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]

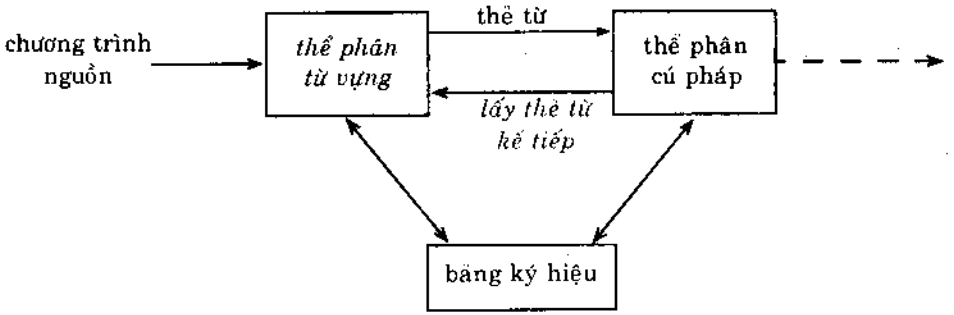
Một công cụ phần mềm tự động xây dựng thể phân từ vựng cho phép nhiều người với những hiểu biết khác nhau có thể áp dụng kỹ thuật so mẫu (đối sánh mẫu) vào những lãnh vực chuyên môn của họ. Thí dụ Jarvis [1976] đã sử dụng một *bộ sinh thể phân từ vựng* (lexical-analyzer generator) để tạo ra một chương 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 đề 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.

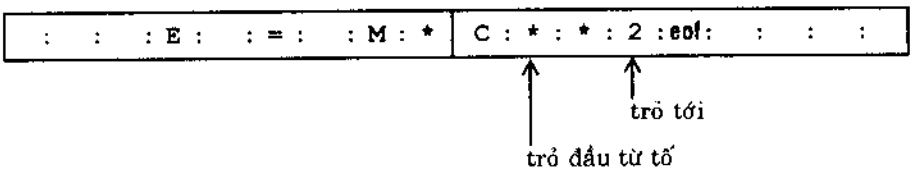
1. Sử dụng một chương trình để tạo ra thể phân từ vựng (bộ sinh thể phân từ vựng) từ phần đặc tả bằng biểu thức chính qui, chẳng hạn như trình biên dịch Lex sẽ được thảo luận trong Phần 3.5. Trong trường hợp này, chính bộ sinh thể phân từ vựng sẽ cung cấp các thủ tục để đọc và đệm nguyên liệu.
2. Viết 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 sai 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 )
```

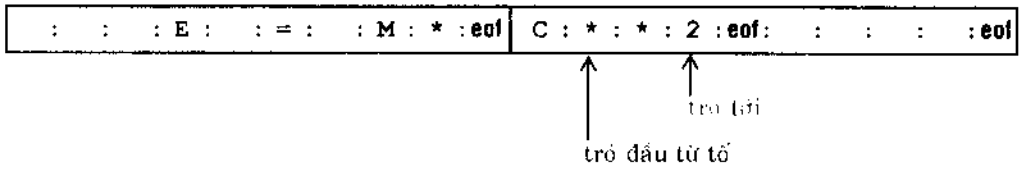
trong một chương trình PL/I, chúng ta không thể xác định được DECLARE là một từ khóa hay là tên mảng cho đến khi chúng nhìn thấy ký tự theo sau dấu ngoặc đóng. Trong mỗi trường hợp này, từ tố sẽ kết thúc tại ký tự E thứ hai, nhưng lượng sai với cần phải biết lại tỷ lệ với số lượng các đối mà về nguyên 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 ∅, là tập rỗng, hoặc {ε}, 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 $L \cup M$	$L \cup M = \{ s \mid s \text{ thuộc } L \text{ hoặc } s \text{ thuộc } M \}$
ghép nối của L và M được viết là LM	$LM = \{ st \mid s \text{ thuộc } L \text{ và } t \text{ thuộc } M \}$
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 $n\mathtt{H}a_1a_2 \ldots a_n$ 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:

$$
\begin{array}{lcl}
\mathit{stmt} & \rightarrow & \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{stmt} \\
 & | & \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{stmt}\ \textbf{else}\ \mathit{stmt} \\
 & | & \varepsilon \\
\mathit{expr} & \rightarrow & \mathit{term}\ \textbf{relop}\ \mathit{term} \\
 & | & \mathit{term} \\
\mathit{term} & \rightarrow & \textbf{id} \\
 & | & \textbf{num}
\end{array}
$$

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.

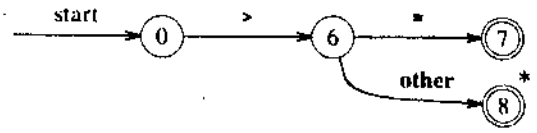
Chúng ta giả sử sơ đồ chuyển vị của phần này là *tất định* (deterministic);[5] nghĩa là không có ký hiệu nào khớp với nhãn của hai cạnh đi ra khỏi một trạng thái. Bắt đầu từ Phần 3.5, chúng ta sẽ giảm bớt điều kiện này, và tạo đơn giản hơn cho nhà thiết kế thể phân từ vựng và với những công cụ thích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

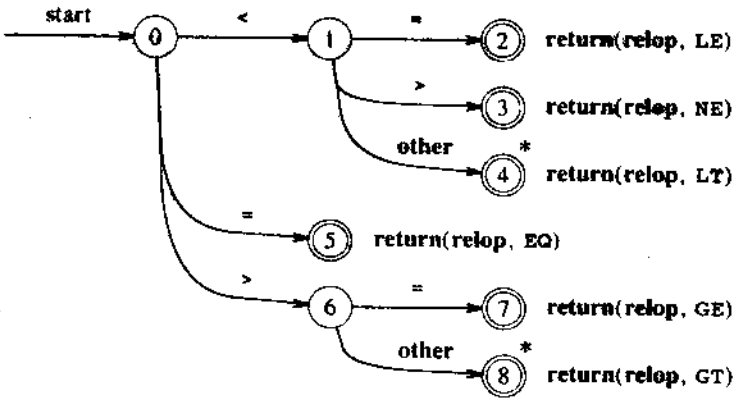
Khi đến được trạng thái 6, chúng ta đọc ký tự tiếp theo. Cạnh có nhãn = từ trạng thái 6 được đi theo để đến trạng thái 7 nếu ký tự này là =. Bằng không cạnh có nhãn **other** chỉ ra rằng chúng ta phải đi đến trạng thái 8. Vòng tròn kép ở trạng thái 7 chỉ ra rằng đây là *trạng thái kiểm nhận* (accepting 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.

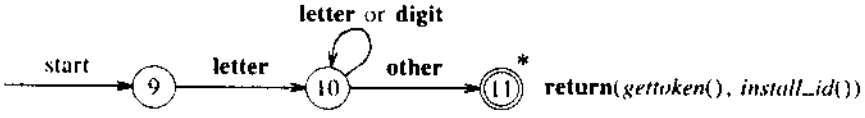
Nói chung thường có nhiều sơ đồ chuyển vị, mỗi sơ đồ đặc tả một nhóm thẻ từ. Nếu xảy ra thất bại khi chúng ta đang đi theo một sơ đồ chuyển vị thì chúng ta dịch lui con trỏ tới về nơi nó đã ở trong trạng thái khởi đầu của sơ đồ này rồi kích hoạt sơ đồ chuyển vị tiếp theo. Do con trỏ đầu từ tố và con 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.



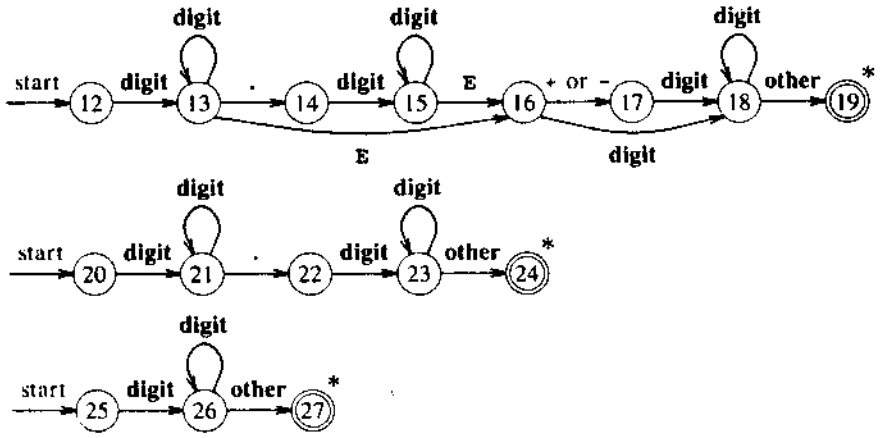
Hình 3.13. Sơ đồ chuyển vị cho định danh và từ khóa.

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

$$\textbf{num} \rightarrow \textbf{digit}^+ \; (\; . \; \textbf{digit}^+ \;)? \; (\; \texttt{E} \; (\; + \; | \; - \;)? \; \textbf{digit}^+ \;)?$$

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.

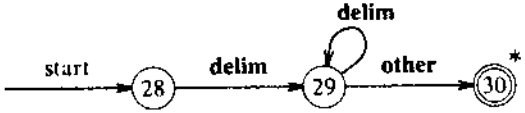
Hành động khi đến được một trong các trạng thái kiểm nhận 19, 24 hoặc 27 là gọi thủ tục *install_num* để nhập từ tố vào trong bảng các số và trả một con trỏ chỉ đến *mục ghi vừa được tạo ra. Thể phân từ vựng trả thẻ từ* **num** *cùng với con trỏ này làm* giá trị từ vựng. □

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 sai 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*.

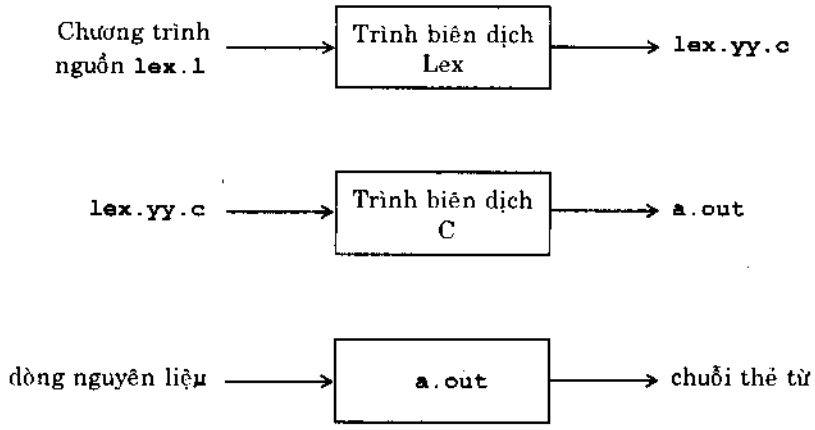
3.5 MỘT NGÔN NGỮ ĐẶC TẢ THỂ PHÂN TỪ VỰNG

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 lập 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 sải 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)

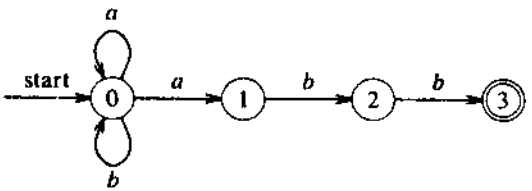
Một NFA có thể được biểu diễn một cách hình ảnh bằng đồ thị có hướng và có nhãn,

[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*:

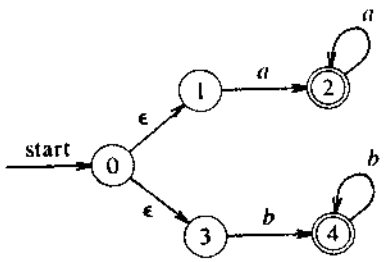
$$0 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$$

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.

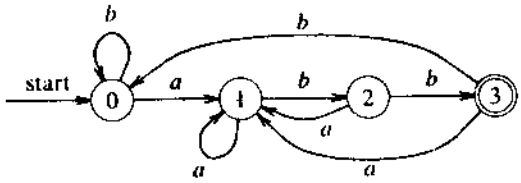
Thành phẩm. Câu trả lời "yes" nếu D kiểm nhận được x; nếu không thì trả lời "no".

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