên liệu và ký hiệu ε nếu cần. Mục ghi cho hàng i và ký hiệu a trong bảng là tập trạng thái (hoặc trong thực hành là một con trỏ chỉ đến tập trạng thái) có thể đến được bằng một chuyển vị từ trạng thái i trên nguyên liệu a. Bảng chuyển vị cho NFA của Hình 3.19 được trình bày trong Hình 3.20.

TRẠNG THÁI	KÝ HIỆU NGUYÊN LIỆU	
	a	b
0	{0, 1}	{0}
1	–	{2}
2	–	{3}

Hình 3.20. Bảng chuyển vị cho automat của Hình 3.19.

Biểu diễn dạng bảng có ưu điểm là nó cho phép truy xuất nhanh chóng đến các chuyển vị của một trạng thái đã cho trên một ký tự đã cho; khuyết điểm chính của nó

là có thể chiếm nhiều không gian khi bộ chữ cái lớn và phần lớn các chuyển vị đều có tập rỗng. Biểu diễn bằng *danh sách kề* (adjacency list) cho hàm chuyển vị tạo ra một cài đặt cô đặc hơn nhưng khi truy xuất thì khá chậm. Hiển nhiên là có thể dễ dàng chuyển đổi qua lại giữa hai lối cài đặt này.

Một NFA *kiểm nhận* một chuỗi nguyên liệu x nếu và chỉ nếu có một đường đi trong đồ thị chuyển vị từ trạng thái khởi đầu đến một trạng thái kiểm nhận nào đó sao cho các nhãn dọc theo đường đi này sẽ "tái hiện lại" x. NFA của Hình 3.19 kiểm nhận được các chuỗi *abb*, *aabb*, *babb*, *aaabb*, . . . Thí dụ *aabb* được kiểm nhận qua đường đi từ 0, theo cạnh có nhãn a đến lại trạng thái 0 rồi đến các trạng thái 1, 2, và 3 qua các cạnh có nhãn tương ứng là a, b, và b.

Một đường đi có thể được biểu diễn bằng một dãy các chuyển vị trạng thái gọi là *bước chuyển* (move). Sơ đồ sau trình bày các bước chuyển được thực hiện khi kiểm nhận chuỗi *aabb*:

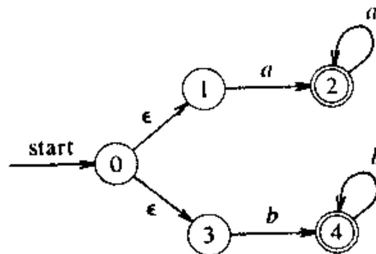
$$0 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$$

Nói chung, có thể có nhiều dãy bước chuyển dẫn đến một trạng thái kiểm nhận. Chú ý rằng nhiều dãy bước chuyển khác có thể được thực hiện trên chuỗi *aabb* nhưng dường như không có dãy nào kết thúc được ở một trạng thái kiểm nhận. Thí dụ một dãy bước chuyển trên nguyên liệu *aabb* cứ trở lại trạng thái 0

$$0 \xrightarrow{a} 0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{b} 0$$

Ngôn ngữ được định nghĩa bởi một NFA là tập chuỗi nguyên liệu được nó kiểm nhận. Không có gì khó khăn để chứng minh rằng NFA của Hình 3.19 kiểm nhận $(a|b)^*abb$.

Thí dụ 3.13. Trong Hình 3.21 chúng ta thấy một NFA nhận dạng $aa^*|bb^*$. Chuỗi *aaa* được kiểm nhận bằng cách di chuyển qua các trạng thái 0, 1, 2, 2, và 2. Nhãn của các cạnh này là ε, a, a, và a, được ghép lại thành *aaa*. Chú ý rằng ký hiệu ε "biến mất" khi được ghép. □



Hình 3.21. NFA kiểm nhận $aa^*|bb^*$.

Automat hữu hạn đơn định

Một *automat hữu hạn đơn định* (deterministic finite automaton, viết tắt là DFA) là một trường hợp đặc biệt của automat hữu hạn đa định, trong đó

1. Không có trạng thái nào có ε–chuyển vị, nghĩa là một chuyển vị trên nguyên liệu ε, và
2. Với mỗi trạng thái s và ký hiệu nguyên liệu a chỉ có tối đa một cạnh có nhãn a đi ra khỏi s.

Một DFA có tối đa một chuyển vị từ mỗi trạng thái trên một ký hiệu nguyên liệu. Nếu chúng ta đang dùng một bảng chuyển vị để biểu diễn hàm chuyển vị của một DFA thì mỗi mục trong bảng chuyển vị chỉ có một trạng thái duy nhất. Kết quả là chúng ta dễ dàng xác định được một DFA có kiểm nhận một chuỗi nguyên liệu hay không bởi vì có tối đa một đường đi từ trạng thái khởi đầu với chuỗi đó làm nhãn. Thuật toán sau đây trình bày cách mô phỏng hành động của một DFA trên một chuỗi nguyên liệu.

Thuật toán 3.1. Mô phỏng một DFA.

Nguyên liệu. Một chuỗi nguyên liệu x kết thúc bằng một ký tự cuối-tập-tin **eof**. Một DFA D với trạng thái khởi đầu s_0 và tập trạng thái kiểm nhận F.

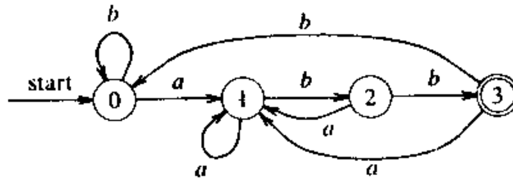
Thành phẩm. Câu trả lời "yes" nếu D kiểm nhận được x; nếu không thì trả lời "no".

Phương pháp. Áp dụng thuật toán trong Hình 3.22 cho chuỗi nguyên liệu x. Hàm *move*(s, c) cho biết trạng thái đến được từ trạng thái s trên ký tự c bằng một chuyển vị. Hàm *nextchar* trả về ký tự kế tiếp của chuỗi nguyên liệu x. □

```
s := s0;
c := nextchar;
while c ≠ eof do
    s := move(s, c);
    c := nextchar;
end;
if s thuộc F then
    return "yes"
else return "no";
```

Hình 3.22. Mô phỏng một DFA.

Thí dụ 3.14. Trong Hình 3.23 chúng ta thấy một đồ thị chuyển vị của một automat hữu hạn đơn định kiểm nhận ngôn ngữ $(a|b)^*abb$, là ngôn ngữ được NFA của Hình 3.19 kiểm nhận. Với DFA này và chuỗi nguyên liệu *ababb*, Thuật toán 3.1 đi qua dãy trạng thái 0, 1, 2, 1, 2, 3 và trả lời "yes". □



Hình 3.23. DFA kiểm nhận $(a|b)^*abb$.

Biến đổi NFA thành DFA

Chú ý rằng NFA của Hình 3.19 có hai chuyển vị từ trạng thái 0 trên nguyên liệu a; nghĩa là nó có thể đi đến trạng thái 0 hoặc 1. Tương tự NFA của Hình 3.21 có hai chuyển vị trên ε từ trạng thái 0. Mặc dù chúng ta không trình bày thí dụ về nó, tình huống có thể chọn một chuyển vị trên ε hoặc trên một ký hiệu nguyên liệu thực sự cũng gây ra tình trạng đa nghĩa. Các tình huống hàm chuyển vị có nhiều giá trị gây khó khăn cho việc mô phỏng một NFA bằng một chương trình máy tính. Định nghĩa về *kiểm nhận* (acceptance) chỉ khẳng định rằng phải có một đường đi có nhãn là chuỗi nguyên liệu đang xét dẫn từ trạng thái khởi đầu đến một trạng thái kiểm nhận. Nhưng nếu có nhiều đường đi đưa ra cùng một chuỗi nguyên liệu thì chúng ta phải xem xét tất cả chúng trước khi tìm ra một đường dẫn đến kiểm nhận hoặc khám phá ra rằng không có đường đi nào như thế.

Bây giờ chúng ta đưa ra một thuật toán xây dựng một DFA từ một NFA nhận dạng cùng một ngôn ngữ. Thuật toán này, thường được gọi là *phép dựng tập con* (subset construction), rất có ích trong việc mô phỏng một NFA bằng một chương trình máy tính. Nó là một thuật toán có quan hệ gần gũi và có vai trò cơ bản trong việc xây dựng thể phân cú pháp LR ở chương tiếp theo.

Trong bảng chuyển vị của một NFA, mỗi mục ghi là một tập trạng thái; trong bảng chuyển vị của một DFA, mỗi mục chỉ là một trạng thái. Ý tưởng chung ẩn chứa sau phép biến đổi NFA thành DFA là mỗi trạng thái DFA tương ứng với một tập trạng thái NFA. DFA sử dụng trạng thái của nó để theo dõi mọi trạng thái mà NFA có thể đến được sau khi đọc mỗi ký hiệu nguyên liệu. Điều đó nói lên rằng, sau khi đọc nguyên liệu $a_1a_2 \ldots a_n$, DFA chuyển đến trạng thái biểu diễn cho một tập con T các trạng thái của NFA mà chúng ta có thể đến được từ trạng thái khởi đầu của NFA này khi đi theo một đường đi có nhãn $a_1a_2 \ldots a_n$. Số lượng trạng thái của DFA có thể là hàm mũ theo số lượng trạng thái của NFA nhưng trong thực hành, trường hợp xấu nhất này rất hiếm khi xảy ra.

Thuật toán 3.2. (Phép dựng tập con) Xây dựng một DFA từ một NFA.

Nguyên liệu. Một NFA *N*.

Thành phẩm. Một DFA *D* cũng kiểm nhận ngôn ngữ của *N*.

Phương pháp. Thuật toán xây dựng một bảng chuyển vị *Dtran* cho *D*. Mỗi trạng thái DFA là một tập các trạng thái NFA và chúng ta xây dựng *Dtran* sao cho *D* sẽ hoạt động "song song" bằng cách mô phỏng tất cả mọi bước chuyển khả hữu mà *N* có thể thực hiện trên một chuỗi nguyên liệu đã cho.

Chúng ta dùng các phép toán trong Hình 3.24 để theo dõi các tập trạng thái của NFA (*s* biểu thị một trạng thái NFA và *T* là một tập trạng thái NFA).

PHÉP TOÁN	MÔ TẢ
ε-*closure*(*s*)	Tập trạng thái NFA đến được từ trạng thái NFA *s* trên các ε-chuyển vị
ε-*closure*(*T*)	Tập trạng thái NFA đến được từ một trạng thái *s* trong *T* trên các ε-chuyển vị
move(*T*, *a*)	Tập trạng thái NFA đến được qua một chuyển vị trên nguyên liệu *a* từ một trạng thái *s* thuộc *T*

Hình 3.24. Các phép toán trên các trạng thái NFA.

Trước khi thấy ký hiệu nguyên liệu đầu tiên, *N* có thể ở trong một trạng thái bất kỳ thuộc tập ε-*closure*(s_0), trong đó s_0 là trạng thái khởi đầu của *N*. Giả sử rằng chỉ các trạng thái trong tập *T* là có thể đến được từ s_0 trên một dãy ký hiệu nguyên liệu đã cho, và gọi *a* là ký hiệu kế tiếp. Khi thấy *a*, *N* có thể di chuyển đến một trong các trạng thái thuộc tập *move*(*T*, *a*). Khi cho phép dùng cả các ε-chuyển vị, *N* có thể ở một trong những trạng thái của tập ε-*closure*(*move*(*T*, *a*)) sau khi đọc *a*.

Chúng ta xây dựng *Dstates*, tập trạng thái của *D*, và *Dtran*, bảng chuyển vị cho *D* bằng cách sau đây. Mỗi trạng thái của *D* tương ứng với một tập trạng thái DFA mà *N* có thể đến được sau khi đọc một chuỗi ký hiệu nguyên liệu nào đó có chứa tất cả các ε-chuyển vị trước hoặc sau khi các ký hiệu được đọc. Khởi trạng của *D* là ε-*closure*(s_0). Các trạng thái và chuyển vị được thêm vào *D* bằng thuật toán của Hình 3.25. Một trạng thái của *D* là trạng thái kiểm nhận nếu nó là một tập trạng thái NFA chứa ít nhất một trạng thái kiểm nhận của *N*.

Việc tính ε-*closure*(*T*) chính là quá trình tìm kiếm các nút có thể đến được từ một tập nút đã cho trong một đồ thị. Trong trường hợp này, các trạng thái của *T* là tập nút đã cho và đồ thị chỉ bao gồm các cạnh có nhãn ε của NFA. Một thuật toán đơn giản tính ε-*closure*(*T*) có dùng một *chồng xếp* (stack) để lưu trạng thái có các cạnh chưa