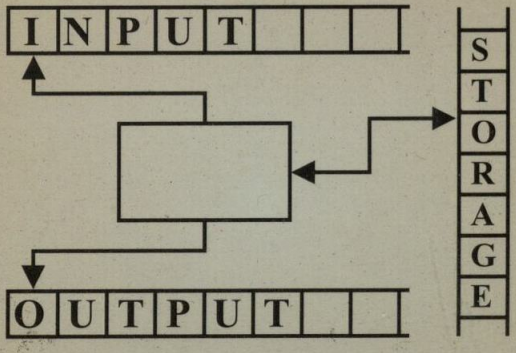
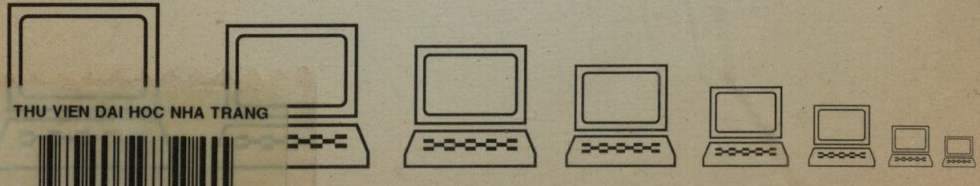
HỒ VĂN QUÂN

Giáo trình

LÝ THUYẾT AUTOMAT
VÀ
NGÔN NGỮ HÌNH THỨC





NHÀ XUẤT BẢN
ĐẠI HỌC QUỐC GIA

ĐẠI HỌC QUỐC GIA TP HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA

HỒ VĂN QUÂN

GIÁO TRÌNH
LÝ THUYẾT AUTOMAT VÀ NGÔN NGỮ HÌNH THỨC

NHÀ XUẤT BẢN ĐẠI HỌC QUỐC GIA
TP HỒ CHÍ MINH - 2002

MỤC LỤC

LỜI NÓI ĐẦU

Lý thuyết automat và ngôn ngữ hình thức *là một trong những lý thuyết nền tảng của khoa học máy tính. Automat mà đỉnh cao của nó là máy vạn năng Turing là một mô hình toán học của các máy tính (computer). Việc nghiên cứu lý thuyết automat cho chúng ta một cái nhìn đúng đắn về bản chất của máy tính. Trong khi đó lý thuyết ngôn ngữ hình thức lại nghiên cứu bản chất của ngôn ngữ và đặc trưng của các lớp ngôn ngữ. Việc nghiên cứu lý thuyết ngôn ngữ hình thức cho chúng ta một cái nhìn rõ ràng hơn về ngôn ngữ đồng thời những kết quả của nó cho phép chúng ta ứng dụng vào việc xây dựng các ngôn ngữ lập trình cũng như trong việc xử lý ngôn ngữ tự nhiên. Mặc dầu ban đầu chúng là hai lý thuyết riêng biệt, nhưng trong quá trình nghiên cứu người ta thấy giữa chúng có mối quan hệ mật thiết với nhau nên sau đó đã đặt chúng vào trong một lý thuyết có tên chung là* ***Lý thuyết automat và ngôn ngữ hình thức****. Vì tính chất cơ bản và quan trọng của lý thuyết này trong khoa học máy tính nên nó ngày càng trở thành một môn học chuẩn trong hầu hết các chương trình đào tạo đại học của ngành khoa học máy tính.*

Giáo trình LÝ THUYẾT AUTOMAT VÀ NGÔN NGỮ HÌNH THỨC *được thiết kế với mục đích giới thiệu cho sinh viên trong chương trình đào tạo khoa học máy tính ở bậc đại học, đặc biệt là các sinh viên năm thứ hai, thứ ba.*

Mặc dù nội dung môn học này là quan trọng đáng kể trong lĩnh vực khoa học máy tính, nhưng nguồn gốc của nó lại nằm ở các ngành khoa học khác. Một vài lý thuyết về tính toán bắt nguồn từ ngôn ngữ học và kỹ thuật hệ thống, nhưng hầu hết bắt nguồn bằng toán học. Trong một thời gian dài, những nguồn gốc đó đã ảnh hưởng đến cách mà giáo trình này đã được dạy, và cho đến gần đây, nhấn mạnh trên hình thức toán học, sự chính xác và chặt chẽ của toán học. Những thuận tiện từ một sự tiếp cận như vậy yêu cầu một sự chuẩn bị toán học toàn diện.

Cũng như trong các môn học khác, một khóa học giới thiệu về lý thuyết automat và ngôn ngữ hình thức phải được giới hạn trong một số phạm vi nhất định, tập trung trên các ý tưởng nền tảng hơn là bao phủ toàn diện, và trình bày các vấn đề cơ bản vững chắc cho việc nghiên cứu sâu hơn.

Như một yêu cầu, sinh viên phải quen thuộc với một vài ngôn ngữ lập trình cấp cao và với các khái niệm cơ bản của cấu trúc dữ liệu. Sinh viên cũng phải học xong khóa học về toán rời rạc.

Giáo trình được chia thành chín chương và một phụ lục. Chương 1 giới thiệu về các đối tượng sẽ nghiên cứu. Từ chương 2 đến chương 4 trình bày về lớp automat đầu tiên, đơn giản nhất đó là lớp automat hữu hạn trạng thái và lớp ngôn ngữ tương ứng, lớp ngôn ngữ chính qui. Từ chương 5 đến chương 8 trình bày về lớp ngôn ngữ cao cấp hơn, mạnh hơn, lớp ngôn ngữ phi ngữ cảnh, và các vấn đề liên quan đến nó như phân tích cú pháp, sự nhập nhằng và lớp automat tương ứng là automat đẩy xuống. Chương cuối, chương 9 và một phần của phụ lục trình bày về máy Turing, mô hình cao cấp nhất của automat, và trình bày về sức mạnh vạn năng của nó. Phần phụ lục giới thiệu một số giải thuật và các kết quả khác có liên quan đến môn học.

Để hoàn tất được giáo trình này, tôi xin chân thành cảm ơn các thầy cô trong Khoa Công nghệ Thông tin, Trường Đại học Bách khoa – Đại học Quốc gia Tp Hồ Chí Minh, những người đã trao cho tôi những kiến thức nền tảng quí báu và hướng tôi vào con đường khoa học. Tôi cũng xin chân thành cám ơn những bạn bè của tôi và cả những sinh viên của Khoa đã có nhiều ý kiến đóng góp quí giá cho giáo trình này. *Cho dù tôi đã rất cố gắng và mong muốn giới thiệu đến quí vị một giáo trình tốt nhất trong khả năng của mình về môn học này, song vẫn khó có thể tránh khỏi sai sót. Vì vậy, nếu quí vị độc giả, đặc biệt là các bạn sinh viên, thấy giáo trình còn có điều gì sai sót xin vui lòng gởi ý kiến về cho tôi theo địa chỉ:*

Hồ Văn Quân,

Khoa Công nghệ Thông tin – Trường Đại học Bách khoa – Đại học Quốc gia Tp Hồ Chí Minh,

268 Lý Thường Kiệt – Quận 10 – Thành phố Hồ Chí Minh.

Email: hvquan@dit.hcmut.edu.vn

Tôi xin chân thành cám ơn tất cả các ý kiến đóng góp của quí vị và sẽ cố gắng hoàn thiện hơn nữa trong lần tái bản sau của giáo trình này. Một lần nữa tôi xin chân thành cảm ơn tất cả.

Tháng 11 năm 2001

Tác giả

GIỚI THIỆU

Như tên gọi của môn học, môn học này bao gồm việc nghiên cứu hai phần: phần ôtômát và phần ngôn ngữ hình thức. Trong việc ứng dụng vào lĩnh vực khoa học máy tính, hai phần này có liên quan mật thiết với nhau nên chúng được đặt chung trong một môn học có tên như các bạn đã biết. Lý thuyết ôtômát là lý thuyết làm nền tảng cho việc hiểu các mô hình tự động mà điển hình nhất là máy tính số ngày nay. Lý thuyết ngôn ngữ hình thức là lý thuyết làm nền tảng cho việc hiểu khái niệm ngôn ngữ trong đó bao gồm cả khái niệm ngôn ngữ lập trình lẫn ngôn ngữ tự nhiên, và các vấn đề cơ bản của ngôn ngữ như xây dựng ngôn ngữ (xây dựng ngôn ngữ lập trình, phân tích cú pháp), dịch từ ngôn ngữ này sang ngôn ngữ khác (dịch từ ngôn ngữ lập trình cấp cao sang ngôn ngữ lập trình cấp thấp hay ngôn ngữ máy, dịch từ ngôn ngữ tự nhiên này sang ngôn ngữ tự nhiên khác...). Và đây cũng chính là các vấn đề cơ bản cần giải quyết trong quá trình phát triển của ngành khoa học máy tính. Chương này sẽ cho chúng ta biết các yêu cầu về kiến thức nền tảng mà người học cần có để học môn này, sau đó giới thiệu cho chúng ta các khái niệm cơ bản và một số ứng dụng của môn học. Về các ứng dụng của môn học, chúng tôi xin được trình bày nhiều hơn trong lần xuất bản sau của sách này.

Về hệ thống các kí hiệu được sử dụng trong sách này có thể khác với một số tài liệu khác cũng nói về môn học này, đây cũng là điều khó tránh khỏi. Vấn đề là các độc giả, khi đọc tài liệu nào thì phải cố gắng làm quen với hệ thống kí hiệu mà tài liệu đó sử dụng để hiểu được các vấn đề mà tài liệu trình bày.

1.1. YÊU CẦU VỀ KIẾN THỨC NỀN

Lý thuyết ôtômát và ngôn ngữ hình thức đặt nền tảng mạnh mẽ trên lý thuyết ***tập hợp***, bao gồm các khái niệm: tập hợp, hàm, ánh xạ, quan hệ, cũng như một số khái niệm và kết quả của lý thuyết ***đồ thị***.

Hai kỹ thuật chứng minh quan trọng được áp dụng rất phổ biến trong sách này để chứng minh các kết quả của môn học là kỹ thuật chứng minh ***qui nạp*** và kỹ thuật chứng minh ***phản chứng***. Một kỹ thuật nữa được áp dụng phổ biến trong sách là kỹ thuật ***mô phỏng***, kỹ thuật dùng để xây dựng một đối tượng tương đương với một đối tượng khác về một

mặt nào đó. Các kiến thức này thông thường được trình bày trong môn Toán Rời rạc hay Toán Máy tính được dạy ở năm thứ hai hay năm thứ ba trong chương trình đại học về khoa học máy tính. Một số tài liệu tham khảo về các kiến thức này các bạn có thể xem ở phần **Tài liệu tham khảo**.

1.2. BA KHÁI NIỆM CƠ BẢN

Ba khái niệm cơ bản của môn học này là

- ***Ngôn ngữ*** (*languages*)
- ***Văn phạm*** (*grammar*)
- ***Ôtômát*** (*automata*)

Các khái niệm này sẽ được trình bày xuyên suốt trong môn học này từ các lớp đơn giản nhất đến các lớp phức tạp nhất, cũng như mối quan hệ mật thiết giữa chúng.

Ghi chú: sau này để đảm bảo cho tính trung thực của từ ngữ tôi xin được phép sử dụng thuật ngữ "automat" (đại diện cho automata và automaton) thay cho thuật ngữ tiếng Việt: "ôtômát".

Đầu tiên chúng ta làm quen với khái niệm ngôn ngữ.

1. Ngôn ngữ

Tất cả chúng ta đều quen thuộc với khái niệm ngôn ngữ tự nhiên, chẳng hạn như tiếng Anh, tiếng Pháp. Tuy vậy, thật khó để nói chính xác:

Ngôn ngữ là gì?

Các từ điển định nghĩa ngôn ngữ một cách không chính xác là một hệ thống thích hợp cho việc biểu thị các ý nghĩ, các sự kiện, hay các khái niệm, bao gồm một tập các kí hiệu và các qui tắc để vận dụng chúng.

Nhận xét: định nghĩa trên chỉ cho chúng ta một ý niệm trực quan về ngôn ngữ là gì, nó chưa đủ là một định nghĩa chính xác để nghiên cứu về ngôn ngữ hình thức. Chúng ta cần một định nghĩa chính xác hơn cho nó. Chúng ta bắt đầu xây dựng định nghĩa này bằng khái niệm ***bảng chữ cái***, khái niệm mà mọi ngôn ngữ đều phải đặt nền tảng trên đó.

Bảng chữ cái (*alphabet*)

Là một tập hữu hạn không trống các kí hiệu (symbol) thường được kí hiệu là Σ.

Ví dụ

{A, B, C, ..., Z} : bảng chữ cái La tinh

{α, β, γ, ..., φ} : bảng chữ cái Hilạp

{0, 1, 2, ..., 9} : bảng chữ số thập phân
{I, *V*, *X*, L, C, D, M}: bảng chữ số La mã

***Chuỗi* (*string*)**

Là một dãy hữu hạn các kí hiệu từ bảng chữ cái.

Ví dụ: với Σ = {a, b}, thì abab và aaabbba là các chuỗi trên Σ.

Qui ước

Với một vài ngoại lệ, chúng ta sẽ sử dụng các chữ cái thường *a*, *b*, *c*,... cho các phần tử của Σ còn các chữ cái *u*, *v*, *w*,... cho các tên chuỗi.

Ví dụ

Nếu ta viết w = abaaa có nghĩa là chuỗi có tên là w có giá trị chỉ định là abaaa.

Các phép toán trên chuỗi

Kết nối (*concatenation*)

Kết nối *của hai chuỗi w và v là chuỗi nhận được bằng cách nối các kí hiệu của chuỗi v vào cuối cùng bên phải của chuỗi w, tức là, nếu*

$$w = a_1a_2...a_n$$

và

$$v = b_1b_2...b_m$$

thì kết nối của hai chuỗi w và v, được ký hiệu bởi wv, là

$$wv = a_1a_2...a_nb_1b_2...b_m$$

Đảo (*reverse*)

Đảo của một chuỗi *là chuỗi nhận được bằng cách viết các kí hiệu theo thứ tự ngược lại; nếu w là chuỗi như được trình bày ở trên thì chuỗi ngược của nó* w^R *là*

$$w^R = a_n...a_2a_1$$

Tiếp đầu ngữ (*prefix*)**, *tiếp vĩ ngữ*** (*suffix*)

Nếu w = vu, thì v được gọi là ***tiếp đầu ngữ*** *và u được gọi là* ***tiếp vĩ ngữ*** *của w.*

Chiều dài (*length*)

Chiều dài của chuỗi *w được ký hiệu là* $|w|$*, là số kí hiệu trong chuỗi.*

Chuỗi trống (*empty string*)

Chuỗi trống *là chuỗi không có kí hiệu nào, và thường được kí hiệu bằng* λ. *(Một số tài liệu khác kí hiệu là* ε*)*

Nhận xét

1 – Các quan hệ sau đây là đúng với mọi chuỗi w:

$$|\lambda| = 0$$

$$\lambda w = w\lambda = w$$

2 – Nếu u, v là các chuỗi thì:

$$|uv| = |u| + |v|$$

Lũy thừa *(power)*

Nếu w là một chuỗi thì w^n là một chuỗi nhận được bằng cách kết nối chuỗi w với chính nó n lần.

$$w^n = \underbrace{w \ldots w}_{n \text{ lần}}$$

Đặc biệt, chúng ta định nghĩa: $w^0 = \lambda$, đối với mọi chuỗi w.

Σ^*, Σ^+

Nếu Σ là một bảng chữ cái thì Σ^* là tập tất cả các chuỗi trên Σ kể cả chuỗi trống.

Còn Σ^+ là tập tất cả các chuỗi trên Σ ngoại trừ chuỗi trống.

$$\Sigma^+ = \Sigma^* - \{\lambda\}$$

Nhận xét: Σ thì hữu hạn còn Σ^* và Σ^+ là vô hạn và là vô hạn đếm được.

Ngôn ngữ

*Một **ngôn ngữ** được định nghĩa rất tổng quát như là một tập con của Σ^*. Hay nói cách khác **ngôn ngữ** là một tập bất kỳ các câu trên bộ chữ cái và thường được kí hiệu là L.*

Ví dụ 1.1. Cho $\Sigma = \{a, b\}$,

thì $\Sigma^* = \{\lambda, a, b, aa, ab, ba, bb, aaa, aab, \ldots\}$.

Tập $\{a, aa, aab\}$ là một ngôn ngữ trên Σ. Nó là một ngôn ngữ hữu hạn.

Tập $L = \{a^n b^n : n \geq 0\}$ cũng là một ngôn ngữ trên Σ. Nó là một ngôn ngữ vô hạn.

Các chuỗi *aabb* và *aaaabbbb* là các chuỗi nằm trong ngôn ngữ L, nhưng chuỗi *abb* thì không.

Từ *(word)*, ***câu*** *(sentence)*

*Một **từ** hay **câu** là một chuỗi của ngôn ngữ. Trong lý thuyết ngôn ngữ hình thức, chúng ta không phân biệt giữa từ và câu và sử dụng chúng*

thay thế lẫn nhau được. (Không giống như trong ngôn ngữ tự nhiên, chúng là hai khái niệm phân biệt).

Các phép toán trên ngôn ngữ

Nhận xét

Vì ngôn ngữ là tập hợp, nên dĩ nhiên nó có các phép toán **hội** (*union*), **giao** (*intersection*), và **hiệu** (*diference*) như trong lý thuyết tập hợp và chúng ta mặc nhiên dùng mà không cần định nghĩa lại. Ở đây sẽ định nghĩa thêm một số phép toán quan trọng khác.

Bù (*complement*)

Bù của ngôn ngữ L trên bảng chữ cái Σ, *được kí hiệu là*

$$\overline{L} = \Sigma^* - L$$

Chú ý: *khái niệm bù của ngôn ngữ được định nghĩa dựa trên* Σ^*.

Kết nối (*concatenation*)

Nối của hai ngôn ngữ L_1 *và* L_2 *trên cùng một bảng chữ cái là tập tất cả chuỗi nhận được bằng cách nối bất kỳ một chuỗi của ngôn ngữ* L_1 *với một chuỗi bất kỳ của ngôn ngữ* L_2.

$$L_1L_2 = \{xy : x \in L_1, y \in L_2\}$$

Định nghĩa này cũng có thể mở rộng ra cho trường hợp hai ngôn ngữ không cùng bảng chữ cái, lúc đó bảng chữ cái của ngôn ngữ kết quả là hội của hai bảng chữ cái của hai ngôn ngữ thành phần.

Lũy thừa (*power*)

Chúng ta định nghĩa L^n *như là L được nối với chính nó n lần:*

$$L^n = \underbrace{L \ldots L}_{n \text{ lần}}$$

Với trường hợp đặc biệt $L^0 = \{\lambda\}$ *đối với mọi ngôn ngữ L.*

Ví dụ 1.2. Nếu $L = \{a^n b^n : n \geq 0\}$,

thì $L^2 = \{a^n b^n a^m b^m : n \geq 0, m \geq 0\}$

L*, L+

Bao đóng-sao (*star-closure*) *của một ngôn ngữ L được kí hiệu là* L^* *với*

$$L^* = L^0 \cup L^1 \cup L^2 \ldots$$

còn ***bao đóng dương*** (*positive closure*) *được kí hiệu là* L^+ *với*

$$L^+ = L^1 \cup L^2 \cup L^3 \ldots$$

hay

$$L^+ = L.\, L^*$$

Ví dụ 1.3. Nếu $L = \{a, ba\}$
thì

$L^2 = \{aa, aba, baa, baba\}$

$L^3 = \{aaa, aaba, abaa, ababa, baaa, baaba, babaa, bababa\}$

$L^* = \{\lambda, a, ba, aa, aba, baa, baba, aaa, aaba, abaa, ababa, baaa, baaba, babaa, bababa, ...\}$

Ghi chú: bao đóng–sao và bao đóng dương đôi khi còn được gọi là ***bao đóng–Kleene.***

2. Văn phạm

Để nghiên cứu ngôn ngữ, chúng ta cần một cơ chế để mô tả nó. Một trong số những cơ chế đó mà được áp dụng một cách phổ biến và mạnh mẽ đó là cơ chế dùng ***văn phạm***. Vậy:

Văn phạm là gì?

Các từ điển định nghĩa văn phạm một cách không chính xác là một tập các qui tắc về cấu tạo từ và các qui tắc về cách liên kết các từ lại thành câu.

Để hiểu rõ hơn khái niệm này các bạn có thể quan sát ví dụ sau:

Ví dụ 1.4. Cho đoạn văn phạm tiếng Anh sau:

<*sentence*> → <*noun phrase*><*predicate*>,

<*noun phrase*> → <*article*><*noun*>,

<*predicate*> → <*verb*>,

<*article*> → *a|the*,

<*noun*> → *boy|dog*,

<*verb*> → *runs|walks*,

thì các câu "*a boy runs*" và "*the dog walks*" là có "dạng đúng", có nghĩa là "được sinh ra" từ các luật của văn phạm.

Nhận xét

Nếu chúng ta có một văn phạm đầy đủ cho nó (ngôn ngữ tiếng Anh) thì theo lý thuyết, mọi câu đúng đều phải có thể được giải thích theo kiểu như trên.

Ví dụ này minh họa cho việc định nghĩa một khái niệm tổng quát thông qua lần lượt các khái niệm đơn giản hơn. Chúng ta bắt đầu với khái niệm ở mức trên cùng, ở đây đó là <sentence>, và lần lượt định nghĩa nó thành những phần ***không thể rút gọn*** được nữa của ngôn ngữ (trong ngôn ngữ tự nhiên đó chính là các từ). Tổng quát hóa những ý tưởng này dẫn ta đến một định nghĩa cho khái niệm văn phạm hình thức.

Định nghĩa 1.1. *Một văn phạm G được định nghĩa như là bộ bốn*

$$G = (V, T, S, P)$$

trong đó: V – là một tập hữu hạn các đối tượng được gọi là các ***biến*** *(variable), đôi khi còn được gọi là các* ***kí hiệu không kết thúc*** *(nonterminal symbol),*

T – là tập hữu hạn các đối tượng được gọi là các ***ký hiệu kết thúc*** *(terminal symbol),*

$S \in V$ – là một ký hiệu đặc biệt được gọi là ***biến khởi đầu*** *(start variable) đôi khi còn được gọi là* ***kí hiệu mục tiêu.***

P – là một tập hữu hạn các ***luật sinh*** *(production) đôi khi còn được gọi là các* ***qui tắc*** *(rule) hay* ***luật viết lại*** *(written rule)*

Các luật sinh có dạng $x \to y$ trong đó $x \in (V \cup T)^+$ và có chứa ít nhất một biến, $y \in (V \cup T)^*$.

Qui ước

1. Các kí tự chữ hoa A, B, C, D, E, và S biểu thị các biến; S là kí hiệu khởi đầu (biến khởi đầu) trừ phi được phát biểu khác đi.
2. Các kí tự chữ thường a, b, c, d, e, các kí số (*digits*), và các chuỗi in đậm biểu thị các kí hiệu kết thúc.
3. Các kí tự chữ hoa X, Y, và Z biểu thị các kí hiệu có thể là kí hiệu kết thúc hoặc biến.
4. Các kí tự chữ thường u, v, w, x, y, và z biểu thị chuỗi các kí hiệu kết thúc.
5. Các kí tự chữ thường Hi lạp α, β, và γ biểu thị chuỗi có thể bao gồm các biến và các kí hiệu kết thúc.

Nhận xét

Bằng qui ước này chúng ta có thể suy ra các biến, các kí hiệu kết thúc và kí hiệu khởi đầu của văn phạm một cách duy nhất bằng cách khảo sát các luật sinh. Vì vậy chúng ta có thể biểu diễn văn phạm một cách đơn giản bằng cách liệt kê ra các luật sinh của chúng.

Từ văn phạm để "sinh ra" được các câu ta định nghĩa khái niệm:

Dẫn xuất trực tiếp *(directly derive)*

Nếu $x \to y$ là một luật sinh và $w = uxv$ là một chuỗi thì ta bảo rằng luật sinh đó có thể áp dụng tới chuỗi này, khi áp dụng ta sẽ nhận được chuỗi mới $z = uyv$ gọi là ***chuỗi dẫn xuất*** *của chuỗi w và được kí hiệu là*

$$uxv \Rightarrow uyv$$

Quan hệ trên đây được gọi là quan hệ ***dẫn xuất trực tiếp*** *(gọi tắt là quan hệ dẫn xuất) và chúng ta nói rằng* ***w dẫn xuất ra z*** *hay ngược lại* ***z được dẫn xuất ra từ w.***

$\overset{*}{\Rightarrow}$, $\overset{+}{\Rightarrow}$

Nếu

$$w_1 \Rightarrow w_2 \Rightarrow \ldots \Rightarrow w_n,$$

chúng ta nói rằng w_1 *dẫn xuất ra* w_n *và viết*

$$w_1 \overset{*}{\Rightarrow} w_n$$

Nếu có ít nhất một luật sinh phải được áp dụng thì chúng ta viết:

$$w_1 \overset{+}{\Rightarrow} w_n$$

Hai dẫn xuất trên được gọi là ***dẫn xuất gián tiếp***.

Chú ý: từ định nghĩa trên suy ra w $\overset{*}{\Rightarrow}$ w là đúng còn w $\overset{+}{\Rightarrow}$ w là sai.

Nhận xét: các luật sinh là ***trái tim*** của văn phạm; chúng chỉ ra làm thế nào văn phạm ***biến đổi*** một chuỗi thành một chuỗi khác hay ***sinh ra*** một chuỗi này từ một chuỗi khác, và thông qua cách này chúng – các luật sinh – định nghĩa một ngôn ngữ liên kết với văn phạm.

Định nghĩa 1.2. *Cho* $G = (V, T, S, P)$ *là một văn phạm, thì tập*

$$L(G) = \{w \in T^* : S \overset{*}{\Rightarrow} w\}$$

là ngôn ngữ được sinh ra bởi G.

Sự dẫn xuất câu *(derivation)*

Nếu $w \in L(G)$*, thì phải tồn tại dãy dẫn xuất*

$$S \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \ldots \Rightarrow w_n \Rightarrow w$$

và dãy này được gọi là một ***sự dẫn xuất câu*** *của w.*

Dạng câu *(sentential forms)*

Dãy $S, w_1, w_2, \ldots, w_n$*, ở trên mà chứa các biến cũng như các kí hiệu kết thúc, được gọi là* ***các dạng câu*** *của sự dẫn xuất. Câu w cũng được xem là một dạng câu đặc biệt.*

Ví dụ 1.5. Cho văn phạm sau:

$$G = (\{S\}, \{a, b\}, S, P)$$

với P: $S \rightarrow aSb,$

$S \rightarrow \lambda.$

thì $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$

là một dãy dẫn xuất và vì vậy có thể viết

$$S \overset{*}{\Rightarrow} aabb$$

Chuỗi *aabb* là một câu của ngôn ngữ được sinh ra bởi *G*, còn *aaSbb* là một dạng câu.

Ngôn ngữ tương ứng với văn phạm này là

$$L(G) = \{a^n b^n : n \geq 0\}.$$

Ví dụ 1.6. Tìm văn phạm sinh ra ngôn ngữ sau:

$$L(G) = \{ a^n b^{n+1} : n \geq 0\}.$$

Văn phạm kết quả là

$$S \rightarrow Ab,$$
$$A \rightarrow aAb,$$
$$A \rightarrow \lambda.$$

Ví dụ 1.7. Cho $\Sigma = \{a, b\}$, và $n_a(w)$, $n_b(w)$ lần lượt biểu thị số kí hiệu a và số kí hiệu b trong chuỗi w, thì văn phạm G sau:

$$S \rightarrow SS,$$
$$S \rightarrow \lambda,$$
$$S \rightarrow aSb,$$
$$S \rightarrow bSa,$$

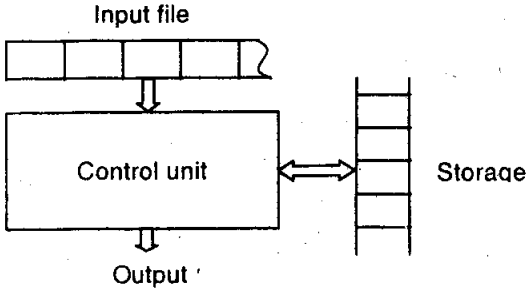
sinh ra ngôn ngữ $L = \{w \in \{a, b\}^* : n_a(w) = n_b(w)\}$.

3. Automat (máy tự động)

Automat là gì?

Automat, dịch nghĩa là máy tự động, được hiểu là các máy tự động, là các máy thực hiện một công việc nào đó một cách tự động. Tự động có nghĩa là tự thực hiện mà ***không cần sự can thiệp của con người***. Dĩ nhiên để có được những quá trình tự động như vậy con người thường phải "lập trình" sẵn cho nó một "lộ trình" thực hiện và các máy chỉ việc theo lộ trình đó mà thực hiện. Và trong số những máy tự động đó điển hình mạnh nhất có thể nói chính là máy tính số ngày nay. Vậy có thể định nghĩa bằng lời rằng:

Automat là một mô hình trừu tượng của một máy tính số như được vẽ ra trong H.1.1. *Nó bao gồm các thành phần chủ yếu sau:*



Hình 1.1

Thiết bị đầu vào *(input file)*

Là nơi mà các chuỗi nhập (input string) được ghi lên, và được automat đọc nhưng không thay đổi được nội dung của nó. Nó được chia thành các ô (cells, squares), mỗi ô giữ được một kí hiệu.

Cơ cấu nhập *(input mechanism)*

Là bộ phận có thể đọc input file từ trái sang phải, một kí tự tại một thời điểm. Nó cũng có thể dò tìm được điểm kết thúc của chuỗi nhập (bằng cách cảm giác một điều kiện kết thúc file). Kí hiệu kết thúc của input file thường được kí hiệu là ***eof****. Trong tài liệu này chúng ta sử dụng một kí tự thay thế là #.*

Bộ nhớ tạm *(temporary storage)*

Là thiết bị bao gồm một số không giới hạn các ô nhớ (cell), mỗi ô có thể giữ một kí hiệu từ một bảng chữ cái (không nhất thiết giống với bảng chữ cái ngõ nhập). Automat có thể đọc và thay đổi được nội dung của các ô nhớ lưu trữ (storage cell).

Đơn vị điều khiển *(control unit)*

Mỗi automat có một đơn vị điều khiển, cái mà có thể ở trong một trạng thái bất kỳ trong một số hữu hạn các ***trạng thái nội*** *(internal state), và có thể đổi trạng thái trong một kiểu được định nghĩa sẵn nào đó.*

Hoạt động của một automat

Một automat được giả thiết là hoạt động trong một ***khung thời gian rời rạc*** *(discrete time frame). Tại một thời điểm bất kỳ đã cho, đơn vị điều khiển là đang ở vào trong một* ***trạng thái nội*** *(internal state) nào đó, và cơ cấu nhập là đang quét (scanning) một kí hiệu cụ thể nào đó trên input file. Trạng thái nội của đơn vị điều khiển tại thời điểm kế tiếp được xác định bởi* ***trạng thái kế*** *(next state) hay bởi* ***hàm chuyển trạng thái*** *(transition function). Trong suốt quá trình chuyển trạng thái từ khoảng thời gian này đến khoảng thời gian kế, kết quả (output) có thể được sinh ra và thông tin trong bộ nhớ lưu trữ có thể được thay đổi.*

Trạng thái nội *(internal state)*

Là một trạng thái của đơn vị điều khiển mà nó có thể ở vào.

Trạng thái kế *(next state)*

Là một trạng thái nội của đơn vị điều khiển mà nó sẽ ở vào tại thời điểm kế tiếp.

Hàm chuyển trạng thái *(transition function)*

Là hàm gởi ra trạng thái kế của automat dựa trên trạng thái hiện hành, kí hiệu nhập hiện hành được quét, và thông tin hiện hành trong bộ nhớ tạm.

Cấu hình *(configuration)*

Được sử dụng để tham khảo đến bộ ba thông tin: trạng thái cụ thể mà đơn vị điều khiển đang ở vào, vị trí của cơ cấu nhập trên thiết bị nhập (hay nói cách khác automat đang đọc đến kí hiệu nào của thiết bị nhập), và nội dung hiện hành của bộ nhớ lưu trữ tạm.

Di chuyển *(move)*

Là sự chuyển trạng thái của automat từ một cấu hình này sang cấu hình kế tiếp.

Phân loại các automat

Dựa vào các yếu tố của automat mà có các cách phân loại sau:

Một, dựa vào hoạt động của automat, có đơn định hay không mà có hai loại automat.

Automat đơn định *(hay tất định) (deterministic automat)*

Là một automat mà trong đó mỗi di chuyển (move) là được xác định duy nhất bởi cấu hình hiện tại. Sự duy nhất này thể hiện tính đơn định, có nghĩa là ta sẽ biết chắc nó sẽ di chuyển đến đâu, đến trạng thái hay cấu hình nào.

Automat không đơn định *(non-deterministic automat)*

Là một automat mà tại mỗi thời điểm, nó có một vài khả năng lựa chọn để di chuyển (move). Việc có một vài khả năng lựa chọn thể hiện tính không đơn định, nghĩa là ta sẽ không biết chắc nó sẽ di chuyển đến trạng thái hay cấu hình nào trong số các khả năng có thể.

Hai, dựa vào kết quả xuất ra của automat mà có hai loại automat.

Accepter

Là một automat mà đáp ứng ở ngõ ra của nó được giới hạn trong hai trạng thái đơn giản "yes" hay "no". "Yes" tương ứng với việc chấp nhận chuỗi nhập, "no" tương ứng với việc từ chối, không chấp nhận chuỗi nhập. Automat này có thể được hình dung khi được dùng để ***cho biết một chuỗi có là một câu của một ngôn ngữ*** *nào đó hay không. Nếu trong ngữ cảnh ngôn ngữ tự nhiên thì điều này tương đương với việc* ***kiểm tra một câu có đúng cú pháp của một ngôn ngữ tự nhiên*** *nào đó hay không. Còn nếu nói trong ngữ cảnh của ngôn ngữ lập trình thì điều này tương*

đương với việc ***kiểm tra một chương trình có đúng cú pháp của một ngôn ngữ lập trình*** *nào đó hay không cái mà phải được thực hiện trước khi thực hiện quá trình dịch chương trình đó sang ngôn ngữ máy.*

Transducer

Là một automat tổng quát hơn, có khả năng sinh ra các chuỗi kí tự ở ngõ xuất. Máy tính số là một transducer điển hình. Hay một ví dụ đơn giản hơn các máy tính tay: bạn nhập vào một số dữ liệu đầu vào chẳng hạn đó là hai số và phép cộng nó sẽ xuất ra một số là kết quả của phép cộng là một ví dụ về transducer.

1.3. MỘT VÀI ỨNG DỤNG

Lý thuyết ngôn ngữ hình thức và automat có những ứng dụng gì?

Mặc dù chúng ta nhấn mạnh tính trừu tượng và tính toán học của ngôn ngữ hình thức và automat, thật ra những khái niệm này lại có những ứng dụng rộng rãi trong khoa học máy tính và trong thực tế, và là một chủ đề mà có liên quan đến nhiều lĩnh vực đặc biệt. Trong phần này chủ yếu nêu ra các lĩnh vực mà môn học này có ứng dụng vào còn các ví dụ cụ thể của các ứng dụng sẽ được trình bày trong một phiên bản sau của sách này.

1. Xây dựng các ngôn ngữ lập trình và các trình dịch

Ngôn ngữ hình thức và văn phạm được sử dụng rộng rãi trong sự liên kết với các ngôn ngữ lập trình. Các kết quả nghiên cứu của nó được ứng dụng vào việc xây dựng các ngôn ngữ lập trình cũng như các trình biên dịch cho chúng. Chẳng hạn

Dùng văn phạm để định nghĩa các thành phần của ngôn ngữ lập trình

Ví dụ 1.8. Giả thiết một danh hiệu của Pascal được bắt đầu bằng một kí tự chữ và được theo sau bởi một số tùy ý kí tự chữ và số bất kỳ, thì văn phạm sau là một văn phạm mô tả các danh hiệu của Pascal theo giả thiết này:

$$\begin{aligned} \langle id\rangle &\rightarrow \langle letter\rangle\langle rest\rangle, \\ \langle rest\rangle &\rightarrow \langle letter\rangle\langle rest\rangle|\langle digit\rangle\langle rest\rangle|\lambda, \\ \langle letter\rangle &\rightarrow a|b|\ldots \\ \langle digit\rangle &\rightarrow 0|1|\ldots \end{aligned}$$

Ghi chú: một tham khảo đầy đủ hơn đến văn phạm mô tả cho ngôn ngữ Pascal có thể xem ở sách số [4], phần phụ lục A, trang 745.

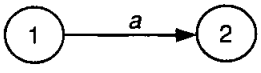
Dùng accepter để định nghĩa một vài thành phần của ngôn ngữ lập trình

Một cách hữu hiệu khác để định nghĩa một ngôn ngữ lập trình là dùng accepter. Để thuận tiện cho việc trình bày trước hết ta giới thiệu khái niệm đồ thị chuyển trạng thái – ĐTCTT (*transition graph*) là loại đồ thị dùng để biểu diễn cho các automat hữu hạn.

Đồ thị chuyển trạng thái (*transition graph*)

Là đồ thị mà trong đó các đỉnh tương ứng với các trạng thái của automat, còn các cạnh tương ứng với các chuyển trạng thái. Nhãn của cạnh cho biết điều gì xảy ra (đối với ngõ nhập và ngõ xuất) trong khi chuyển trạng thái.

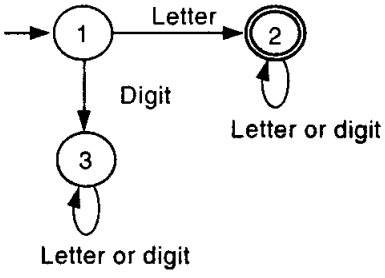
Ví dụ 1.9. Hình 1.2 biểu diễn một chuyển trạng thái từ trạng thái 1 đến trạng thái 2, xảy ra khi kí hiệu nhập (*input symbol*) là *a*.



Hình 1.2

Ví dụ 1.10. Automat sau trong H.1.3 là một automaton chấp nhận các danh hiệu của Pascal theo giả thiết ở trên.

Trong đó trạng thái 1 là trạng thái khởi đầu của automat, trạng thái 2 là trạng thái "yes" của accepter, trạng thái 3 là trạng thái "no" của accepter. Ở đây chúng ta giả thiết rằng không có loại kí tự nào khác ngoài hai loại kí tự chữ và số nói trên.



Hình 1.3

Ghi chú: các trình biên dịch và các trình dịch khác sử dụng khá rộng rãi các ý tưởng được trình bày trong các ví dụ trên.

Xây dựng các bộ phân tích từ vựng (*lexical analyzer*) và cú pháp (*parser*) trong các ngôn ngữ lập trình

Điều này sẽ được trình bày trong các chương 2 và chương 6 của giáo trình.

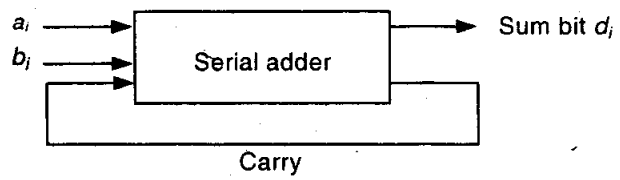
Xử lý chuỗi trong ngôn ngữ lập trình và trong ngôn ngữ tự nhiên.

Chẳng hạn vấn đề tìm kiếm, thay thế chuỗi theo mẫu trong các trình xử lý văn bản, hay vấn đề "*tagging*" (thêm chú thích) trong vào các từ, các câu trong một khối văn liệu (*corpus*), cái mà được ứng dụng rất nhiều vào xử lý các vấn đề của ngôn ngữ tự nhiên.

2. Ứng dụng lý thuyết automat vào trong lĩnh vực thiết kế số

Lý thuyết automat là một công cụ thiết kế số hữu hiệu. Nó là một trong những cách hữu hiệu được dùng để "mô tả chức năng hoạt động" của các mạch số.

Ví dụ 1.11. Xét một bộ cộng nhị phân tuần tự hai số nguyên dương. Một sơ đồ khối của nó được vẽ trong H.1.4.



Hình 1.4

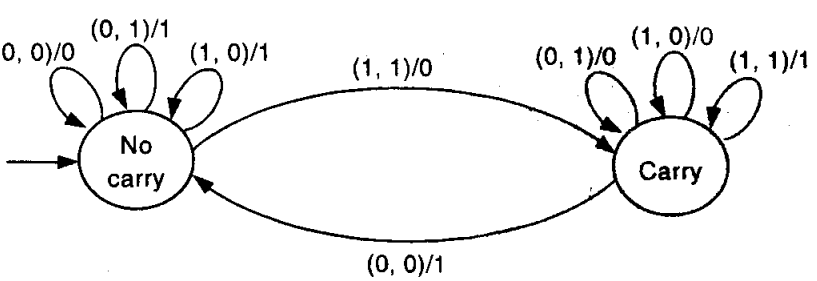
Trong đó hai chuỗi cộng: $x = a_0 a_1 \ldots a_n$

$y = b_0 b_1 \ldots b_m$

biểu diễn cho hai số nguyên: $v(x) = \sum_{i=0}^{n} a_i 2^i$

$v(y) = \sum_{i=0}^{m} b_i 2^i$

Nhận xét: sơ đồ khối này chỉ mô tả những gì mà một bộ cộng phải làm chứ không giải thích về hoạt động bên trong của bộ cộng. Để làm điều này người ta thường sử dụng automat để làm. Hình 1.5 sau đây vẽ ra một automat (trong trường hợp này là một transducer) mô tả hoạt động bên trong của bộ cộng nói trên.



Hình 1.5

Dữ liệu đầu vào (*input*) cho transducer này là các cặp bit (a_i, b_i), kết quả xuất ra (*output*) sẽ là bit tổng d_i. Các nhãn cạnh của nó sẽ có dạng $(a_i, b_i)/d_i$. Một hình vẽ đầy đủ của transducer này được cho ở H.1.5.

Nhận xét

Như ví dụ này chỉ ra một automat có thể phục vụ như một cầu nối giữa việc mô tả chức năng mức cao của một mạch và việc hiện thực logic của nó thông qua các transistor, các cổng gate, và các flip–flop. Ngày nay đã có những giải thuật để biến đổi những mô tả chức năng của một mạch số bằng công cụ automat thành những mạch số thực sự hoạt động đúng như những gì mô tả bằng automat. Vì vậy người ta thường dùng automat vào trong các phương pháp thiết kế số.

AUTOMAT HỮU HẠN

Trong chương 1 chúng ta chỉ mới giới thiệu các khái niệm cơ bản (ngôn ngữ, văn phạm, automat) một cách ngắn gọn và không hình thức. Để tiếp tục nghiên cứu chúng ta phải cung cấp các định nghĩa hình thức hơn, và bắt đầu phát triển các kết quả chặt chẽ trên chúng. Chúng ta sẽ bắt đầu với các accepter hữu hạn, là một trường hợp đặc biệt đơn giản của sơ đồ tổng quát được giới thiệu trong chương trước.

2.1. ACCEPTER HỮU HẠN ĐƠN ĐỊNH

1. Accepter đơn định và đồ thị chuyển trạng thái

Định nghĩa 2.1. *Một* ***accepter hữu hạn đơn định*** *hay dfa được định nghĩa bởi bộ năm*

$$M = (Q, \Sigma, \delta, q_0, F),$$

trong đó: Q *– là một tập hữu hạn các* ***trạng thái nội*** *(internal states),*

Σ *– là một tập hữu hạn các ký hiệu được gọi là* ***bảng chữ cái ngõ nhập*** *(input alphabet),*

δ: $Q \times \Sigma \to Q$ *– là một hàm được gọi là* ***hàm chuyển trạng thái*** *(transition function). Để chuyển trạng thái automat dựa vào trạng thái hiện hành* $q \in Q$ *nó đang ở vào và kí hiệu nhập* $a \in \Sigma$ *nó đang đọc được, nó sẽ chuyển sang trạng thái kế được định nghĩa sẵn trong* δ.

$q_0 \in Q$ *– là* ***trạng thái khởi đầu*** *(initial state),*

$F \subseteq Q$ *– là một tập các* ***trạng thái kết thúc*** *(final states) (hay còn gọi là* ***trạng thái chấp nhận****).*

Hoạt động của một dfa

Một dfa hoạt động theo cách sau:

- Tại thời điểm khởi đầu, nó được giả thiết đang ở trong trạng thái khởi đầu q_0, với cơ cấu nhập (đầu đọc) của nó đang ở trên kí hiệu đầu tiên bên trái của chuỗi nhập.
- Trong suốt mỗi lần di chuyển (*move*) của automat, cơ cấu nhập tiến về phía phải một kí hiệu, như vậy mỗi lần di chuyển sẽ lấy một kí hiệu ngõ nhập.
- Khi gặp kí hiệu kết thúc chuỗi, chuỗi là được chấp nhận (*accept*) nếu automat đang ở vào một trong các trạng thái kết thúc của nó. Ngược lại thì có nghĩa là chuỗi bị từ chối.

Để biểu diễn một cách trực quan cho dfa người ta sử dụng đồ thị chuyển trạng thái.

Đồ thị chuyển trạng thái (*transition graph*)

- Các đỉnh biểu diễn các trạng thái.
- Các cạnh biểu diễn các chuyển trạng thái.
- Các nhãn trên các đỉnh là tên các trạng thái.
- Các nhãn trên các cạnh là giá trị hiện tại của kí hiệu nhập.
- Trạng thái khởi đầu sẽ được nhận biết bằng một mũi tên đi vào không mang nhãn mà không xuất phát từ bất kỳ đỉnh nào.
- Các trạng thái kết thúc được vẽ bằng một vòng tròn đôi (*double circle*).

Cách vẽ đồ thị chuyển trạng thái – ĐTCTT cho một dfa

Nếu $M = (Q, \Sigma, \delta, q_0, F)$ là một dfa, thì ĐTCTT tương ứng với nó G_M là

- Có đúng $|Q|$ đỉnh.
- Mỗi đỉnh mang một nhãn khác nhau $q_i \in Q$.
- Đối với mỗi qui tắc chuyển trạng thái $\delta(q_i, a) = q_j$, đồ thị có một cạnh
- (q_i, q_j) mang nhãn a.
- Đỉnh tương ứng với q_0 được gọi là ***đỉnh khởi đầu***.
- Những đỉnh mang nhãn $q_f \in F$ là những ***đỉnh kết thúc*** (hay ***đỉnh chấp nhận***)

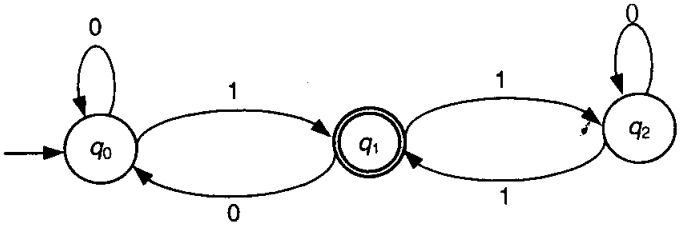
Ví dụ 2.1. Cho dfa sau: $M = (Q, \Sigma, \delta, q_0, F)$

trong đó: $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{0, 1\}$, $F = \{q_1\}$, còn δ được cho bởi

$$\delta(q_0, 0) = q_0, \qquad \delta(q_0, 1) = q_1,$$
$$\delta(q_1, 0) = q_0, \qquad \delta(q_1, 1) = q_2,$$
$$\delta(q_2, 0) = q_2, \qquad \delta(q_2, 1) = q_1,$$

ĐTCTT tương ứng biểu diễn cho dfa này được cho trong H.2.1.

Dễ thấy dfa này chấp nhận chuỗi 01 nhưng không chấp nhận chuỗi 00.



Hình 2.1

Nhận xét

Hàm chuyển trạng thái δ của dfa đã định nghĩa cho tính đơn định của dfa, vì ứng với một trạng thái q_i và một kí hiệu ngõ nhập a automat chỉ có một khả năng chuyển trạng thái duy nhất đi đến trạng thái q_j với $q_j = \delta(q_i, a)$.

Để thuận tiện trong một số tính toán và định nghĩa người ta đưa ra khái niệm:

Hàm chuyển trạng thái mở rộng *(extended transition function)*

Hàm chuyển trạng thái mở rộng (HCTTMR) δ^* *được định nghĩa một cách đệ qui như sau:*

$$\delta^*(q, \lambda) = q, \tag{2.1}$$

$$\delta^*(q, wa) = \delta(\delta^*(q, w), a), \tag{2.2}$$

đối với mọi $q \in Q$, $w \in \Sigma^*$, $a \in \Sigma$.

Ví dụ 2.2. Nếu

$$\delta(q_0, a) = q_1, \text{ và } \delta(q_1, b) = q_2,$$

thì

$$\delta^*(q_0, ab) = q_2,$$

2. Ngôn ngữ và dfa

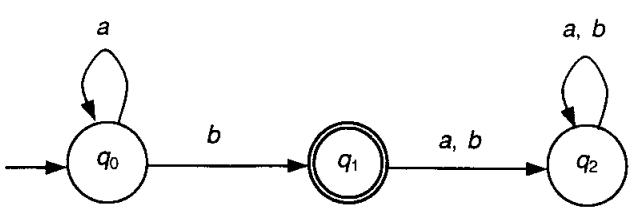
Định nghĩa 2.2. *Ngôn ngữ được chấp nhận bởi dfa* $M = (Q, \Sigma, \delta, q_0, F)$ *là tập tất cả các chuỗi trên* Σ *được chấp nhận bởi M. Biểu diễn hình thức bằng kí hiệu*

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \in F\}.$$

Nhận xét

$$\overline{L(M)} = \{w \in \Sigma^* : \delta^*(q_0, w) \notin F\},$$

Ví dụ 2.3. Xét dfa trong H.2.2



Hình 2.2

Theo kí hiệu tập hợp, ngôn ngữ được chấp nhận bởi automat trên là

$$L = \{a^n b : n \geq 0\}.$$

Lưu ý

1. Trong H.2.2 chúng ta cho phép sử dụng hai nhãn trên một cạnh đơn, biểu diễn cho hai chuyển trạng thái tương ứng với hai nhãn đó.

2. Trong hình này cũng giới thiệu cho chúng ta một trạng thái, q_2, có tính chất đặc biệt là automat sau khi đi vào nó thì ở luôn trong đó và không thoát ra được. Một trạng thái có tính chất như vậy được gọi là trạng thái bẫy.

Trạng thái bẫy *(trap state)*

Là trạng thái mà sau khi automat đi vào sẽ không bao giờ thoát ra được.

Chú ý

Trạng thái bẫy có thể là trạng thái kết thúc hoặc không.

Định nghĩa trên cũng có thể mở rộng ra cho nhóm các trạng thái bẫy kết thúc hay không kết thúc. Nhóm các trạng thái bẫy kết thúc là nhóm các trạng thái mà các trạng thái trong nhóm đều là trạng thái kết thúc và có tính bẫy, có nghĩa là automat sau khi đi vào nhóm thì sẽ không bao giờ thoát ra được khỏi nhóm. Và tương tự cho nhóm các trạng thái bẫy không kết thúc.

Các trạng thái bẫy và nhóm trạng thái bẫy kết thúc hay không kết thúc có một vài ý nghĩa quan trọng trong hoạt động của automat.

Nhận xét

ĐTCTT khá thuận lợi khi làm việc với automat hữu hạn. Để đảm bảo mọi lý luận dựa trên đồ thị cũng đúng như những lý luận dựa trên các tính chất của hàm δ và δ^* ta có định lý sau.

Định lý 2.1. *Cho* $M = (Q, \Sigma, \delta, q_0, F)$ *là một accepter hữu hạn đơn định, và* G_M *là đồ thị chuyển trạng thái tương ứng của nó, thì đối với mọi* $q_i, q_j \in Q$, *và* $w \in \Sigma^+$, $\delta^*(q_i, w) = q_j$ *nếu và chỉ nếu có trong* G_M *một con đường mang nhãn là* w *đi từ* q_i *đến* q_j.

Chứng minh

Sinh viên tự chứng minh bằng qui nạp trên chiều dài của w.

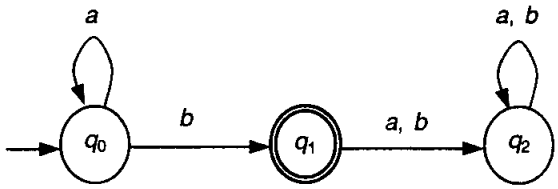
Một cách biểu diễn hữu hiệu khác cho các automat hữu hạn là bảng truyền.

Bảng truyền – Bảng chuyển trạng thái *(transition table)*

Là bảng trong đó các nhãn của hàng (ô tô đậm trên hàng trong H.2.3*) biểu diễn cho trạng thái hiện tại, còn nhãn của cột (ô tô đậm trên cột trong* H.2.3*) biểu diễn cho ký hiệu nhập hiện tại. Các điểm nhập (entry) trong bảng định nghĩa cho trạng thái kế tiếp.*

Ví dụ 2.4. Bảng truyền trong H.2.3 là tương đương với dfa trong H.2.2.

STATE	INPUT SYMBOL	
	a	*b*
q_0	q_0	q_1
q_1	q_2	q_2
q_2	q_2	q_2



Hình 2.3

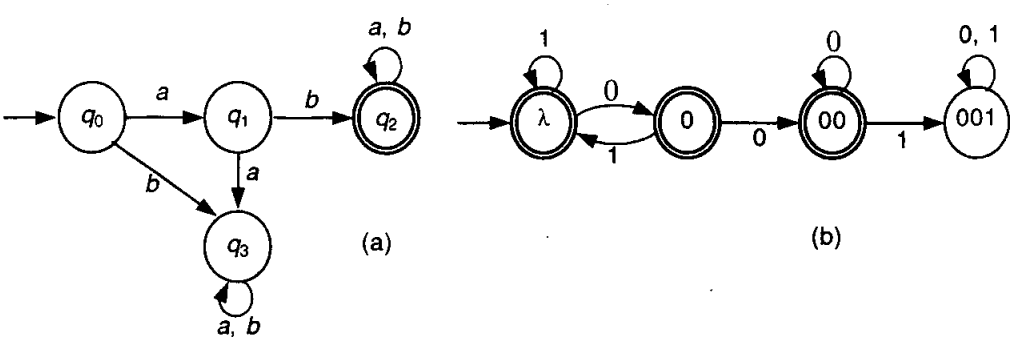
Nhận xét

Bảng truyền gợi ý cho chúng ta một cấu trúc dữ liệu để mô tả cho automat hữu hạn. Đồng thời cũng gợi ý cho chúng ta rằng một dfa có thể dễ dàng hiện thực được thành một chương trình máy tính; chẳng hạn bằng một dãy các phát biểu "if".

Ví dụ 2.5

1. Tìm một dfa chấp nhận tập tất cả các chuỗi trên $\Sigma = \{a, b\}$ được bắt đầu bằng chuỗi *ab*. Dfa kết quả được cho trong H.2.4(a)

2. Tìm một dfa chấp nhận tập tất cả các chuỗi trên $\Sigma = \{0, 1\}$, ngoại trừ những chuỗi chứa chuỗi con 001. Dfa kết quả được cho trong H.2.4(b)



Hình 2.4

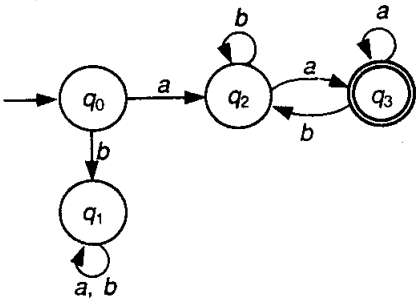
3. Ngôn ngữ chính qui (*regular languages – rl*)

Mỗi automat hữu hạn chấp nhận một ngôn ngữ nào đó. Vậy tương ứng với lớp automat ta sẽ có một lớp ngôn ngữ mà người ta gọi lớp ngôn ngữ này lớp ngôn ngữ chính qui – NNCQ (hay họ NNCQ).

Định nghĩa 2.3. *Một ngôn ngữ L được gọi là chính qui nếu và chỉ nếu tồn tại một accepter hữu hạn đơn định M nào đó sao cho L = L(M).*

Ví dụ 2.6. Chứng minh rằng ngôn ngữ $L = \{awa : w \in \{a,b\}^*\}$ là chính qui.

Chứng minh bằng cách xây dựng một dfa cho nó. Một dfa như vậy được cho trong H.2.5.



Hình 2.5

2.2. ACCEPTER HỮU HẠN KHÔNG ĐƠN ĐỊNH

1. Định nghĩa của một accepter không đơn định

Định nghĩa 2.4. *Một accepter hữu hạn không đơn định hay nfa được định nghĩa bằng bộ năm*

$$M = (Q, \Sigma, \delta, q_0, F),$$

trong đó: Q, Σ, q_0, F *– được định nghĩa như đối với accepter hữu hạn đơn định còn*

$$\delta: Q \times (\Sigma \cup \{\lambda\}) \to 2^Q$$

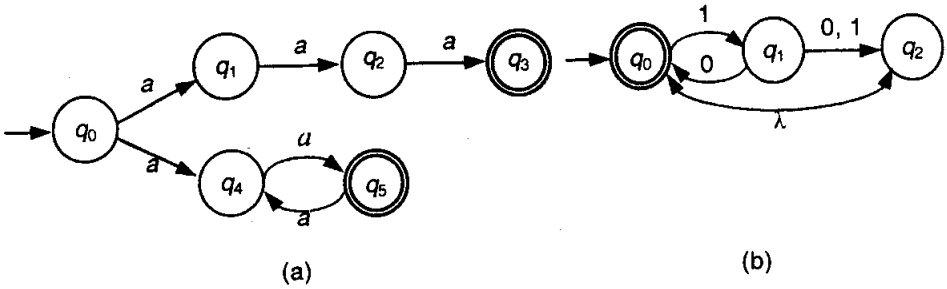
Nhận xét

Có hai khác biệt chính giữa định nghĩa này và định nghĩa của một dfa. Đối với nfa miền giá trị của δ là tập 2^Q, vì vậy giá trị của nó không còn là một phần tử đơn của Q, mà là một tập con của nó và đặc biệt có thể là ∅.

Thứ hai định nghĩa này cho phép λ như là một đối số thứ hai của δ. Điều này có nghĩa là nfa có thể thực hiện một sự chuyển trạng thái mà không cần phải lấy vào một kí hiệu nhập nào.

Tương tự như dfa, một nfa cũng có thể được biểu diễn bằng một ĐTCTT.

Ví dụ 2.7. Hai automat vẽ trong H.2.6 là các nfa theo định nghĩa ở trên.



Hình 2.6

Nfa đầu tiên có hai cạnh chuyển trạng thái mang nhãn a đi ra từ trạng thái q_0, còn nfa thứ hai có cạnh chuyển trạng thái–λ.

Chú ý

Một nfa có thể không định nghĩa cho một chuyển trạng thái $\delta(q_i, a)$ nào đó như nfa trong H.2.6(b) chỉ ra, nó không định nghĩa cho chuyển trạng thái $\delta(q_2, 0)$. Người ta gọi trường hợp này là một ***cấu hình chết*** (*dead configuration*), và có thể hình dung trong trường hợp này là automat dừng lại, không hoạt động nữa. Về mặt hình thức nó được hiểu như là $\delta(q_2, 0) = \varnothing$.

Hàm chuyển trạng thái mở rộng

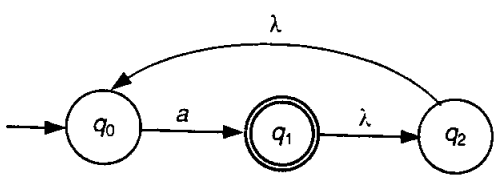
Tương tự, ở đây hàm chuyển trạng thái mở rộng cũng có thể được mở rộng sao cho đối số thứ hai của nó là một chuỗi.

Định nghĩa 2.5. *Cho một nfa, hàm chuyển trạng thái mở rộng được định nghĩa sao cho $\delta^*(q_i,w)$ chứa q_j nếu và chỉ nếu có một con đường trong ĐTCTT từ q_i đến q_j mang nhãn w. Điều này đúng với mọi $q_i, q_j \in Q$ và $w \in \Sigma^*$.*

Ví dụ 2.8. Xét ĐTCTT trong H2.7 ta có

$$\delta^*(q_1, a) = \{q_0, q_1, q_2\},$$
$$\delta^*(q_2, \lambda) = \{q_0, q_2\},$$

Chúng ta sẽ đề cập đến giải thuật tính δ^* trong trường hợp này ở phần sau.



Hình 2.7

Tương tự như bên dfa một chuỗi được chấp nhận bởi một nfa nếu có bất kỳ một dãy các chuyển trạng thái có thể mà sẽ đặt automat vào trạng thái kết thúc khi đọc đến hết chuỗi.

Định nghĩa 2.6. *Ngôn ngữ được chấp nhận bởi nfa $M = (Q, \Sigma, \delta, q_0, F)$, được định nghĩa như là một tập tất cả các chuỗi được chấp nhận bởi nfa trên. Một cách hình thức,*

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \cap F \neq \varnothing\}.$$

Ví dụ 2.9. Ngôn ngữ được chấp nhận bởi automat cho trong H.2.6(b) là

$$L = \{(10)^n : n \geq 0\}$$

Ghi chú: trong quá trình tính toán δ^* thường xuất hiện một dạng mở rộng khác của hàm δ và δ^* đó là sự mở rộng trên đối số thứ nhất của

δ^* thay vì nó là một trạng thái nó trở thành một tập trạng thái. Cụ thể với một tập con T nào đó của Q thì ta định nghĩa:

$$\delta(T,a) = \bigcup_{q\in T}\delta(q,a)$$

$$\delta^*(T,a) = \bigcup_{q\in T}\delta^*(q,a)$$

$$\delta^*(T,\lambda) = \bigcup_{q\in T}\delta^*(q,\lambda)$$

Người ta thường hiện thực trong các chương trình máy tính cách tính các hàm này $\delta(q, a)$, $\delta(T, a)$, $\delta^*(q, \lambda)$, $\delta^*(T, \lambda)$ lần lượt bằng các hàm **move(q, a)**, **move(T, a)**, **λ–closure(q)**, **λ–closure(T)** (**λ–closure** đọc là bao đóng–λ)

Từ định nghĩa trên:

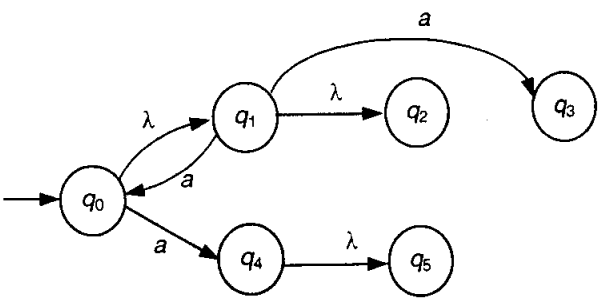
Để tính $\delta^*(q, a)$ và $\delta^*(T, a)$ ta có công thức sau:

$$\delta^*(q, a) = \textbf{λ–closure(move(λ–closure}(q), a))$$

$$\delta^*(T, a) = \textbf{λ–closure(move(λ–closure}(T), a))$$

Một tham khảo chi tiết hơn về các hàm λ – closure() và move() có thể xem trong [4] chương 3 trang 118.

Ví dụ 2.10. Cho nfa như trong H.2.8, có bảng truyền tương ứng ở bên cạnh, hãy tính $\delta^*(q_0, a)$.



Bảng truyền		
	a	λ
q_0	q_4	q_1
q_1	q_0, q_3	q_2
q_2		
q_3		
q_4		q_5
q_5		

Hình 2.8

Theo công thức ta có:

$$\delta^*(q_0, a) = \lambda\text{–closure(move(}\lambda\text{–closure}(q_0), a))$$

Dựa vào bảng truyền ta tính được:

$$\lambda\text{–closure}(q_0) = \{ q_0, q_1, q_2\}$$

$$\text{move}(\{ q_0, q_1, q_2\}, a) = \{ q_4, q_0, q_3\}$$

$$\lambda\text{–closure}(\{ q_4, q_0, q_3\}) = \{ q_4, q_0, q_3, q_5, q_1, q_2\}$$

Vậy $\delta^*(q_0, a) = \{ q_0, q_1, q_2, q_3, q_4, q_5\}$

Một định nghĩa khác về dfa – dfa mở rộng

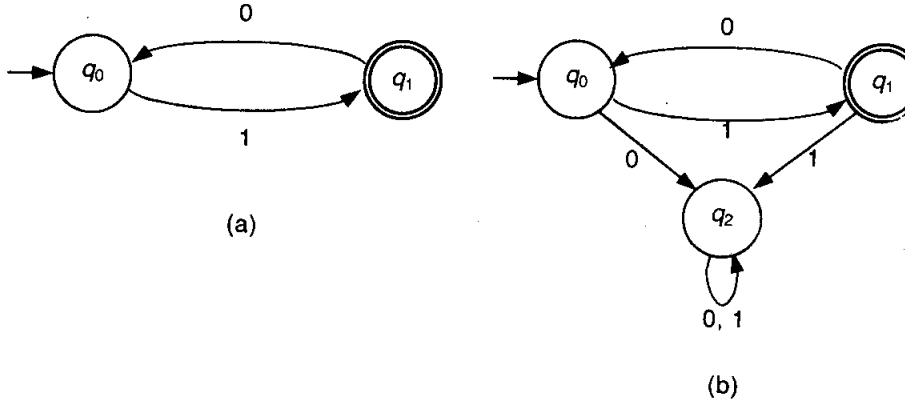
Một dfa là một trường hợp đặc biệt của một nfa trong đó:

1. *Không có chuyển trạng thái–rỗng,*
2. *Đối với mỗi trạng thái q và một kí hiệu nhập a, có tối đa một cạnh chuyển trạng thái đi ra khỏi q và có nhãn là a.*

Nhận xét

Về bản chất định nghĩa này và định nghĩa trước đây là tương đương nhau (cùng định nghĩa tính đơn định của dfa). Nó chỉ khác định nghĩa thứ nhất ở chỗ cho phép khả năng không có một sự chuyển trạng thái nào đối với một cặp trạng thái và kí hiệu nhập. Trong trường hợp này thì ta xem như nó rơi vào một trạng thái bẫy không kết thúc mà trạng thái này không được vẽ ra.

Ví dụ 2.11. Hình 2.9(a) biểu diễn một dfa trên bảng chữ cái {0, 1} theo định nghĩa thứ hai vừa nêu. Nó không định nghĩa cho chuyển trạng thái $\delta(q_0, 0)$, trong trường hợp này người ta bảo dfa rơi vào cấu hình chết tương tự như bên nfa. Và những chuỗi nào làm cho dfa rơi vào cấu hình chết trong khi xử lý nó là những chuỗi không được chấp nhận bởi dfa. Vì vậy về mặt ngôn ngữ được chấp nhận thì dfa ở trong H.2.9(a) tương đương với dfa ở trong H.2.9(b). (Khái niệm hai automat tương đương nhau được định nghĩa ở phần sau, tuy nhiên có thể hiểu đơn giản là chúng cùng chấp nhận một ngôn ngữ như nhau)



Hình 2.9

Nhận xét: dễ thấy dfa trong H.2.9(a) đơn giản hơn dfa ở trong H.2.9(b) mặc dù chúng cùng chấp nhận một ngôn ngữ như nhau.

Vậy dfa mở rộng và dfa dfa đầy đủ theo định nghĩa ban đầu thật sự là tương đương nhau và chúng chỉ ***khác nhau ở một trạng thái bẫy không kết thúc***.

2. Tại sao không đơn định?

Nhận xét

Các máy tính số là hoàn toàn đơn định; trạng thái của chúng tại một thời điểm bất kỳ là tiên đoán được một cách duy nhất từ chuỗi nhập và trạng thái bắt đầu.

Vậy thì tại sao chúng ta lại nghiên cứu không đơn định để làm gì?

Thứ nhất, trong nhiều bài toán có nhiều hướng giải quyết (nhiều cách đi), thì thông thường hướng giải quyết tốt nhất (cách đi tốt nhất) là không biết trước được (chẳng hạn trong nhiều chương trình chơi game), nhưng có thể tìm thấy được bằng cách sử dụng một sự tìm kiếm toàn diện có quay lui (*backtracking*). Khi có một vài khả năng lựa chọn có thể thay thế nhau được, chúng ta chọn một khả năng và đi theo nó cho đến khi xác định được nó là đã tốt nhất hay chưa. Nếu chưa chúng ta quay về lại điểm quyết định cuối cùng trước đó và khảo sát một cách chọn lựa khác. Một giải thuật mô phỏng quá trình tìm kiếm này là một giải thuật không đơn định. Và như chúng ta sẽ biết sau này, chúng ta có thể biến đổi một giải thuật không đơn định này thành một giải thuật đơn định tương đương và tất nhiên trong giải thuật đơn định này sẽ không có sự quay lui.

Thứ hai, không đơn định đôi khi còn rất hữu hiệu trong việc giải quyết các bài toán một cách dễ dàng. Chẳng hạn có một số bài toán thì việc tìm nfa có vẻ tự nhiên và đơn giản hơn việc tìm một dfa cho nó. Như sau này chúng ta sẽ thấy một số ví dụ khác, thuyết phục hơn về sự hiệu quả của tính không đơn định.

Cùng giống như vậy, không đơn định là một cơ chế hữu hiệu để mô tả một vài ngôn ngữ một cách súc tích. Chú ý rằng định nghĩa của một văn phạm có bao gồm yếu tố không đơn định. Trong

$$S \rightarrow aSb|\lambda$$

chúng ta tại một điểm bất kỳ có thể chọn hoặc luật sinh thứ nhất hoặc luật sinh thứ hai để áp dụng.

Cuối cùng có một lý do kỹ thuật để giới thiệu cơ chế không đơn định. Như chúng ta sẽ thấy, một vài kết quả là dễ dàng được chứng minh đối với nfa hơn là đối với dfa. Và như sau này chúng ta sẽ thấy không có một sự khác biệt quan trọng nào giữa hai loại automat này. Vì vậy, việc cho phép cơ chế không đơn định thường làm đơn giản hóa các lý luận hình thức mà không ảnh hưởng tới tính tổng quát của kết luận.

2.3. SỰ TƯƠNG ĐƯƠNG GIỮA ACCEPTER HỮU HẠN ĐƠN ĐỊNH VÀ ACCEPTER HỮU HẠN KHÔNG ĐƠN ĐỊNH

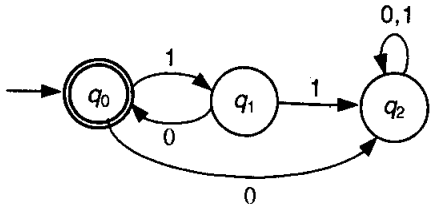
Dfa và nfa khác nhau về cái gì?

Mặc dù có sự khác nhau trong định nghĩa của chúng, nfa được định nghĩa tổng quát hơn dfa, như vậy về sức mạnh nfa có khả năng mạnh hơn dfa. Nhưng có thật là như vậy không? Có thật là có một sự khác biệt quan trọng giữa chúng? Để nghiên cứu trước hết chúng ta định nghĩa:

Sự tương đương giữa hai automat

Chúng ta nói rằng hai accepter là tương đương nếu chúng chấp nhận cùng một ngôn ngữ như nhau.

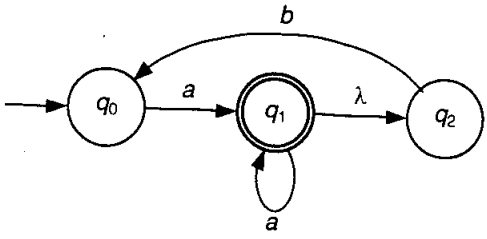
Ví dụ 2.12. Dfa được trình bày ở H.2.10 là tương đương với nfa ở H.2.6(b) vì cả hai chúng đều chấp nhận ngôn ngữ $\{(10)^n : n \geq 0\}$.



Hình 2.10

Nhận xét: dfa về bản chất là một loại giới hạn của nfa, nên lớp các dfa là một lớp con của lớp nfa. Nhưng nó có phải là một lớp con thực sự hay không? Rất hay là người ta đã chứng minh được rằng hai lớp này là tương đương nhau, tức là với một nfa thì sẽ có một dfa tương đương với nó.

Ví dụ 2.13. Biến đổi nfa trong H.2.11 (bảng truyền tương ứng ở bên cạnh) thành dfa tương đương.



Bảng truyền			
	a	*b*	λ
q_0	q_1		
q_1	q_1		q_2
q_2		q_0	

Hình 2.11

Để xây dựng một dfa tương đương ta sẽ tìm cách mô phỏng lại quá trình chấp nhận một chuỗi bất kỳ của nfa trên.

Tại thời điểm khởi đầu nfa có thể ở vào tập trạng thái có thể sau:

$$\delta^*(q_0, \lambda) = \{q_0\}$$

Từ tập trạng thái khởi đầu này ta sẽ mô phỏng lại quá trình xử lý các kí hiệu nhập của nfa ở vào từng thời điểm.

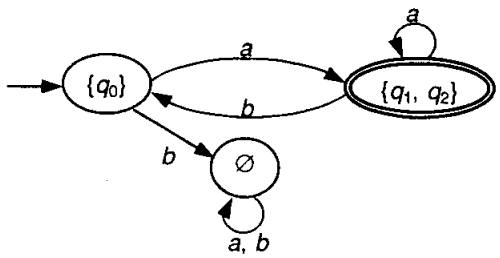
Trước hết ta mô phỏng cho tập trạng thái khởi đầu, ta có:

$$\delta^*(\{q_0\}, a) = \{q_1, q_2\}$$
$$\delta^*(\{q_0\}, b) = \varnothing$$

Lấy các tập trạng thái kết quả trên và tiếp tục mô phỏng cho chúng, ta có:

$$\delta^*(\{q_1, q_2\}, a) = \{q_1, q_2\}$$
$$\delta^*(\{q_1, q_2\}, b) = \{q_0\}$$

Đến đây ta thấy không có thêm tập trạng thái nào nữa để mô phỏng. Do đó ta vẽ được dfa tương đương như trong hình dưới đây. Chú ý để xác định trạng thái kết thúc, ta dựa vào định nghĩa chấp nhận chuỗi của nfa rằng: nfa chấp nhận chuỗi w nếu sau khi xử lý xong w tồn tại một trạng thái trong tập trạng thái mà nfa rơi vào là trạng thái kết thúc. Vì vậy bất kỳ tập trạng thái nào tìm được trong các bước trên có chứa một trạng thái kết thúc của nfa thì nó trở thành trạng thái kết thúc của dfa. Ở đây đó là trạng thái $\{q_1, q_2\}$ là vì nó có chứa trạng thái kết thúc q_1 của nfa.



Hình 2.12

Định lý 2.2. *Cho L là ngôn ngữ được chấp nhận bởi một accepter hữu hạn không đơn định $M_N = (Q_N, \Sigma, \delta_N, q_0, F_N)$, thì tồn tại một accepter hữu hạn đơn định $M_D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ sao cho $L = L(M_D)$.*

Chứng minh

Bằng cách sử dụng thủ tục nfa–to–dfa bên dưới. Thủ tục này có hai phiên bản, phiên bản thứ hai là một sự cải tiến trên phiên bản đầu. Vì vậy các bạn nên sử dụng phiên bản thứ hai khi ứng dụng.

Thủ tục*: *nfa_to_dfa (*phiên bản* 1)

Input : nfa $M = (Q, \Sigma, \delta_N, q_0, F_N)$

Output : ĐTCTT G_D của dfa

1. Tạo một đồ thị G_D với đỉnh khởi đầu là $\{q_0\}$.
2. Lặp lại các bước sau cho đến khi không còn cạnh nào thiếu.
   - Lấy một đỉnh bất kỳ $\{q_i, q_j, ..., q_k\}$ của G_D mà có một cạnh còn chưa được định nghĩa đối với một $a \in \Sigma$ nào đó.
   - Tính $\delta_N^*(\{q_i, q_j, ..., q_k\}, a)$.
   - Tạo một đỉnh cho G_D có nhãn $\{q_l, q_m, ..., q_n\}$ nếu nó chưa tồn tại.
   - Thêm vào G_D một cạnh từ $\{q_i, q_j, ..., q_k\}$ đến $\{q_l, q_m, ..., q_n\}$ và gán nhãn cho nó bằng a.
3. Mỗi trạng thái của G_D mà nhãn của nó chứa một q_f bất kỳ $\in F_N$ thì được coi là một đỉnh kết thúc.
4. Nếu M_N chấp nhận λ, thì đỉnh $\{q_0\}$ trong G_D cũng được làm thành một đỉnh kết thúc.

Thủ tục: nfa_to_dfa *(phiên bản 2)*

Input : nfa $M = (Q, \Sigma, \delta_N, q_0, F_N)$

Output : ĐTCTT G_D của dfa

Phiên bản này cải tiến trên việc tạo trạng thái khởi đầu cho dfa.

1. Tạo một đồ thị G_D với đỉnh khởi đầu là tập $\delta_N^*(q_0, \lambda)$.
2. Lặp lại các bước sau cho đến khi không còn cạnh nào thiếu.
   – Lấy một đỉnh bất kỳ $\{q_i, q_j, ..., q_k\}$ của G_D mà có một cạnh còn chưa được định nghĩa đối với một $a \in \Sigma$ nào đó.
   - Tính $\delta_N^*(\{q_i, q_j, ..., q_k\}, a) = \{q_l, q_m, ..., q_n\}$.
   - Tạo một đỉnh cho G_D có nhãn $\{q_l, q_m, ..., q_n\}$ nếu nó chưa tồn tại.
   - Thêm vào G_D một cạnh từ $\{q_i, q_j, ..., q_k\}$ đến $\{q_l, q_m, ..., q_n\}$ và gán nhãn cho nó bằng a.
3. Mỗi trạng thái của G_D mà nhãn của nó chứa một q_f bất kỳ $\in F_N$ thì được coi là một đỉnh kết thúc.

Nhận xét

1. Mặc dù phiên bản thứ hai có khác và đơn giản hơn phiên bản thứ nhất nhưng về bản chất chúng là như nhau, cùng tạo ra các dfa kết quả như nhau.
2. Khi thực hiện hai giải thuật trên thì chú ý kết quả sau:

   Nếu T là một tập trạng thái, và:

$$\lambda\text{–closure}(T) = T_1 \text{ thì } \lambda\text{–closure}(T_1) = T_1$$

hay nói cách khác:

$$\lambda\text{–closure}(\lambda\text{–closure}(T)) = \lambda\text{–closure}(T).$$

Kỹ thuật xây dựng ở trên được gọi là ***kỹ thuật mô phỏng***. Đây cũng là kỹ thuật được sử dụng khá phổ biến trong tài liệu này. Và để chứng minh hai đối tượng được đề cập trong kỹ thuật này là tương đương, người ta thường chứng minh bằng phương pháp qui nạp. Và đây cũng là kỹ thuật chứng minh ***đặc trưng*** cho phương pháp này. Sinh viên tự thực hiện một chứng minh qui nạp cho định lý trên.

Ví dụ 2.14. Biến đổi nfa trong H.2.13, có bảng truyền tương ứng ở bên cạnh thành dfa tương đương.

Ta sẽ sử dụng phiên bản thứ hai của thủ tục nfa–to–dfa để biến đổi.

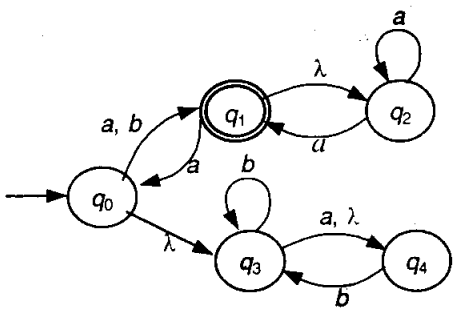
Đầu tiên ta có:

$$\delta^*(q_0, \lambda) = \lambda\text{–closure}(q_0) = \{q_0, q_3, q_4\}$$

$$\delta^*(\{q_0, q_3, q_4\}, a) = \lambda\text{–closure}(\text{move}(\lambda\text{–closure}(q_0, q_3, q_4]), a)$$

$$= \lambda\text{–closure}(\text{move}(\{q_0, q_3, q_4\}, a)$$

$$= \lambda\text{–closure}(\{q_1, q_4\}) = \{q_1, q_2, q_4\}$$



Bảng truyền			
	a	b	λ
q_0	q_1	q_1	q_3
q_1	q_0		q_2
q_2	q_1, q_2		
q_3	q_4	q_3	q_4
q_4		q_3	

Hình 2.13

Tiếp tục theo kiểu này cuối cùng ta có kết quả sau:

$\delta^*(q_0, \lambda) = \{q_0, q_3, q_4\}$

$\delta^*(\{q_0, q_3, q_4\}, a) = \{q_1, q_2, q_4\}$

$\delta^*(\{q_1, q_2, q_4\}, a) = \{q_0, q_1, q_2, q_3, q_4\}$

$\delta^*(\{q_1, q_2, q_3, q_4\}, a) = \{q_0, q_1, q_2, q_3, q_4\}$

$\delta^*(\{q_0, q_1, q_2, q_3, q_4\}, a) = \{q_0, q_1, q_2, q_3, q_4\}$

$\delta^*(\{q_3, q_4\}, a) = \{q_4\}$

$\delta^*(\{q_0, q_1, q_2, q_3, q_4\}, a) = \{q_0, q_1, q_2, q_3, q_4\}$

$\delta^*(\{q_4\}, a) = \varnothing$

$\delta^*(\{q_0, q_3, q_4\}, b) = \{q_1, q_2, q_3, q_4\}$

$\delta^*(\{q_1, q_2, q_4\}, b) = \{q_3, q_4\}$

$\delta^*(\{q_1, q_2, q_3, q_4\}, b) = \{q_3, q_4\}$

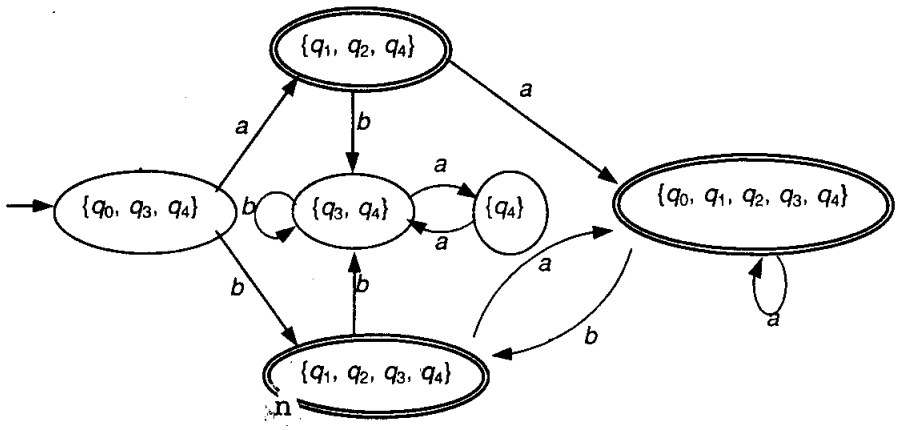
$\delta^*(\{q_0, q_1, q_2, q_3, q_4\}, b) = \{q_1, q_2, q_3, q_4\}$

$\delta^*(\{q_3, q_4\}, b) = \{q_3, q_4\}$

$\delta^*(\{q_0, q_1, q_2, q_3, q_4\}, b) = \{q_1, q_2, q_3, q_4\}$

$\delta^*(\{q_4\}, a) = \{q_3, q_4\}$

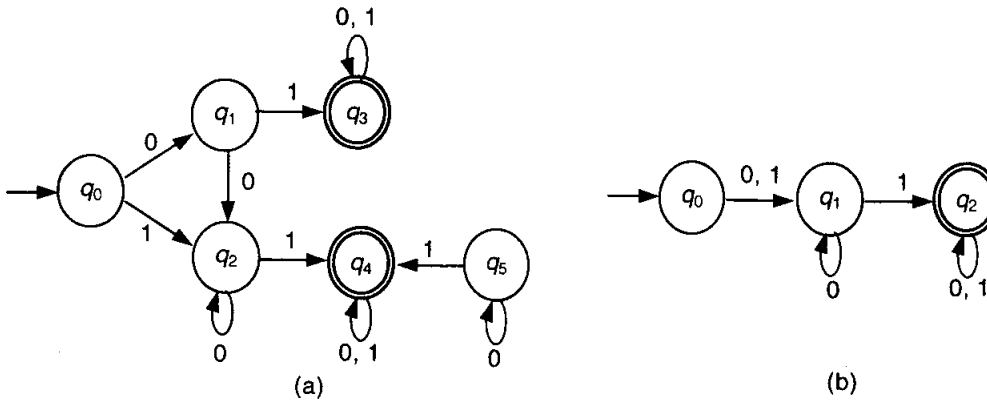
Dfa kết quả được cho trong H.2.14



Hình 2.14

2.4. RÚT GỌN SỐ TRẠNG THÁI CỦA MỘT AUTOMAT HỮU HẠN

Ví dụ 2.15. Hai dfa được vẽ trong H.2.15(a) và H.2.15(b) là tương đương nhau



Hình 2.15

Nhận xét

1. Trong H.2.15(a) có một trạng thái đặc biệt, trạng thái q_5, nó là trạng thái không đạt tới được từ trạng thái khởi đầu, người ta gọi nó là trạng thái không đạt tới được.

Trạng thái không đạt tới được *(inaccessible state)*

Là trạng thái mà không tồn tại con đường đi từ trạng thái khởi đầu đến nó.

2. Những trạng thái không đạt tới được (TTKĐTĐ) có thể xóa bỏ đi (kèm với các cạnh chuyển trạng thái liên quan tới nó) mà không làm ảnh hưởng tới ngôn ngữ được chấp nhận bởi automat.

3. Các chuyển trạng thái từ sau đỉnh q_1 và q_2 "có vẻ giống nhau", đối xứng nhau và automat thứ hai "có vẻ như" kết hợp hai phần này. Từ đây dẫn tới định nghĩa hai trạng thái giống nhau hay không phân biệt được cái mà chúng ta hi vọng sẽ có thể gom chung chúng lại thành một, như vậy chúng ta đã thực hiện được sự rút gọn trạng thái. Khái niệm ***giống nhau*** được định nghĩa tổng quát dựa trên việc: *với mọi chuỗi nếu xuất phát từ hai trạng thái này thì kết quả về mặt chấp nhận chuỗi là giống nhau tức là hoặc cùng rơi vào trạng thái kết thúc, hoặc không cùng rơi vào trạng thái kết thúc.* Như vậy hai trạng thái này có thể gom chung lại với nhau mà kết quả chấp nhận chuỗi không thay đổi. Mà như vậy thì có nghĩa ta đã rút gọn được trạng thái. Để thực hiện ý tưởng này chúng ta định nghĩa chặt chẽ hơn khái niệm này qua định nghĩa sau:

Định nghĩa 2.7. *Hai trạng thái p và q của một dfa được gọi là* ***không phân biệt được*** *(indistinguishable) hay* ***giống nhau*** *nếu với mọi* $w \in \Sigma^*$

$$\delta(q, w) \in F \text{ suy ra } \delta(p, w) \in F,$$

và

$$\delta(q, w) \notin F \text{ suy ra } \delta(p, w) \notin F,$$

Còn nếu tồn tại một chuỗi w nào đó $\in \Sigma^*$ *sao cho*

$$\delta(q, w) \in F \text{ còn } \delta(p, w) \notin F,$$

hay ngược lại thì p và q được gọi là ***phân biệt được*** *(distinguishable) hay khác nhau bởi chuỗi w.*

Nhận xét: một trạng thái kết thúc và một trạng thái không kết thúc không thể giống nhau được.

Chú ý: quan hệ giống nhau là một ***quan hệ tương đương***. Vì vậy quan hệ này sẽ phân hoạch tập trạng thái Q thành các tập con rời nhau, mỗi tập con bao gồm các trạng thái giống nhau.

Để xác định các cặp trạng thái không phân biệt được, người ta thực hiện ngược lại là xác định các cặp trạng thái không giống nhau trước từ đó cũng sẽ suy ra được điều chúng ta muốn. Để làm điều này người ta sử dụng thủ tục ***mark*** (đánh dấu) bên dưới.

Thủ tục: mark

Input: các cặp trạng thái, gồm ($|Q|X$ ($|Q|$ −1)/ 2) cặp, của dfa đầy đủ

Output: các cặp trạng thái được đánh dấu.

1. Loại bỏ tất cả các TTKĐTĐ. Dùng giải thuật trong lý thuyết đồ thị để xác định các đỉnh không có đường đi tới nó từ đỉnh khởi đầu.

2. Xét tất cả các cặp trạng thái (p, q). Nếu $p \in F$ và $q \notin F$ hay ngược lại, đánh dấu cặp (p, q) là phân biệt được. Các cặp trạng thái được đánh dấu ở bước này sẽ được ghi là đánh dấu ở bước số 0 (gọi là bước cơ bản).

3. Lặp lại bước sau cho đến khi không còn cặp nào không được đánh dấu trước đó được đánh dấu ở bước này.

- Đối với mọi cặp (p, q) chưa được đánh dấu và mọi $a \in \Sigma$, tính $\delta(p, a) = p_a$ và $\delta(q, a) = q_a$. Nếu cặp (p_a, q_a) đã được đánh dấu là phân biệt được, thì đánh dấu (p, q) là phân biệt được. Các cặp được đánh dấu ở bước này sẽ được ghi là được đánh dấu ở bước thứ i nếu đây là lần thứ i băng qua vòng lặp.

Định lý 2.3. *Thủ tục mark, áp dụng cho một dfa bất kỳ*

$$M = (Q, \Sigma, \delta, q_0, F),$$

kết thúc và xác định tất cả các trạng thái phân biệt được.

Chứng minh

Để chứng minh định lý này ta chủ yếu chứng minh ba bổ đề sau.

Bổ đề 2.1. *Cặp trạng thái q_i và q_j là phân biệt được bằng chuỗi có độ dài n, nếu và chỉ nếu có các chuyển trạng thái*

$$\delta(q_i, a) = q_k$$

và

$$\delta(q_j, a) = q_l$$

với một a nào đó $\in \Sigma$, và q_k và q_l là cặp trạng thái phân biệt được bằng chuỗi có độ dài n–1.

Bổ đề này được chứng minh dựa vào định nghĩa phân biệt được và không phân biệt được.

Bổ đề 2.2. *Khi băng qua vòng lặp trong bước 3 lần thứ n, thủ tục sẽ đánh dấu được thêm tất cả các cặp trạng thái phân biệt được bằng chuỗi có độ dài n.*

Bổ đề này được chứng minh qui nạp theo n với chú ý n = 0 tương ứng được thực hiện ở bước cơ sở.

Bổ đề 2.3. *Nếu thủ tục dừng lại sau n lần băng qua vòng lặp trong bước 3, thì không có cặp trạng thái nào của dfa mà phân biệt được bằng chuỗi có chiều dài lớn hơn n.*

Bổ đề này được chứng minh bằng phản chứng với sự kết hợp của bổ đề đầu. (Sinh viên tự chứng minh)

Thủ tục: reduce *(thu gọn)*

Input : dfa $M = (Q, \Sigma, \delta, q_0, F)$

Output : dfa tối giản $\hat{M} = (\hat{Q}, \Sigma, \hat{\delta}, \hat{q_0}, \hat{F})$.

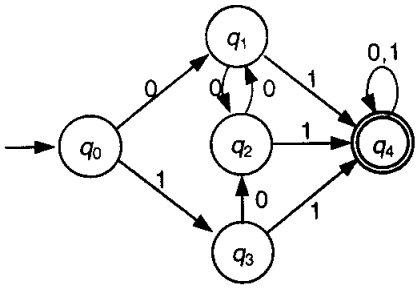
1. Sử dụng thủ tục ***mark*** để tìm tất cả các cặp trạng thái phân biệt được. Từ đây tìm ra các tập của tất cả các trạng thái không phân biệt được, gọi là $\{q_i, q_j, ..., q_k\}$, $\{q_l, q_m, ..., q_n\}$, ... như được mô tả ở trên.
2. Đối với mỗi tập $\{q_i, q_j, ..., q_k\}$ các trạng thái không phân biệt được như thế, tạo ra một trạng thái có nhãn $ij ... k$ cho $\hat{M}$.
3. Đối với mỗi quy tắc chuyển trạng thái của M có dạng

$$\delta(q_r, a) = q_p,$$

tìm các tập mà q_r và q_p thuộc về. Nếu $q_r \in \{q_i, q_j, ..., q_k\}$ và $q_p \in \{q_l, q_m, ..., q_n\}$, thì thêm vào $\hat{\delta}$ quy tắc $\hat{\delta}(ij ... k, a) = lm ... n$.

4. Trạng thái khởi đầu $\hat{q_0}$ là trạng thái của $\hat{M}$ mà nhãn của nó có chứa 0.
5. $\hat{F}$ là tập tất cả các trạng thái mà nhãn của nó chứa i sao cho $q_i \in F$.

Ví dụ 2.16. Rút gọn trạng thái của dfa sau với bảng truyền tương ứng bên cạnh:



Bảng truyền		
	0	1
q_0	q_1	q_3
q_1	q_2	q_4
q_2	q_1	q_4
q_3	q_2	q_4
q_4	q_4	q_4

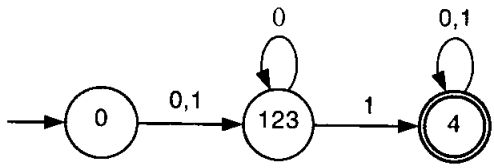
Hình 2.16

Dựa vào thủ tục ***mark*** ta đánh dấu được các cặp trạng thái phân biệt (bằng dấu √) mà kết quả được cho trong bảng sau:

(q_0, q_1) √ 1	(q_0, q_3) √ 1	**(q_1, q_2)**	(q_1, q_4) √ 0	(q_2, q_4) √ 0
(q_0, q_2) √ 1	(q_0, q_4) √ 0	**(q_1, q_3)**	**(q_2, q_3)**	(q_4, q_4) √ 0

Ghi chú: các con số ghi bên cạnh các dấu √ cho biết cặp đó được đánh dấu ở lần lặp thứ mấy khi băng qua vòng lặp ở bước 3 trong thủ tục ***mark***.

Từ kết quả này ta có các trạng thái q_1, q_2, q_3 là các trạng thái không phân biệt được, từ đây suy ra được dfa rút gọn như trong H.2.17:



Hình 2.17

Định lý 2.4. *Cho một dfa M bất kỳ, áp dụng thủ tục* ***reduce*** *tạo ra một dfa* $\hat{M}$ *khác sao cho*

$$L(M) = L(\hat{M})$$

Hơn nữa $\hat{M}$ *là tối giản theo nghĩa không có một dfa nào khác có số trạng thái nhỏ hơn mà cũng chấp nhận L(M).*

Chứng minh

Sinh viên tự chứng minh phần $L(M) = L(\hat{M})$. Ở đây chỉ chứng minh $\hat{M}$ là dfa tối giản.

Giả thiết, dfa $\hat{M}$ có $\hat{Q}=\{p_0, p_1, ..., p_m\}$ với p_0 là trạng thái khởi đầu. Giả sử tồn tại dfa M_1 tương đương với $\hat{M}$ và có số trạng thái ít hơn, với hàm chuyển trạng thái là δ_1 và trạng thái khởi đầu là q_0. Vì các trạng thái trong $\hat{M}$ là các trạng thái đạt tới được suy ra tồn tại các chuỗi phân biệt $w_0, w_1, ..., w_m$ sao cho:

$$\hat{\delta}^*(p_0, w_i) = p_i,\ i = 0, 1, ..., m.\ (\text{ở đây } w_0 = \lambda)$$

Vì M_1 có ít trạng thái hơn $\hat{M}$ suy ra tồn tại hai chuỗi nào đó chẳng hạn w_k, w_l sao cho:

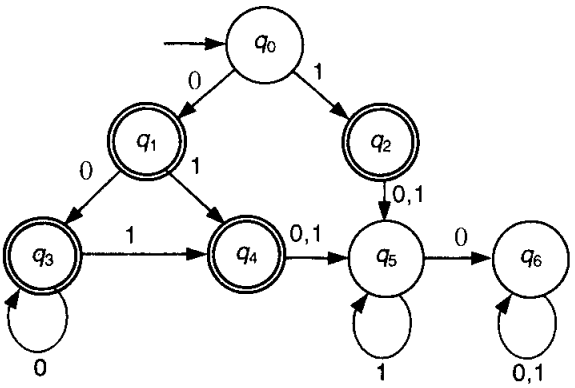
$$\delta_1^*(q_0, w_k) = \delta_1^*(q_0, w_l)$$

Vì p_k và p_l là phân biệt được nên tồn tại một chuỗi x nào đó sao cho $\hat{\delta}^*(p_0, w_kx) = \hat{\delta}^*(p_k, x)$ là $\in \hat{F}$ còn $\hat{\delta}^*(p_0, w_lx) = \hat{\delta}^*(p_l, x)$ là $\notin \hat{F}$. Hay nói cách khác $w_kx \in L(\hat{M})$ còn $w_lx \notin L(\hat{M})$. Mặt khác ta có

$$\delta_1^*(q_0, w_kx) = \delta_1^*(q_0, w_lx)$$

suy ra w_kx và w_lx hoặc cùng $\in L(M_1)$ hoặc cùng $\notin L(M_1)$. Suy ra M_1 và $\hat{M}$ không tương đương. Điều này mâu thuẫn với giả thiết. Vậy $\hat{M}$ là dfa có số trạng thái ít nhất.

Ví dụ 2.17. Rút gọn trạng thái của dfa sau có bảng truyền tương ứng bên cạnh:



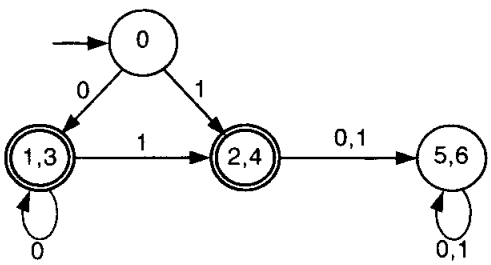
Bảng truyền		
	0	1
q_0	q_0	q_1
q_1	q_3	q_4
q_2	q_5	q_5
q_3	q_3	q_4
q_4	q_5	q_5
q_5	q_6	q_5
q_6	q_6	q_6

Hình 2.18

Kết quả đánh dấu các cặp trạng thái phân biệt:

(q_0, q_1) √ 0	(q_1, q_2) √ 0	(q_2, q_4)	(q_4, q_5) √ 0
(q_0, q_2) √ 0	(q_1, q_3)	(q_2, q_5) √ 0	(q_4, q_6) √ 0
(q_0, q_3) √ 0	(q_1, q_4) √ 1	(q_2, q_6) √ 0	(q_5, q_6)
(q_0, q_4) √ 0	(q_1, q_5) √ 0	(q_3, q_4) √ 1	
(q_0, q_5) √ 1	(q_1, q_6) √ 0	(q_3, q_5) √ 0	
(q_0, q_6) √ 1	(q_2, q_3) √ 1	(q_3, q_6) √ 0	

Suy ra dfa rút gọn là



Hình 2.19

Chú ý: trong trường hợp muốn rút gọn dfa mở rộng thì trước hết phải biến đổi thành dfa đầy đủ trước khi rút gọn.

Tham khảo

Cho biết các hàm và thủ tục sau:

Nextchar : trả về kí tự nhập kế tiếp.

Move(T, a) : trả về tập trạng thái kế của tập trạng thái hiện hành T và kí tự nhập là a.

λ-*closure*(T) : trả về tập bao đóng của tập trạng thái T.

Chương trình mô phỏng một dfa

Input : một chuỗi x kết thúc bởi kí tự eof. Một DFA D có trạng thái khởi đầu q_0 và tập trạng thái kết thúc F.

Output : "yes" nếu D chấp nhận X, "no" nếu ngược lại.

```
q := q0;
c := nextchar;
while c ≠ eof do begin
        q:= move(q, c);
        c:= nextchar;
end;
if q is in F then
return "yes"
else return "no";
```

Chương trình mô phỏng một nfa

Input : một chuỗi X kết thúc bởi kí tự eof. Một NFA N có trạng thái khởi đầu q_0 và tập trạng thái kết thúc F.

Output : "yes" nếu N chấp nhận X, "no" nếu ngược lại.

```
S := λ–closure({q0});
c := nextchar;
while a ≠ eof do begin
        S := λ–closure(move(S, a));
        c := nextchar;
end;
if S ∩ F ≠ ∅ then
return "yes"
  else return "no";
```

NGÔN NGỮ CHÍNH QUI VÀ VĂN PHẠM CHÍNH QUI

Ngoài cách sử dụng accepter hữu hạn để mô tả NNCQ còn một số cách khác được dùng để mô tả NNCQ mà sẽ được giới thiệu ở trong chương này. Một trong số đó có một cách mô tả NNCQ khá súc tích và ngắn gọn đó chính là biểu thức chính qui.

3.1. BIỂU THỨC CHÍNH QUI (*REGULAR EXPRESSION*)

Biểu thức chính qui (BTCQ) là gì?

*Là một sự kết hợp các chuỗi kí hiệu của một bảng chữ cái Σ nào đó, các dấu ngoặc, và các phép toán +, ·, và *. trong đó phép + biểu thị cho phép hội, phép · biểu thị cho phép kết nối, phép * biểu thị cho phép bao đóng sao.*

Ví dụ

- Ngôn ngữ {***a***} được biểu thị bởi BTCQ *a*.

- Ngôn ngữ {***a, b, c***} được biểu thị bởi BTCQ ***a + b + c***.

- Ngược lại BTCQ $(a + b{\cdot}c)^*$ biểu thị cho ngôn ngữ {λ, ***a, bc, aa, abc, bca, bcbc, aaa, aabc***, . . .}.

1. Định nghĩa hình thức của một biểu thức chính qui

Định nghĩa 3.1

1. Cho Σ là một bảng chữ cái, thì ∅, λ, và a ∈ Σ tất cả đều là những BTCQ, hơn nữa chúng được gọi là những ***BTCQ nguyên thủy****.*

2. Nếu r_1 và r_2 là những BTCQ, thì $r_1 + r_2$, $r_1 r_2$, r_1^*, và (r_1) cũng vậy.

3. Một chuỗi là một BTCQ nếu và chỉ nếu nó có thể được dẫn xuất từ các BTCQ nguyên thủy bằng một số lần hữu hạn áp dụng các quy tắc trong (2).

Ví dụ 3.1. Cho Σ = {*a, b, c*}, thì chuỗi $(a + b{\cdot}c)^*{\cdot}(c + \varnothing)$ là BTCQ, vì nó được xây dựng bằng cách áp dụng các qui tắc ở trên. Còn (*a + b* +) không phải là BTCQ.

2. Ngôn ngữ tương ứng với biểu thức chính qui

Định nghĩa 3.2. *Ngôn ngữ L(r) được biểu thị bởi BTCQ bất kỳ là được định nghĩa bởi các qui tắc sau:*

1. ∅ là BTCQ biểu thị tập trống,

2. λ *là BTCQ biểu thị* $\{\lambda\}$,

3. *Đối với mọi* $a \in \Sigma$, a *là BTCQ biểu thị* $\{a\}$,

Nếu r_1 *và* r_2 *là những BTCQ, thì*

4. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$,

5. $L(r_1 \cdot r_2) = L(r_1)L(r_2)$,

6. $L((r_1)) = L(r_1)$,

7. $L(r_1^*) = (L(r_1))^*$.

Ví dụ 3.2. Trình bày ngôn ngữ $L(a^* \cdot (a + b))$ trong kí hiệu tập hợp.

$$\begin{aligned} L(a^* \cdot (a + b)) &= L(a^*)\, L(a + b) \\ &= (L(a))^*\, (L(a) \cup L(b)) \\ &= \{\lambda, a, aa, aaa, \ldots\}\{a, b\} \\ &= \{a, aa, aaa, \ldots, b, ab, aab, \ldots\} \end{aligned}$$

Nhận xét: định nghĩa trên chưa qui định về độ ưu tiên của các phép toán.

Ví dụ 3.3. Cho $r = a{\cdot}b + c$

$\Rightarrow$ $r = r_1 = (a \cdot b) + c$ tương ứng với ngôn ngữ $L = \{ab, c\}$

hay $r = r_2 = a \cdot (b + c)$ tương ứng với ngôn ngữ $L = \{ab, ac\}$?

Qui định về độ ưu tiên

Độ ưu tiên của các phép toán theo thứ tự từ cao đến thấp là phép ***bao đóng – sao****, phép* ***kết nối****, phép* ***hội****. Ngoài ra, ký hiệu đối với phép kết nối có thể bỏ qua, có thể viết* r_1r_2 *thay cho* $r_1{\cdot}r_2$.

Xác định ngôn ngữ cho BTCQ

Ví dụ 3.4. Tìm ngôn ngữ của BTCQ sau:

$$r = (aa)^*(bb)^*b$$

Kết quả: $L(r) = \{a^{2n}b^{2m+1} : n \geq 0, m \geq 0\}$

Tìm BTCQ cho ngôn ngữ

Ví dụ 3.5. Tìm BTCQ cho ngôn ngữ sau trên $\{0, 1\}$:

$$L(r) = \{w: w \text{ có ít nhất một cặp số 0 liên tiếp}\}$$

Kết quả: $r = (0 + 1)^*\, 00\, (0 + 1)^*$

Ví dụ 3.6. Tìm BTCQ cho ngôn ngữ sau trên $\{0, 1\}$:

$$L(r) = \{w: w \text{ không có một cặp số 0 liên tiếp nào}\}$$

Kết quả: $r_1 = (1^*\, 011^*)^*\, (0 + \lambda) + 1^*(0 + \lambda)$

$= (1^*\, 011^* + 1)^*\, (0 + \lambda)$

hay $r_2 = (1 + 01)^*\, (0 + \lambda)$

Nhận xét: việc tìm một BTCQ cho một ngôn ngữ khó hơn việc xác định ngôn ngữ của BTCQ vì không có giải thuật cho loại bài toán này.

Ghi chú: trong một số tài liệu phép cộng (hoặc) trong biểu thức chính qui được kí hiệu bằng dấu | thay cho dấu + . Chẳng hạn $(a + b).c$ thì được viết là $(a \mid b).c$. Ngoài ra còn giới thiệu thêm một số phép toán khác nữa rất hữu hiệu. Tuy nhiên, các phép toán này đều có thể biểu diễn thông qua ba phép toán mà chúng ta đã học. Chẳng hạn:

***Phép chọn lựa* ?:** nghĩa là có hoặc không có, cụ thể:

$$r\,? = (r + \lambda)$$

Đôi khi ***r?*** còn được kí hiệu là ***[r]***

***Phép bao đóng dương* $^+$: $r^+ = r.r^*$**

Chú ý:
$$(r^*)^* = r^*$$
$$(r_1^* + r_2)^* = (r_1 + r_2)^*$$
$$(r_1 r_2^* + r_2)^* = (r_1 + r_2)^*$$

3.2. MỐI QUAN HỆ GIỮA BTCQ VÀ NGÔN NGỮ CHÍNH QUI

BTCQ và NNCQ có một mối quan hệ chặt chẽ với nhau. Chúng ta sẽ chứng minh rằng hai lớp này là tương đương nhau. Tức là với mỗi NNCQ thì có một BTCQ, và ngược lại với mỗi BTCQ thì có một NNCQ tương ứng. Đầu tiên chúng ta sẽ chứng minh nếu r là một BTCQ, thì $L(r)$ là một NNCQ.

1. Biểu thức chính qui biểu thị ngôn ngữ chính qui

Định lý 3.1. *Cho r là một BTCQ, thì tồn tại một nfa nào đó mà chấp nhận L(r). Vì vậy, L(r) là NNCQ.*

Chứng minh

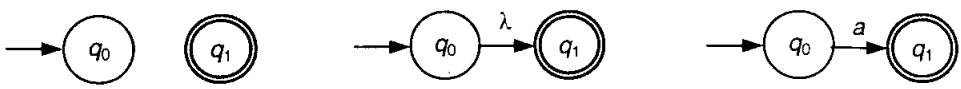
Ta chứng minh bằng cách xây dựng một nfa cho BTCQ.

Thủ tục*: *re-to-nfa

Input : biểu thức chính qui r.

Output : nfa $M = (Q, \Sigma, \delta, q_0, F)$.

Đầu tiên chúng ta xây dựng các nfa cho các BTCQ nguyên thủy. Xem H.3.1



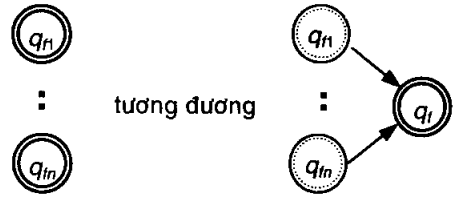
(a) nfa chấp nhận ∅ (b) nfa chấp nhận {λ} (c) nfa chấp nhận {a}

Hình 3.1

Để xây dựng nfa cho các BTCQ phức tạp hơn trước hết ta chứng minh bổ đề sau:

Bổ đề 3.1. *Với mọi nfa luôn luôn có một nfa tương đương với chỉ một trạng thái kết thúc.*

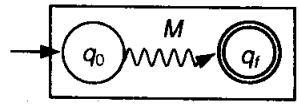
Nếu nfa không có trạng thái kết thúc nào thì ta sẽ thêm vào một trạng thái kết thúc không đạt tới được. Còn nếu nfa có nhiều hơn một trạng thái kết thúc thì chúng ta thêm vào một trạng thái kết thúc mới, cho những trạng thái kết thúc cũ không còn là kết thúc nữa và cho chúng chuyển trạng thái–λ vào trạng thái kết thúc mới.



Hình 3.2

Chú ý trạng thái có một nét liền và một nét đứt biểu thị ý rằng trước đây nó là trạng thái kết thúc nhưng sau đó không còn là kết thúc nữa.

Từ bổ đề trên mọi nfa có thể được biểu diễn bằng sơ đồ như sau:

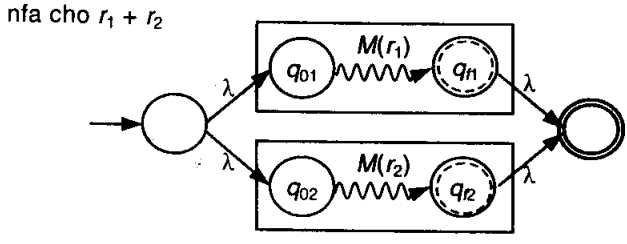


Hình 3.3

Dễ thấy các nfa trong H.3.1 thỏa mãn bổ đề.

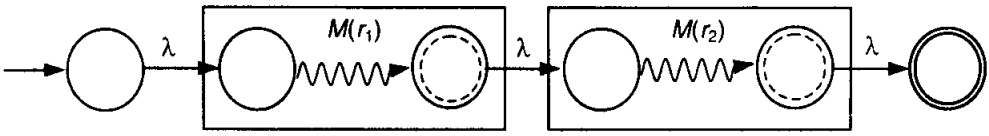
Vì các BTCQ phức tạp được xây dựng nên từ các BTCQ nguyên thủy thông qua các phép toán +, . và * nên chúng ta cũng sẽ xây dựng nfa cho các BTCQ phức tạp từ các nfa nguyên thủy trong H.3.1 ở trên thông qua các phép toán nêu trên.

Giả sử ta đã có hai nfa cho hai BTCQ r_1 và r_2 và thỏa mãn bổ đề trên. Ta sẽ xây dựng nfa cho các BTCQ $r_1 + r_2$, r_1r_2 và r_1^* lần lượt như sau:

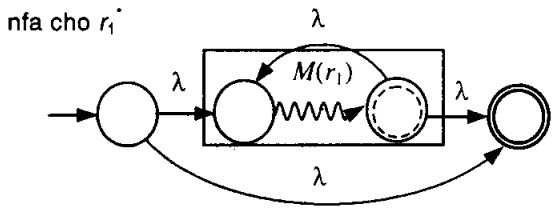


Hình 3.4

Hai cách xây dựng nfa cho $r_1 r_2$



Hình 3.5



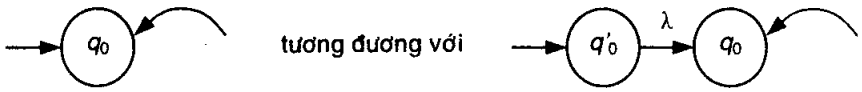
Hình 3.6

Nhận xét: các nfa được xây dựng theo cách trên đây đều thỏa mãn bổ đề 3.1.

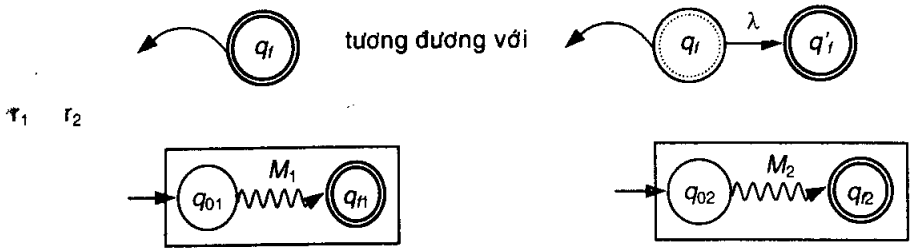
Các phương án cải tiến để giảm bớt số trạng thái

Bổ đề 3.2. *Với mọi nfa luôn luôn có một nfa tương đương ngoài việc thỏa mãn bổ đề 3.1 nó còn có tính chất không có cạnh nào đi vào trạng thái khởi đầu và không có cạnh nào đi ra khỏi trạng thái kết thúc.*

Nếu nfa có cạnh đi vào trạng thái khởi đầu thì thêm vào một trạng thái khởi đầu mới, trạng thái khởi đầu cũ không còn là trạng thái khởi đầu nữa và cho chuyển trạng thái–λ từ trạng thái khởi đầu mới vào trạng thái khởi đầu cũ.



Nếu nfa có cạnh đi ra trạng thái kết thúc thì thêm vào một trạng thái kết thúc mới, trạng thái kết thúc cũ sẽ không còn là kết thúc nữa và cho chuyển trạng thái–λ từ trạng thái kết thúc cũ vào trạng thái kết thúc mới.

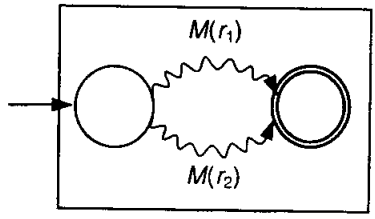


Hình 3.7

1- Cải tiến trên $r_1 + r_2$

B1. Nếu hai automat chưa thỏa bổ đề 3.2 thì áp dụng bổ đề này cho chúng. Dĩ nhiên nếu đã thỏa thì thôi và chính trong trường hợp này như chúng ta sẽ thấy chúng ta sẽ tiết kiệm được trạng thái.

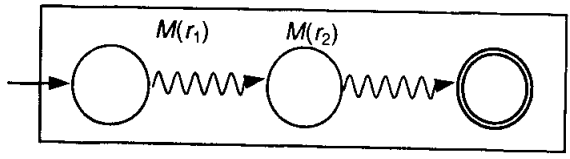
B2. Ghép hai trạng thái khởi đầu thành một trạng thái khởi đầu chung, ghép hai trạng thái kết thúc thành một trạng thái kết thúc chung.



Hình 3.8

2- Cải tiến trên $r_1 . r_2$

Nếu không đồng thời xảy ra: có cạnh đi ra khỏi trạng thái kết thúc q_{f1} của M_1 và có cạnh đi vào trạng thái khởi đầu q_{02} của M_2 thì ghép chung q_{f1} và q_{02} thành một trạng thái, như hình bên dưới

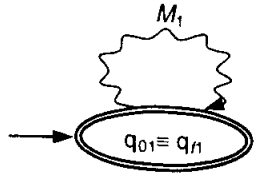


Hình 3.9

3- Cải tiến trên r_1^*

B1. Áp dụng bổ đề 3.2 cho M_1 nếu nó chưa thỏa bổ đề này.

B2. Ghép trạng thái khởi đầu và kết thúc lại thành một trạng thái.

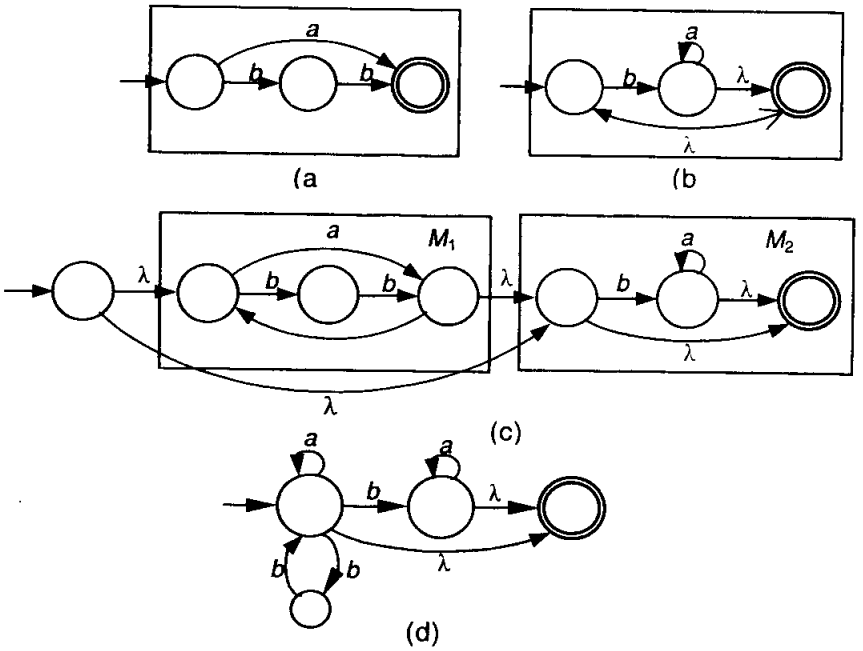


Hình 3.10

Ví dụ 3.7. Tìm nfa chấp nhận ngôn ngữ của BTCQ sau:

$$r = (a + bb)^*(ba^* + \lambda)$$

Hình 3.11(a) là nfa cho BTCQ $(a + bb)$, hình 3.11(b) là nfa cho BTCQ $(ba^* + \lambda)$, còn H.3.11(c) là nfa cho BTCQ bài ra. Hình 3.11(d) là nfa cải tiến.



Hình 3.11

2. Biểu thức chính qui cho ngôn ngữ chính qui

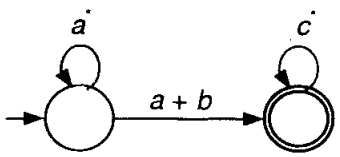
Ta sẽ chứng minh mệnh đề nghịch đảo của định lý 3.1 cũng đúng, bằng cách tìm một BTCQ có khả năng sinh ra tất cả các nhãn của các con đường đi từ q_0 đến một trạng thái kết thúc nào đó của nfa. Để làm điều này ta sẽ sử dụng một khái niệm mới là **đồ thị chuyển trạng thái tổng quát (ĐTCTTTQ)**.

Đồ thị chuyển trạng thái tổng quát *(generallized transition graphs)*

Là một ĐTCTT ngoại trừ các cạnh của nó được gán nhãn bằng các BTCQ.

Ngôn ngữ được chấp nhận bởi nó là tập tất cả các chuỗi được sinh ra bởi các BTCQ mà là nhãn của một con đường nào đó đi từ trạng thái khởi đầu đến một trạng thái kết thúc nào đó của ĐTCTT tổng quát (ĐTCTTTQ).

Ví dụ 3.8. Hình 3.12 biểu diễn một ĐTCTTTQ. Ngôn ngữ được chấp nhận bởi nó là $L(a^* + a^*(a + b)c^*)$



Hình 3.12

Nhận xét

1. ĐTCTT của một nfa bất kỳ có thể được xem là *ĐTTCTTTQ* nếu các nhãn cạnh được diễn dịch như sau:

- Một cạnh được gán nhãn là một kí hiệu đơn a được diễn dịch thành cạnh được gán nhãn là biểu thức a.
- Một cạnh được gán nhãn với nhiều kí hiệu a, b, . . . thì được diễn dịch thành cạnh được gán nhãn là biểu thức $a + b + \ldots$

2. Từ đây, suy ra rằng: *mọi NNCQ đều tồn tại một ĐTCTTTQ chấp nhận nó. Ngược lại, mỗi ngôn ngữ mà được chấp nhận bởi một ĐTCTTTQ là chính qui.*

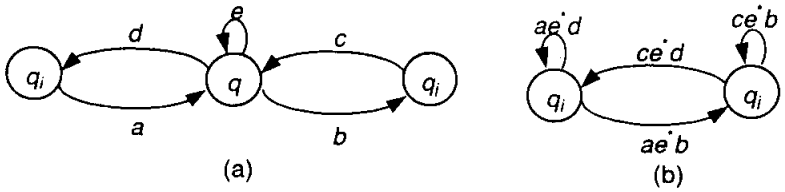
Chú ý: sự tương đương giữa các ĐTCTTTQ cũng được định nghĩa trong thuật ngữ của ngôn ngữ được chấp nhận.

Rút gọn trạng thái của một đồ thị chuyển trạng thái tổng quát

Để tìm BTCQ cho một ĐTCTTTQ ta sẽ thực hiện quá trình rút gọn các trạng thái trung gian của nó thành một ĐTCTTTQ tương đương đơn giản nhất có thể được.

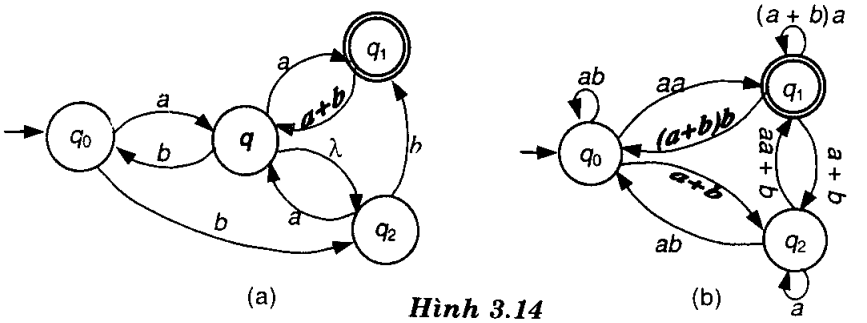
Trạng thái trung gian: *là trạng thái mà không phải là trạng thái khởi đầu, cũng không phải là trạng thái kết thúc.*

Bổ đề 3.3. *ĐTCTTTQ được trình bày trong* H.3.13(b) *là tương đương với ĐTCTTTQ được trình bày trong* H.3.13(a) *sau khi đã rút gọn trạng thái trung gian q.*



Hình 3.13

Ví dụ 3.9. Rút gọn trạng thái q của ĐTCTTTQ cho trong hình sau.



Hình 3.14

Kết quả được cho trong H.3.14(b)

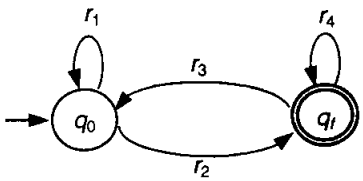
Định lý 3.2. *Cho L là một NNCQ, thì tồn tại một BTCQ r sao cho*

$$L = L(r)$$

Chứng minh

Cho M là một nfa chấp nhận L. Theo bổ đề 3.1 chúng ta có thể giả sử M có một trạng thái khởi đầu và một trạng thái kết thúc.

Biến đổi ĐTCTT của M thành ĐTCTTTQ tương ứng. Sử dụng bổ đề 3.3 rút gọn tất cả các trạng thái trung gian của nó kết quả ta được ĐTCTTTQ tương đương đơn giản nhất có dạng như H.3.15

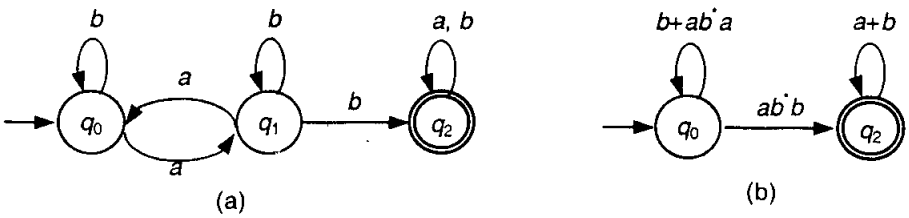


Hình 3.15

BTCQ tương ứng với ĐTCTTTQ này là $r = r_1^* r_2 (r_4 + r_3 r_1^* r_2)^*$

Đây cũng chính là BTCQ tương ứng với L.

Ví dụ 3.10. Xét nfa trong H.3.16(a). ĐTCTTTQ tương ứng sau khi loại bỏ trạng thái q_1 được trình bày trong H.3.16(b). Từ đây suy ra BTCQ tương ứng với nfa này là $r = (b + ab^*a)^* ab^*b(a + b)^*$



Hình 3.16

3. Biểu thức chính qui trong việc mô tả các mẫu (*partern*) đơn giản

Trong các ngôn ngữ lập trình

BTCQ thường được dùng để đặc tả từ vựng cho các ngôn ngữ lập trình **chẳng** hạn đặc tả các danh hiệu, các số nguyên, số thực ...*Ví dụ*:

BTCQ để mô tả các danh hiệu của Pascal là

$$(letter)(letter + digit)^*$$

trong đó: "*letter*" biểu diễn cho A + B + . . . + Z + a + b + . . . + z, còn "*digit*" biểu diễn cho 0 + 1 + . . . + 9.

BTCQ để mô tả tập tất cả các số nguyên của Pascal là

$$sdd^*$$

trong đó: s biểu diễn cho dấu, với các giá trị có thể là {+, –, λ}, còn d biểu diễn cho các kí số từ 0 đến 9 như "digit" ở trên.

Trong các trình soạn thảo văn bản hay các ứng dụng có xử lý chuỗi

Chẳng hạn trình *ed* trong UNIX có lệnh

$$/aba^*c/$$

để tìm sự xuất hiện đầu tiên của chuỗi có dạng trên. Hay lệnh

$$s/bbb^*/b/$$

để thay thế các chuỗi có dạng bbb^* thành chuỗi có một kí tự *b*.

Hay trong các lệnh của MS –DOS và NC, Ví dụ lệnh

$$\text{del } tmp^*.???$$

xóa tất cả các file đi đầu bằng *tmp*, sau đó là một chuỗi bất kỳ và có đuôi là ba kí tự tùy ý. Dấu * trong trường hợp này biểu thị cho một chuỗi bất kỳ, còn dấu ? biểu thị cho một kí tự. Đây cũng là một dạng của biểu thức chính qui có điều qui ước sử dụng kí hiệu khác biệt so với chúng ta.

Hay trong việc tìm kiếm theo mẫu trong các trang web.

Ghi chú

Để làm được điều này các ứng dụng phải thực hiện quá trình chuyển đổi một biểu thức chính qui thành nfa. Và vì cơ chế hoạt động của nfa phức tạp nên thông thường nfa phải được biến đổi tiếp thành dfa tương đương, sau đó rồi mới dùng dfa để thực hiện việc kiểm tra mẫu. Đôi khi người ta cũng có thể rút gọn luôn dfa này thành dfa tối giản để hoạt động được đơn giản hơn.

Theo những gì ta đã biết thì để biến đổi một biểu thức chính qui thành một dfa tương đương phải thông qua nfa. Tuy nhiên có một giải thuật biến đổi trực tiếp một biểu thức chính qui thành một dfa tương đương mà không thông qua nfa. Giải thuật này được trình bày trong phần phụ lục của sách này.

3.3. VĂN PHẠM CHÍNH QUI (*REGULAR GRAMMAR*)

Một cách thứ ba để mô tả NNCQ là bằng một loại văn phạm đơn giản nào đó, người ta gọi nó là văn phạm chính qui (VPCQ).

1. Văn phạm tuyến tính – phải và – trái

Định nghĩa 3.3. *Một văn phạm G = (V, T, S, P) được gọi là* ***tuyến tính – phải*** *(**TT–P**) (right–linear) nếu tất cả luật sinh là có dạng*

$$A \to xB$$
$$A \to x$$

trong đó: $A, B \in V, x \in T^*$.

Một văn phạm được gọi là ***tuyến tính – trái*** **(*TT–T*)** *(left–linear) nếu tất cả các luật sinh là có dạng*

$$A \to Bx,$$

hay $$A \to x.$$

Một ***văn phạm chính qui*** *(VPCQ) là văn phạm mà hoặc là tuyến tính–phải hoặc là tuyến tính–trái.*

Ví dụ 3.11. Văn phạm $G_1 = (\{S\}, \{a, b\}, S, P_1)$, với P_1 được cho là

$$S \to abS|a$$

là TT–*P*.

Văn phạm $G_2 = (\{S, S_1, S_2\}, \{a, b\}, S, P_2)$, với các luật sinh

$$S \to S_1ab,$$
$$S_1 \to S_1ab|S_2,$$
$$S_2 \to a,$$

là TT–*T*. Cả hai văn phạm G_1 và G_2 đều là VPCQ.

Dãy: $S => abS => ababS => ababa$

là một dẫn xuất trong G_1.

Dễ thấy

$$L(G_1) = L((ab)^*a)$$
$$L(G_2) = L(a(ab)^*)$$

Ví dụ 3.12. Văn phạm $G = (\{S, A, B\}, \{a, b\}, S, P)$, với các luật sinh

$$S \to A,$$
$$A \to aB|\lambda,$$
$$B \to Ab,$$

không phải là một VPCQ. Đây là một ví dụ của văn phạm tuyến tính (VPTT).

Văn phạm tuyến tính *(Linear Grammar)*

Một văn phạm được gọi là tuyến tính nếu mọi luật sinh của nó có tối đa một biến xuất hiện ở vế phải của luật sinh và không có sự giới hạn nào trên vị trí xuất hiện của biến này.

Mục đích kế tiếp của chúng ta là chứng minh lớp VPCQ tương đương với lớp NNCQ. Đầu tiên ta chứng minh lớp VPTT–*P* là tương đương với lớp NNCQ thông qua hai phần sau đây.

2. Văn phạm tuyến tính – phải sinh ra ngôn ngữ chính qui

Định lý 3.3. *Cho $G = (V, T, S, P)$ là một VPTT–P thì $L(G)$ là NNCQ.*

Chứng minh

Để chứng minh ta chủ yếu chỉ ra cách xây dựng nfa từ VPTT–P đã cho.

Thủ tục: G_P to nfa

Input : văn phạm tuyến tính–phải $G_P = (V, T, S, P)$

Output: nfa $M = (Q, \Sigma, \delta, q_0, F)$

- Ứng với mỗi biến V_i của văn phạm ta xây dựng một trạng thái mang nhãn V_i cho nfa tức là $Q \supset V$.
- Ứng với biến khởi đầu V_0 , trạng thái V_0 của nfa sẽ trở thành trạng thái khởi đầu, tức là $S = V_0$
- Nếu trong văn phạm có một luật sinh nào đó dạng

$$V_i \rightarrow a_1a_2...a_m$$

thì thêm vào nfa một và chỉ một trạng thái kết thúc V_f.

- Ứng với mỗi luật sinh của văn phạm có dạng

$$V_i \rightarrow a_1a_2...a_mV_j$$

thêm vào nfa các chuyển trạng thái sao cho (xem H.3.17):

$$\delta^*(V_i, a_1a_2...a_m) = V_j$$

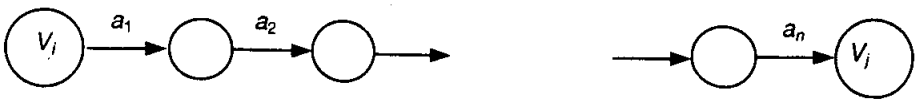
- Ứng với mỗi luật sinh dạng

$$V_i \rightarrow a_1a_2...a_m$$

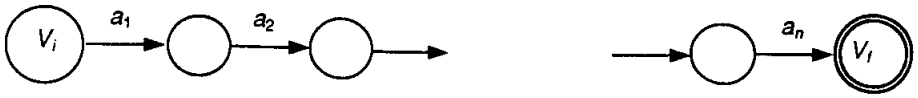
thêm vào nfa các chuyển trạng thái sao cho (xem H.3.17):

$$\delta^*(V_i, a_1a_2...a_m) = V_f$$

Nfa được xây dựng theo cách này sẽ chấp nhận ngôn ngữ được sinh ra bởi văn phạm. Sinh viên tự chứng minh bằng qui nạp.



Biểu diễn $V_i \rightarrow a_1a_2 \cdots a_mV_j$



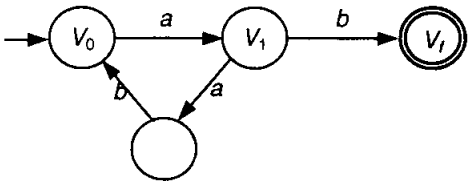
Biểu diễn $V_i \rightarrow a_1a_2 \cdots a_m$

Hình 3.17

Ví dụ 3.13. Xây dựng một automat chấp nhận ngôn ngữ được sinh ra bởi văn phạm sau:

$$V_0 \rightarrow aV_1$$
$$V_1 \rightarrow abV_0|b$$

Nfa kết quả được cho trong H.3.18



Hình 3.18

3. Văn phạm tuyến tính – phải cho ngôn ngữ chính qui

Định lý 3.4. *Nếu L là một NNCQ trên bảng chữ cái* Σ, *thì tồn tại một VPTT–P* $G = (V, \Sigma, S, P)$ *sao cho* $L = L(G)$.

Chứng minh

Ta chủ yếu chỉ ra cách xây dựng VPTT–*P* từ một dfa của NNCQ.

Thủ tục: nfa to G_P

Input : nfa $M = (Q, \Sigma, \delta, q_0, F)$

Output : văn phạm tuyến tính–phải $G_P = (V, T, S, P)$

Giả thiết $Q = \{q_0, q_1, ..., q_n\}$ và $\Sigma = \{a_1, a_2, ..., a_m\}$. Ta xây dựng VPTT–*P* *G* như sau:

- $V = Q$, tức là mỗi trạng thái trong nfa trở thành một biến trong văn phạm, $S = q_0$.
- Đối với mỗi chuyển trạng thái $\delta(q_i, a_j) = q_k$ của *M* ta xây dựng luật sinh TT–*P* tương ứng $q_i \rightarrow a_j q_k$.
- Đối với mỗi trạng thái $q_f \in F$ chúng ta xây dựng luật sinh $q_f \rightarrow \lambda$.

VPTT–*P* được xây dựng theo kiểu này sẽ sinh ra ngôn ngữ *L*. Sinh viên tự chứng minh $L(G) = L(M)$ bằng qui nạp.

Ví dụ 3.14. Xây dựng VPTT–*P* cho ngôn ngữ $L(aab^*a)$.

Dfa và VPTT–*P* cho ngôn ngữ này được cho trong bảng sau:

$\delta(q_0, a) = q_1$	$q_0 \rightarrow aq_1$
$\delta(q_1, a) = q_2$	$q_1 \rightarrow aq_2$
$\delta(q_2, b) = q_2$	$q_2 \rightarrow bq_2$
$\delta(q_2, a) = q_f$	$q_2 \rightarrow aq_f$
$q_f \in F$	$q_f \rightarrow \lambda$

4. Sự tương đương giữa văn phạm chính qui và ngôn ngữ chính qui

Hai định lý trước đã thiết lập mối quan hệ tương đương giữa NNCQ và VPTT–*P*. Người ta có thể thiết lập một mối quan hệ tương tự giữa NNCQ và VPTT–*T*, và vì vậy sẽ chứng minh được sự tương đương hoàn toàn giữa VPCQ và NNCQ.

Định lý 3.5. *Ngôn ngữ L là chính qui nếu và chỉ nếu tồn tại một VPTT–T G sao cho L = L(G).*

Chứng minh

Ta chứng minh mối quan hệ tương đương thông qua VPTT–*P*.

Từ VPTT–*T* G_T đã cho ta xây dựng VPTT–*P* G_P tương ứng như sau:

- Ứng với luật sinh TT–*T* $A \rightarrow Bv$ ta xây dựng luật sinh TT–*P*

$$A \rightarrow v^R B$$

- Ứng với luật sinh TT–*T* $A \rightarrow v$ ta xây dựng luật sinh TT–*P*

$$A \rightarrow v^R$$

- G_P được xây dựng theo cách trên có quan hệ với G_T như sau:

$$L(G_T) = L(G_P)^R$$

Để chứng minh bổ đề này ta chứng minh bằng qui nạp rằng: mọi dạng câu được sinh ra bởi G_P tương ứng có dạng câu nghịch đảo trong G_T, với một chú ý rằng: nếu α, β là các chuỗi thì $(\alpha\beta)^R = \beta^R\alpha^R$. Sinh viên tự thực hiện chứng minh này.

Bổ đề 3.3. *Nếu L là chính qui thì L^R cũng chính qui.*

Chứng minh

Gọi $M = (Q, \Sigma, \delta, q_0, F)$ là nfa cho *L*. Không mất tính tổng quát, giả sử *M* chỉ có một trạng thái kết thúc, $F = \{q_f\}$.

Ta xây dựng nfa $M_R = (Q_R, \Sigma, \delta_R, q_{0R}, F_R)$ từ *M* bằng thủ tục sau:

Thủ tục: nfa to nfa$_R$

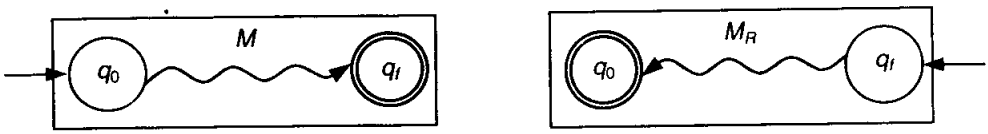
Input : nfa $M = (Q, \Sigma, \delta, q_0, F)$

Output : nfa $M_R = (Q_R, \Sigma, \delta_R, q_{0R}, F_R)$

$Q_R = Q$, $q_{0R} = q_f$, $F_R = q_0$,

còn $\delta_R(q_j, a) = q_i$ nếu và chỉ nếu có: $\delta(q_i, a) = q_j$. Cách xây dựng trên có nghĩa là đảo trạng thái kết thúc q_f của *M* thành trạng thái khởi đầu của M_R, còn trạng thái khởi đầu q_0 của *M* thành trạng thái kết thúc của M_R. Các cạnh chuyển trạng thái thì đổi theo chiều ngược lại.

Xem hình:



Dễ chứng minh rằng trong M có con đường đi từ q_0 đến q_f mang nhãn w khi và chỉ khi trong M_R có con đường ngược lại đi từ q_f đến q_0 mang nhãn w^R. Vậy $L(M_R) = L(M)^R$. Suy ra L^R chính qui.

Kết hợp ba định lý trên (3.3, 3.4 và 3.5) ta suy ra định lý sau:

Định lý 3.6. *Một ngôn ngữ L là chính qui nếu và chỉ nếu tồn tại một VPCQ G sao cho L = L(G).*

Nhận xét

Từ chứng minh của định lý 3.5 ta có thể đưa ra một giải thuật để xây dựng VPTT–T cho một nfa hay ngược lại tìm một nfa cho một VPTT–T. Phần này để lại như một bài tập cho người đọc.

CÁC TÍNH CHẤT CỦA NGÔN NGỮ CHÍNH QUI

Chúng ta đã định nghĩa NNCQ, nghiên cứu một vài cách biểu diễn nó, và đã thấy một vài ví dụ về sự hữu hiệu của nó. Đến đây có một số câu hỏi được đặt ra:

- NNCQ tổng quát là như thế nào? Có phải chăng mọi ngôn ngữ hình thức đều là chính qui?

Để trả lời được câu hỏi này chúng ta phải hiểu biết sâu hơn về các tính chất của NNCQ. Chẳng hạn:

- Khi chúng ta thực hiện các phép toán trên NNCQ thì kết quả sẽ như thế nào, có còn là một NNCQ hay không?

Các phép toán mà chúng ta đã xét là các phép toán đơn giản, chẳng hạn phép kết nối, phép hội, phép bao đóng sao .... Chúng ta tham khảo đến những phép toán này như là những câu hỏi về ***tính đóng***. Tính đóng, mặc dù hầu hết thuộc những quan tâm về lý thuyết, nhưng nó giúp đỡ chúng ta trong việc phân biệt giữa các họ ngôn ngữ khác nhau mà chúng ta sẽ gặp.

Một dạng câu hỏi thứ hai là

- Một ngôn ngữ nào đó có hữu hạn hay không? Có rỗng hay không?

Như chúng ta sẽ thấy, những câu hỏi như vậy dễ dàng trả lời đối với NNCQ, nhưng không dễ dàng trả lời như vậy đối với các họ ngôn ngữ khác.

Cuối cùng là một câu hỏi quan trọng:

- Làm thế nào để biết một ngôn ngữ đã cho có là chính qui hay không?

Để trả lời được điều này chúng ta phải nghiên cứu các tính chất tổng quát của NNCQ. Nếu chúng ta biết được một tính chất nào đó kiểu như thế, và chúng ta có thể chứng tỏ ngôn ngữ đang xét không có tính chất đó, thì chúng ta có thể nói rằng ngôn ngữ đó là không chính qui.

Trong chương này, chúng ta sẽ tìm hiểu các tính chất khác nhau của NNCQ. Những tính chất này cho chúng ta biết những gì mà các NNCQ có thể làm và không thể làm được. Sau này khi chúng ta xem xét các câu hỏi như vậy đối với các họ ngôn ngữ khác, những điểm giống nhau và khác nhau trong những tính chất này sẽ cho phép chúng ta thấy được sự tương phản giữa các họ ngôn ngữ này.

4.1. TÍNH ĐÓNG CỦA NGÔN NGỮ CHÍNH QUI

1. Đóng dưới các phép toán tập hợp đơn giản

Định lý 4.1. *Nếu L_1 và L_2 là các NNCQ, thì $L_1 \cup L_2$, $L_1 \cap L_2$, L_1L_2, $\overline{L_1}$ và L_1^* cũng vậy. Chúng ta nói rằng họ NNCQ là đóng dưới các phép hội, giao, kết nối, bù và bao đóng-sao.*

Chứng minh

Nếu L_1, L_2 là chính qui thì tồn tại các BTCQ r_1, r_2 sao cho $L_1 = L(r_1)$, $L_2 = L(r_2)$. Theo định nghĩa $r_1 + r_2$, r_1r_2 và r_1^* là các BTCQ định nghĩa các ngôn ngữ $L_1 \cup L_2$, L_1L_2, và L_1^*. Vì vậy họ NNCQ là đóng đối với các phép toán này.

Để chứng minh tính đóng đối với p hép bù, cho $M = (Q, \Sigma, \delta, q_0, F)$ là dfa chấp nhận L_1, thì dfa

$$\hat{M} = (Q, \Sigma, \delta, q_0, Q - F)$$

chấp nhận $\overline{L_1}$. Điều này được chứng minh khá dễ dàng bằng định nghĩa của ngôn ngữ bù và định nghĩa ngôn ngữ được chấp nhận bởi một dfa.

Để chứng minh tính đóng đối với phép giao ta có hai cách như sau:

- ***Cách thứ nhất*** là dựa vào qui tắc De Morgan ta có:

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

Dựa vào tính đóng của phép bù và phép hội vừa được chứng minh ở trên ta suy ra tính đóng đối với phép giao.

- ***Cách thứ hai*** là ta sẽ xây dựng một dfa cho $L_1 \cap L_2$.

Cho $M_1 = (Q, \Sigma, \delta_1, q_0, F_1)$ và $M_2 = (P, \Sigma, \delta_2, p_0, F_2)$ là các dfa lần lượt chấp nhận L_1, L_2.

Ta xây dựng dfa $\hat{M} = (\hat{Q}, \Sigma, \hat{\delta}, (q_0, p_0), \hat{F})$ chấp nhận $L_1 \cap L_2$ bằng thủ tục sau:

Thủ tục: intersection

Input : dfa $M_1 = (Q, \Sigma, \delta_1, q_0, F_1)$ và dfa $M_2 = (P, \Sigma, \delta_2, p_0, F_2)$

Output: dfa $\hat{M} = (\hat{Q}, \Sigma, \hat{\delta}, (q_0, p_0), \hat{F})$

- $\hat{Q} = Q \times P$, tức là $\hat{Q} = \{(q_i, p_j)$: trong đó $q_i \in Q$ còn $p_j \in P\}$.
- Các chuyển trạng thái được xây dựng như sau:

$$\hat{\delta}((q_i, p_j), a) = (q_k, p_l)$$

nếu và chỉ nếu: $\delta_1(q_i, a) = q_k$ và $\delta_2(p_j, a) = p_l$

- $\hat{F} = \{(q_i, p_j)$: trong đó $q_i \in F_1$ còn $p_j \in F_2\}$

Dựa vào cách xây dựng $\hat{\delta}$ ta thấy $\hat{M}$ được xây dựng như trên mô phỏng lại quá trình xử lý đồng thời của hai dfa M_1 và M_2. Ngoài ra dựa vào định nghĩa của $\hat{F}$ ta thấy $\hat{M}$ chỉ chấp nhận những chuỗi đồng thời được cả hai dfa M_1 và M_2 chấp nhận. Vì vậy $\hat{M}$ chấp nhận $L_1 \cap L_2$.

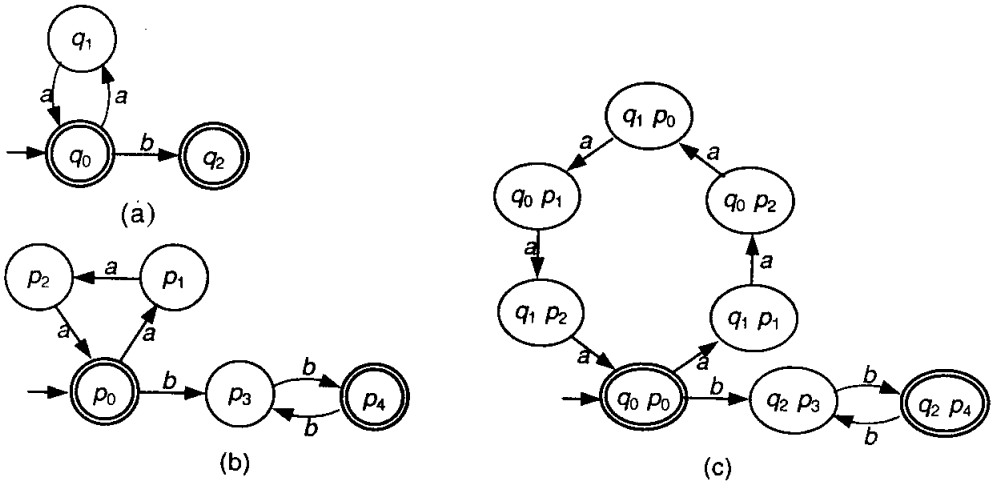
Ví dụ 4.1. Cho

$$L_1 = \{a^{2n}b^m: n, m \geq 0\}$$

và

$$L_2 = \{a^{3n}b^{2m}: n, m \geq 0\}$$

có các dfa lần lượt được cho trong H.4.1(a) và (b). Dfa giao của hai dfa này theo cách xây dựng trên được cho trong H.4.1(c).



Hình 4.1

Định lý 4.2. *Họ NNCQ là đóng dưới phép hiệu và nghịch đảo.*

Để chứng minh tính đóng đối với phép hiệu dựa vào các qui tắc tập hợp ta có:

$$L_1 - L_2 = L_1 \cap \overline{L_2}$$

Dựa vào tính đóng của phép bù và phép giao đã được chứng minh, suy ra tính đóng cho phép hiệu.

Còn tính đóng đối với phép nghịch đảo được suy ra từ chứng minh trong bổ đề 3.5.

2. Đóng dưới các phép toán khác

Ngoài các phép toán chuẩn trên ngôn ngữ, chúng ta giới thiệu thêm hai phép toán điển hình khác và nghiên cứu tính đóng đối với chúng. Đó là phép đồng hình và phép thương đúng.

Định nghĩa 4.1. *Giả sử* Σ *và* Γ *là các bảng chữ cái, thì một hàm*

$$h : \Sigma \to \Gamma^*$$

được gọi là một phép ***đồng hình*** *(homomorphism). Bằng lời, một phép đồng hình là một sự thay thế trong đó mỗi kí hiệu đơn được thay thế bằng một chuỗi. Miền xác định (domain) của hàm h được mở rộng tới chuỗi trong một khuôn mẫu rõ ràng.*

Nếu $w = a_1a_2... a_n$, *thì* $h(w) = h(a_1)h(a_2)...h(a_n)$.

Nếu L là ngôn ngữ trên Σ, *thì* ***ảnh đồng hình*** *(homomorphic image) của nó được định nghĩa là*

$$h(L) = \{h(w): w \in L\}.$$

Ví dụ 4.2. Cho $\Sigma = \{a, b\}$, $\Gamma = \{ a, b, c \}$ và h được định nghĩa như sau:

$$h(a) = ab,$$
$$h(b) = bbc$$

thì

$$h(aba) = abbbcab.$$

Ảnh đồng hình của $L = \{aa, aba\}$ là ngôn ngữ

$$h(L) = \{abab, abbbcab\}.$$

Nhận xét: nếu chúng ta có BTCQ r biểu thị ngôn ngữ L thì BTCQ cho $h(L)$ có thể nhận được bằng cách áp dụng phép đồng hình tới mỗi kí hiệu của r.

Ví dụ 4.3. Cho $\Sigma = \{a, b\}$, $\Gamma = \{ b, c, d \}$ và h được định nghĩa như sau:

$$h(a) = dbcc,$$
$$h(b) = bdc.$$

Nếu L là ngôn ngữ được biểu thị bởi BTCQ

$$r = (a + b^*)(aa)^*,$$

thì

$$r_1 = (dbcc + (bdc)^*)(dbccdbcc)^*,$$

là BTCQ biểu thị cho $h(L)$. Từ đó dẫn ta tới định lý sau:

Định lý 4.3. *Cho h là một đồng hình. Nếu L là một NNCQ, thì ảnh đồng hình của nó h(L) cũng là NNCQ. Họ các NNCQ vì vậy là đóng dưới phép đồng hình bất kỳ.*

Chứng minh

Ta chứng minh dựa vào ý tưởng nói trên.

Vì L là một NNCQ suy ra tồn tại một BTCQ r nào đó biểu thị cho L. Ta tìm $h(r)$ bằng cách thay thế mỗi kí hiệu a trong r bằng $h(a)$. Dễ dàng chứng minh được rằng nếu r sinh ra được chuỗi gì nếu và chỉ nếu

$h(r)$ sinh ra được chuỗi là ảnh đồng hình của chuỗi đó, vì vậy $h(r)$ là BTCQ biểu thị cho $h(L)$. Suy ra họ NNCQ là đóng dưới phép đồng hình.

Định nghĩa 4.2. *Cho $w, v \in \Sigma^*$ thì thương đúng của w cho v được kí hiệu và định nghĩa là $w/v = u$ nếu $w = uv$ có nghĩa là nếu v là tiếp vĩ ngữ của w thì kết quả của w/v là tiếp đầu ngữ tương ứng của w.*

Cho L_1 và L_2 là các ngôn ngữ trên bảng chữ cái giống nhau, thì ***thương đúng*** *(right quotient) của L_1 với L_2 được định nghĩa là*

$$L_1/L_2 = \{w/v : w \in L_1, v \in L_2\}$$
$$= \{x : xy \in L_1 \text{ với một } y \text{ nào đó } \in L_2\} \qquad (4.1)$$

Ví dụ 4.4. Cho

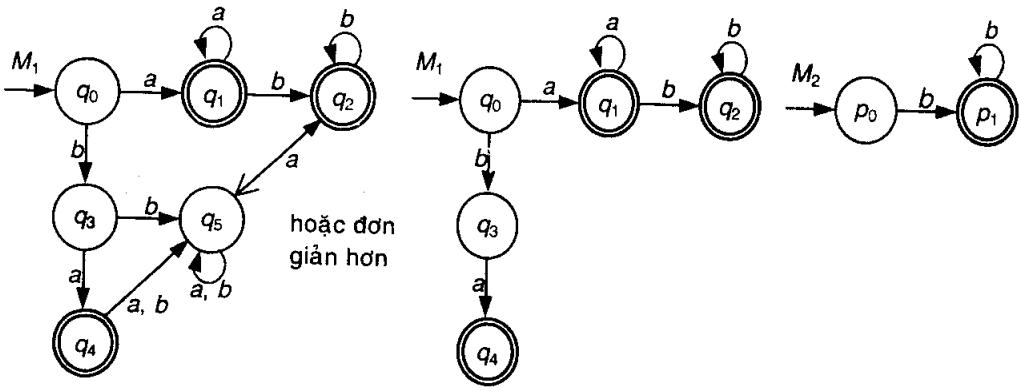
$$L_1 = \{a^n b^m : n \geq 1, m \geq 0\} \cup \{ba\} \text{ và } L_2 = \{b^m : m \geq 1\},$$

thì

$$L_1/L_2 = \{a^n b^m : n \geq 1, m \geq 0\}.$$

Vì L_1, L_2, và L_1/L_2 là các NNCQ , điều này gợi ý cho chúng ta rằng thương đúng của hai NNCQ cũng là NNCQ. Chúng ta sẽ chứng minh điều này trong một định lý sau thông qua cách xây dựng một dfa cho L_1/L_2 từ các dfa của L_1 và L_2. Bây giờ ta thử làm điều này cho ví dụ ở đây.

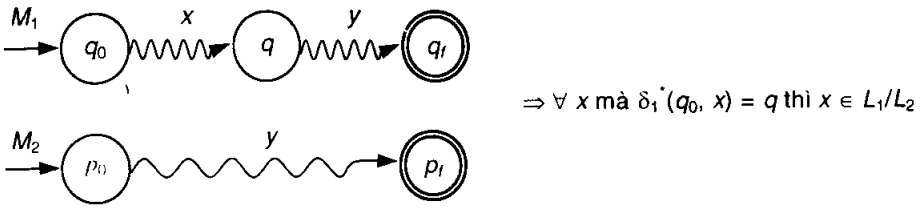
Cho $M_1 = (Q_1, \Sigma, \delta_1, q_0, F_1)$ và $M_2 = (Q_2, \Sigma, \delta_2, p_0, F_2)$ lần lượt là các dfa cho L_1 và L_2 như được vẽ trong H.4.2.



Hình 4.2

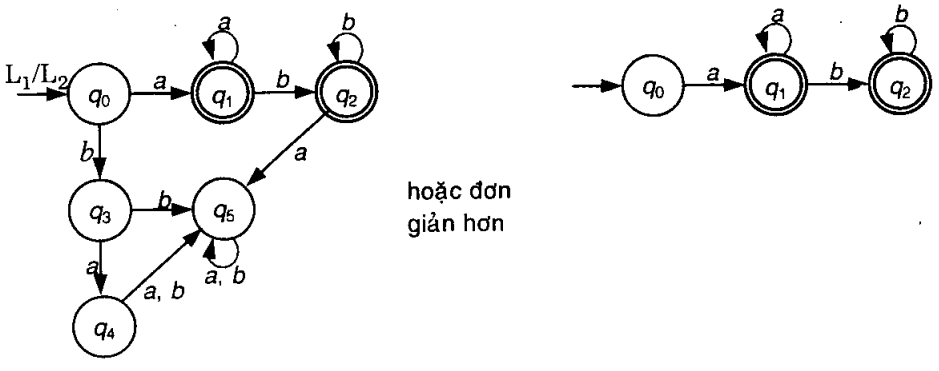
Ta thử biến đổi dfa M_1 thành dfa cho L_1/L_2. Từ định nghĩa của thương đúng ta có một nhận xét: tổng quát sau được trình bày dưới dạng bổ đề.

Bổ đề 4.1. *Cho $M_1 = (Q_1, \Sigma, \delta_1, q_0, F_1)$ là một dfa cho L_1. Nếu một trạng thái q nào đó thuộc Q có tính chất: tồn tại một chuỗi y nào đó sao cho $\delta_1^*(q, y) \in F_1$ và $y \in L_2$ thì mọi chuỗi x mà $\delta_1^*(q_0, x) = q$ sẽ $\in L_1/L_2$. Và vì vậy nếu thay những trạng thái kết thúc của M bằng những trạng thái q có tính chất này thì ta sẽ có một dfa mà chấp nhận L_1/L_2.*



Hình 4.3

Áp dụng vào ví dụ này ta thấy các trạng thái q_1 và q_2 của dfa M_1 ở trên có tính chất này nên suy ra dfa cho L_1/L_2 là



Hình 4.4

Ý tưởng này được tổng quát hóa trong định lý sau:

Định lý 4.4. *Nếu L_1 và L_2 là các NNCQ, thì L_1/L_2 cũng chính qui. Chúng ta nói rằng họ NNCQ đóng dưới phép thương đúng với một NNCQ.*

Chứng minh

Cho $L_1 = L(M)$ trong đó $M = (Q, \Sigma, \delta, q_0, F)$ là một dfa. Ta xây dựng một dfa khác $\hat{M} = (Q, \Sigma, \delta, q_0, \hat{F})$ thông qua thủ tục sau.

Thủ tục: right quotient

Input : dfa $M_1 = (Q, \Sigma, \delta_1, q_0, F_1)$ và dfa $M_2 = (P, \Sigma, \delta_2, p_0, F_2)$

Output: dfa $\hat{M} = (\hat{Q}, \Sigma, \hat{\delta}, (q_0, p_0), \hat{F})$

Đối với mỗi $q_i \in Q$, xác định có tồn tại hay không một chuỗi $y \in L_2$ sao cho $\delta^*(q_i, y) = q_f \in F$ đồng thời $y \in L_2$.

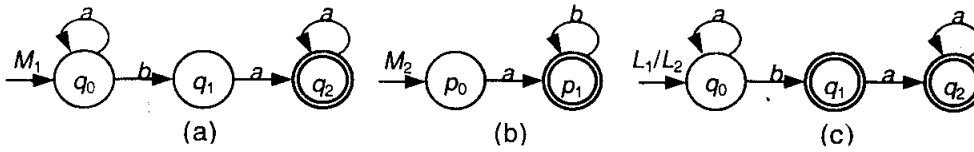
Điều này có thể thực hiện được bằng cách xét các dfa $M_i = (Q, \Sigma, \delta, q_i, F)$. Chính là M nhưng trạng thái khởi đầu q_0 được thay bằng q_i. Rồi xét xem $L_2 \cap L(M_i)$ có $\neq \varnothing$ hay không. Nếu đúng thì thêm q_i vào $\hat{F}$. Lặp lại điều này cho mọi $q_i \in Q$, ta xác định được $\hat{F}$ và vì vậy xây dựng được $\hat{M}$.

Với $\hat{M}$ được xây dựng theo kiểu này, dựa vào bổ đề 4.1, sinh viên tự chứng minh $L(\hat{M}) = L_1/L_2$. Vì vậy họ NNCQ là đóng dưới phép thương đúng.

Ví dụ 4.5. Tìm L_1/L_2 cho

$$L_1 = L(a^*baa^*),$$
$$L_2 = L(ab^*).$$

Đầu tiên ta tìm dfa cho L_1 và L_2, H.4.5(a) và (b):



Hình 4.5

Từ đây không khó khăn lắm ta xác định được

$$L(M_0) \cap L_2 = \varnothing,$$
$$L(M_1) \cap L_2 = \{a\} \neq \varnothing,$$
$$L(M_2) \cap L_2 = \{a\} \neq \varnothing,$$

Vì vậy dfa cho L_1/L_2 là xác định được và như được cho trong H.4.5(c). Nó chấp nhận ngôn ngữ $L(a^*ba^*)$. Vì vậy $L_1/L_2 = L(a^*ba^*)$.

4.2. CÁC CÂU HỎI CƠ BẢN VỀ NGÔN NGỮ CHÍNH QUI

Bây giờ chúng ta đi đến một ý tưởng rất cơ bản:

- Cho một ngôn ngữ L và một chuỗi w, chúng ta có thể xác định được w có phải là một phần tử của L hay không?

Đây là câu hỏi ***thành viên*** (*membership*) và phương pháp để trả lời nó được gọi là giải thuật thành viên. Rất ít những gì có thể làm được với những ngôn ngữ mà chúng ta không thể tìm được những giải thuật thành viên hiệu quả cho nó. Câu hỏi về sự tồn tại và bản chất của giải thuật thành viên sẽ là một mối quan tâm lớn trong việc thảo luận sau này; nó thường là một ý tưởng khó. Nhưng dù vậy đối với NNCQ nó là một vấn đề dễ dàng.

Trước hết để tiện trình bày ta giới thiệu khái niệm ***biểu diễn chuẩn***.

Biểu diễn chuẩn (*standard representation*)

Chúng ta nói rằng một NNCQ là được cho trong một ***biểu diễn chuẩn*** *nếu và chỉ nếu nó được mô tả bởi một automat hữu hạn, một BTCQ hoặc bởi một VPCQ.*

Chú ý từ một dạng biểu diễn chuẩn này cho L luôn có thể xác định được các dạng biểu diễn chuẩn khác cho L nhờ vào các định lý mà ta đã chứng minh trước đây.

Định lý 4.5. *Cho một biểu diễn chuẩn của một NNCQ L bất kỳ trên* Σ *và một chuỗi w bất kỳ* $\in \Sigma^*$*, thì tồn tại một giải thuật để xác định w có ở trong L hay không.*

Chứng minh

Chúng ta biểu diễn ngôn ngữ bằng một dfa nào đó rồi kiểm tra xem w có được chấp nhận bởi dfa này không.

Một số câu hỏi quan trọng khác là

- Một ngôn ngữ đã cho là hữu hạn hay vô hạn?
- Hai ngôn ngữ có giống nhau hay không?
- Có hay không một ngôn ngữ là tập con của một ngôn ngữ khác?

Đối với các NNCQ những câu hỏi này là dễ dàng trả lời được.

Định lý 4.6. *Tồn tại một giải thuật để xác định được một NNCQ đã cho trong một dạng biểu diễn chuẩn là trống, hữu hạn hay vô hạn.*

Chứng minh

Chúng ta biểu diễn ngôn ngữ bằng một dfa. Nếu tồn tại một con đường đi từ trạng thái khởi đầu đến một trạng thái kết thúc bất kỳ thì ngôn ngữ là khác trống.

Để xác định ngôn ngữ có vô hạn hay không, ta tìm tất cả các đỉnh mà có chu trình đi qua nó. Nếu có một đỉnh bất kỳ trong số này là nằm trên một con đường đi từ trạng thái khởi đầu đến trạng thái kết thúc thì ngôn ngữ là vô hạn, ngược lại thì là hữu hạn.

Nhận xét

Thông thường người ta loại bỏ các trạng thái không đạt tới được trước rồi xét tiếp. Nếu có một trạng thái kết thúc trong dfa thì ngôn ngữ là khác trống. Nếu có thêm một chu trình trong đồ thị thì ngôn ngữ là vô hạn, ngược lại thì là hữu hạn.

Câu hỏi về sự tương đương giữa hai ngôn ngữ cũng là một ý tưởng thực tế quan trọng. Thông thường tồn tại một vài định nghĩa của các ngôn ngữ lập trình, và chúng ta cần biết hai định nghĩa là có cùng chỉ định một ngôn ngữ như nhau hay không, cho dù xuất hiện dưới các dạng khác nhau. Điều này về tổng quát là một vấn đề khó; thậm chí đối với cả NNCQ lý luận cũng không thật rõ ràng. Thường là không có khả năng lý luận trên sự so sánh từng từ – và – từ, vì điều này chỉ làm được đối với các ngôn ngữ hữu hạn. Cũng không dễ để biết được câu trả lời bằng cách

xem xét các BTCQ, VPCQ hay là dfa. Một giải pháp rất hay được ứng dụng ở đây là sử dụng các tính chất về tính đóng đã được thiết lập trước đây.

Định lý 4.7. *Cho các biểu diễn chuẩn của hai NNCQ L_1 và L_2, tồn tại một giải thuật để xác định có hay không $L_1 = L_2$.*

Chứng minh

Sử dụng L_1 và L_2 chúng ta xây dựng ngôn ngữ:

$$L_3 = (L_1 \cap \overline{L_2}) \cup (\overline{L_1} \cap L_2).$$

Theo lý thuyết tập hợp ta có $L_1 = L_2$ khi và chỉ khi $L_3 = \varnothing$. Vậy thay vì kiểm tra L_1 có bằng L_2 hay không ta chuyển về kiểm tra $L_3 = \varnothing$ hay không. Bằng tính đóng L_3 là chính qui, và chúng ta có thể tìm thấy dfa M mà chấp nhận L_3. Mà chúng ta đã có giải thuật trong định lý 4.6 để xác định xem L_3 có trống hay không. Nếu $L_3 = \varnothing$ thì $L_1 = L_2$, ngược lại thì không.

4.3. NHẬN BIẾT CÁC NGÔN NGỮ KHÔNG CHÍNH QUI

Nhận xét

Các NNCQ có thể là vô hạn tuy nhiên các NNCQ được liên kết với các automat có số trạng thái hữu hạn, nên phải chịu một vài giới hạn nào đó. Trực giác cho ta biết rằng một ngôn ngữ là chính qui chỉ nếu, trong khi xử lý một chuỗi bất kỳ, thông tin được ghi nhớ tại một ngữ cảnh bất kỳ là hữu hạn. Điều này là đúng, nhưng phải được trình bày một cách chính xác. Có một vài cách để thực hiện điều này.

1. Sử dụng nguyên lý pigeonhole (chuồng chim bồ câu)

Thuật ngữ "nguyên lý pigeonhole" (hay nhiều khi còn gọi là nguyên lý Dirichle) được các nhà toán học sử dụng để nói đến quan sát đơn giản sau đây.

Nếu chúng ta đặt n vật thể vào trong m hộp (pigeonhole), và nếu $n > m$, thì ít nhất có một hộp chứa nhiều hơn một vật thể.

Đây là một thực tế rõ ràng mà thật ngạc nhiên là có nhiều kết quả khá sâu sắc có thể rút ra từ nó.

Ví dụ 4.6. Ngôn ngữ $L = \{a^n b^n : n \geq 0\}$ có chính qui không?

Câu trả lời là không, như chúng ta sẽ chứng tỏ bằng cách sử dụng phương pháp phản chứng sau:

Giả sử L là chính qui thì tồn tại một dfa $M = (Q, \{a,b\}, \delta, q_0, F)$ nào đó cho nó.

Xét $\delta^*(q_0, a^i)$ với $i = 1, 2, 3, \ldots$ Vì có một số không giới hạn các i, nhưng chỉ có một số hữu hạn các trạng thái trong M, theo nguyên lý pigeonhole thì phải có một trạng thái nào đó, chẳng hạn q, sao cho

$$\delta^*(q_0, a^n) = q$$

và

$$\delta^*(q_0, a^m) = q,$$

với $n \neq m$. Nhưng vì M chấp nhận $a^n b^n$ nên ta có

$$\delta^*(q, b^n) = q_f \in F.$$

Kết hợp với ở trên ta suy ra

$$\begin{aligned}\delta^*(q_0, a^m b^n) &= \delta^*(q, b^n) \\ &= q_f\end{aligned}$$

Vì vậy M chấp nhận cả chuỗi $a^m b^n$ với $n \neq m$. Điều này mâu thuẫn với định nghĩa của L, suy ra L là không chính qui.

Nhận xét

Trong lý luận này, nguyên lý pigeonhole đơn giản phát biểu rằng một automat hữu hạn có một bộ nhớ hữu hạn. Để chấp nhận tất cả các chuỗi $a^n b^n$, một automat phải phân biệt giữa mọi tiếp đầu ngữ a^n và a^m. Nhưng vì chỉ có một số hữu hạn các trạng thái nội để thực hiện điều này, nên phải có một n và một m nào đó mà đối với chúng automat không thể phân biệt được.

Để sử dụng kiểu lý luận này chúng ta chuyển đổi nó thành một định lý tổng quát. Có nhiều cách để làm điều này; cách mà chúng ta đưa ra ở đây có lẽ là cách nổi tiếng nhất.

2. Bổ đề bơm (*pumping*)

Định lý 4.8. *Cho L là một NNCQ vô hạn, thì tồn tại một số nguyên dương m nào đó sao cho bất kỳ chuỗi w nào $\in L$ với $|w| \geq m$ đều tồn tại một cách phân tích w thành bộ ba*

$$w = xyz,$$

với

$$|xy| \leq m,$$

và

$$|y| \geq 1,$$

sao cho

$$w_i = xy^i z, \tag{4.2}$$

cũng thuộc L đối với mọi $i = 0, 1, 2, \ldots$

Chứng minh

Nếu L là chính qui, thì tồn tại một dfa chấp nhận nó. Lấy một dfa như thế có tập trạng thái là $q_0, q_1, q_2, \ldots, q_n$. Chọn $m = |Q| = n + 1$. Lấy một chuỗi w bất kỳ $\in L$ và $|w| \geq m$. Vì L được giả thiết là vô hạn,

nên điều này luôn có thể thực hiện được. Xét một dãy các trạng thái mà automat đi qua khi xử lý chuỗi w, giả sử là $q_0, q_i, q_j, \ldots, q_f$.

Vì $|w| = k$ suy ra dãy này có $k + 1$ phần tử. Vì $k + 1 > n + 1$ nên có ít nhất một trạng thái phải được lặp lại, và sự lặp lại này nằm trong $n + 2$ phần tử đầu tiên. Vì vậy dãy trên phải có dạng

$$q_0, q_i, q_j, \ldots, q_r, \ldots, q_r, \ldots, q_f,$$

suy ra phải có các chuỗi con x, y, z của w sao cho

$$\delta^*(q_0, x) = q_r,$$
$$\delta^*(q_r, y) = q_r,$$
$$\delta^*(q_r, z) = q_f,$$

với $|xy| \leq n + 1 = m$, vì sự lặp lại trạng thái xảy ra trong n + 2 phần tử đầu tiên, và $|y| \geq 1$. Từ điều này suy ra

$$\delta^*(q_r, xz) = q_f,$$

cũng như

$$\delta^*(q_r, xy^iz) = q_f,$$

với mọi $i = 0, 1, 2, \ldots$ Đến đây định lý được chứng minh.

Nhận xét: bổ đề bơm, giống như lý luận chuồng chim bồ câu trong ví dụ 4.6 được sử dụng để chứng minh một ngôn ngữ nào đó là không chính qui. Việc chứng minh thường bằng phản chứng.

Ví dụ 4.7. Sử dụng bổ đề bơm để chứng minh $L = \{a^nb^n : n \geq 0\}$ là không chính qui.

Giả sử L là chính qui, dễ thấy L vô hạn. Theo bổ đề bơm tồn tại số nguyên dương m.

Chọn $w = a^mb^m \in L$, $|w| = 2m \geq m$. Theo bổ đề bơm tồn tại một cách phân tích w thành bộ ba: $w = xyz$ trong đó $|xy| \leq m$ (1), $|y| = k \geq 1$ (2). Từ cách chọn w có m kí hiệu đi đầu là a, kết hợp với (1) suy ra xy chỉ chứa toàn a, từ đây suy ra y cũng chỉ chứa toàn a. Vậy $y = a^k$. Xét $w_i = xy^iz$ với $i = 0$, ta có: $w_0 = a^{n-k}b^n \in L$ theo bổ đề bơm, nhưng điều này mâu thuẫn với định nghĩa của L. Vậy L không chính qui.

Nhận xét

Lý luận này có thể được trực quan hóa như một trò chơi chúng ta đấu với một đối thủ. Mục đích của chúng ta là thắng ván chơi bằng cách tạo ra một sự mâu thuẫn của bổ đề bơm, trong khi đối thủ thử chặn đứng chúng ta. Có bốn bước đi trong trò chơi này như sau:

1. Đối thủ lấy m.
2. Với m đã cho chúng ta lấy một chuỗi $w \in L$ có chiều dài $\geq m$.

3. Đối thủ chọn phân hoạch xyz, thỏa $|xy| \leq m$, $|y| \geq 1$. Chúng ta phải giả thiết rằng đối thủ chọn lựa làm sao cho chúng ta khó thắng ván chơi nhất.

4. Chúng ta chọn i sao cho chuỗi được bơm lên không $\in L$.

Bước quyết định ở đây là bước 2. Trong khi chúng ta không thể ép buộc đối thủ lấy một phân hoạch cụ thể của chuỗi w, chúng ta có thể chọn chuỗi w sao cho đối thủ bị hạn chế nghiêm ngặt trong bước 3, ép buộc một sự chọn lựa của x, y, z sao cho cho phép chúng ta tạo ra một mâu thuẫn bổ đề bơm trên bước kế tiếp của chúng ta.

Ví dụ 4.8. Chứng minh ngôn ngữ $L = \{ww^R: w \in \{a, b\}^*\}$ là không chính qui.

Giả sử L là chính qui, dễ thấy L vô hạn. Theo bổ đề bơm tồn tại số nguyên dương m.

Chọn $w = a^m b^m b^m a^m \in L$, $|w| = 4m \geq m$.

Theo bổ đề bơm tồn tại một cách phân hoạch w thành bộ ba:

$w = xyz$; trong đó: $|xy| \leq m$ (1); $|y| = k \geq 1$ (2).

Từ (1) suy ra xy chỉ chứa toàn kí hiệu a, từ đây suy ra y cũng chỉ chứa toàn kí hiệu a. Vậy $y = a^k$.

Xét $w_i = xy^i z$ với $i = 0$, ta có: $w_0 = a^{m-k} b^m b^m a^m \in L$ theo bổ đề bơm, nhưng điều này mâu thuẫn với định nghĩa của L. Vậy L không chính qui.

Ví dụ 4.9. Chứng minh ngôn ngữ $L = \{a^n b^k c^{n+k}: n \geq 0, k \geq 0\}$ là không chính qui.

Không khó để áp dụng bổ đề bơm một cách trực tiếp, nhưng có một cách dễ dàng hơn ở đây bằng cách sử dụng phép đồng hình. Lấy

$$h(a) = a; \quad h(b) = a; \quad h(c) = c.$$

Suy ra

$$h(L) = \{a^{n+k} c^{n+k}: n + k \geq 0\} = \{a^i c^i: i \geq 0\}.$$

Mà ta đã biết ngôn ngữ này là không chính qui, vì vậy L cũng không chính qui theo tính đóng của NNCQ đối với phép đồng hình.

Ví dụ 4.10. Chứng minh ngôn ngữ $L = \{a^n b^l: n \neq l\}$ là không chính qui.

Xét

$$\begin{aligned} L_1 &= \overline{L} \cap L(a^* b^*) \\ &= \{a^n b^n: n \geq 0\} \end{aligned}$$

Mà vì L_1 không chính qui suy ra L cũng không chính qui theo tính đóng của họ NNCQ đối với phép bù và phép giao.

Chương 5

NGÔN NGỮ PHI NGỮ CẢNH

Trong chương trước, chúng ta đã biết rằng không phải tất cả các ngôn ngữ đều là chính qui. Trong khi NNCQ là hiệu quả trong việc mô tả một vài mẫu đơn giản, nó lại khá hạn chế trong việc ứng dụng vào các ngôn ngữ lập trình chẳng hạn nếu chúng ta thử diễn dịch lại một vài ví dụ. Nếu trong $L = \{a^n b^n : n \geq 0\}$ chúng ta thay thế ngoặc trái cho a và ngoặc phải cho b, thì chuỗi các dấu ngoặc chẳng hạn như (()) và ((())) là thuộc L, nhưng (() thì không. Ngôn ngữ này vì vậy mô tả một loại cấu trúc lồng nhau đơn giản được tìm thấy trong các ngôn ngữ lập trình, điều này chỉ ra rằng một vài đặc tính của các ngôn ngữ lập trình đòi hỏi một cái gì đó vượt ra ngoài họ NNCQ. Để bao trùm điều này lẫn một số đặc điểm khác phức tạp hơn chúng ta phải mở rộng nghiên cứu họ các ngôn ngữ. Điều này dẫn ta đến việc nghiên cứu các ngôn ngữ và các văn phạm phi ngữ cảnh (VPPNC).

Chúng ta bắt đầu chương này bằng việc định nghĩa văn phạm và ngôn ngữ phi ngữ cảnh (NNPNC) (*Context Free Languages – CFL*). Tiếp theo, chúng ta xem xét các bài toán thành viên quan trọng; cụ thể là làm thế nào để biết một chuỗi đã cho có được dẫn xuất ra từ một văn phạm đã cho hay không. Việc cắt nghĩa một câu thông qua dẫn xuất của văn phạm là quen thuộc đối với hầu hết chúng ta và việc này được gọi là ***sự phân tích cú pháp*** (*parsing*) (PTCP). PTCP là một cách mô tả cấu trúc câu, rất quan trọng khi chúng ta cần hiểu nghĩa của một câu, chẳng hạn như khi chúng ta dịch câu từ một ngôn ngữ này sang một ngôn ngữ khác. Trong khoa học máy tính, điều này là thích hợp trong các trình thông dịch, biên dịch, và các trình dịch khác.

Chủ đề về NNPNC, có lẽ là mặt quan trọng nhất của lý thuyết ngôn ngữ hình thức vì nó được ứng dụng vào các ngôn ngữ lập trình. Các ngôn ngữ lập trình thực sự có nhiều đặc điểm mà có thể được mô tả một cách đẹp đẽ bằng các phương tiện của NNPNC. Những gì mà lý thuyết ngôn ngữ hình thức nói về NNPNC có một ứng dụng quan trọng trong việc thiết kế các ngôn ngữ lập trình cũng như trong việc xây dựng các trình biên dịch hiệu quả. Chúng ta sẽ đề cập ngắn gọn đến vấn đề này trong phần 5.3.

5.1. VĂN PHẠM PHI NGỮ CẢNH (*CONTEXT FREE GRAMMARS*)

Các luật sinh trong VPCQ là bị giới hạn ở hai chỗ: vế trái của nó là một biến đơn, trong khi vế phải có một dạng đặc biệt. Để tạo ra văn phạm mạnh hơn, chúng ta phải nới lỏng một trong các giới hạn này. Bằng cách duy trì giới hạn trên vế trái, nhưng cho phép bất kỳ cái gì trên vế phải, chúng ta nhận được VPPNC.

Định nghĩa 5.1. *Một văn phạm $G = (V, T, S, P)$ được gọi là phi ngữ cảnh nếu mọi luật sinh trong P có dạng*

$$A \to x,$$

trong đó: $A \in V$ còn $x \in (V \cup T)^*$.

Một ngôn ngữ được gọi là phi ngữ cảnh nếu và chỉ nếu có một VPPNC G sao cho $L = L(G)$.

Nhận xét: mọi NNCQ đều là PNC, nhưng điều ngược lại thì không. Như chúng ta sẽ thấy sau này họ NNCQ là một tập con thực sự của họ NNPNC.

Ghi chú: VPPNC có tên của nó bắt nguồn từ nguyên nhân rằng sự thay thế của biến ở vế trái của luật sinh có thể được thực hiện bất kỳ khi nào biến đó xuất hiện trong dạng câu mà không phụ thuộc vào các vị trí còn lại của dạng câu (ngữ cảnh). Đặc điểm này là kết quả của việc chỉ cho phép có một biến đơn trên vế trái của luật sinh.

1. Các ví dụ về ngôn ngữ phi ngữ cảnh

Ví dụ 5.1. Văn phạm $G = (\{S\}, \{a, b\}, S, P)$, có các luật sinh

$$S \to aSa,$$
$$S \to bSb,$$
$$S \to \lambda,$$

là PNC. Một dẫn xuất điển hình trong văn phạm này là

$$S \Rightarrow aSa \Rightarrow aaSaa \Rightarrow aabSbaa \Rightarrow aabbaa$$

Dễ thấy

$$L(G) = \{ww^R: w \in \{a, b\}^*\}$$

Nhận xét: văn phạm trong ví dụ trên không những là PNC mà còn là tuyến tính. Các VPCQ và tuyến tính rõ ràng là PNC, nhưng một VPPNC không nhất thiết là tuyến tính.

Ví dụ 5.2. Ngôn ngữ

$$L = \{a^n b^n: n \geq 0\} \text{ là PNC.}$$

VPPNC cho ngôn ngữ này là

$$S \to aSb|\lambda$$

Ví dụ 5.3. Ngôn ngữ

$$L = \{a^n b^m : n \neq m\} \text{ là PNC.}$$

Ta xét hai trường hợp:

- Trường hợp $n > m$, dựa vào văn phạm trong ví dụ 5.2 ta xây dựng văn phạm cho trường hợp này là

$$S \rightarrow AS_1$$
$$S_1 \rightarrow aS_1b|\lambda$$
$$A \rightarrow aA|a$$

- Làm tương tự cho trường hợp $n < m$, kết quả ta có văn phạm cho ngôn ngữ trên là

$$S \rightarrow AS_1|S_1B$$
$$S_1 \rightarrow aS_1b|\lambda$$
$$A \rightarrow aA|a$$
$$B \rightarrow bB|b$$

Ví dụ 5.4. Xét văn phạm sau

$$S \rightarrow aSb|SS|\lambda.$$

Văn phạm này sinh ra ngôn ngữ sau:

$$L = \{w \in \{a, b\}^* : n_a(w) = n_b(w) \text{ và } n_a(v) \geq n_b(v),$$

với v là một tiếp đầu ngữ bất kỳ của $w\}$

Nhận xét: nếu trong ngôn ngữ trên thay a bằng dấu mở ngoặc (, b bằng dấu đóng ngoặc), thì ngôn ngữ sẽ tương ứng với cấu trúc ngoặc lồng nhau, chẳng hạn (( )) hay (( ) ( )), phổ biến trong các ngôn ngữ lập trình.

2. Dẫn xuất trái nhất và phải nhất

Trong VPPNC mà không tuyến tính, một dẫn xuất có thể bao gồm nhiều dạng câu với nhiều hơn một biến. Trong trường hợp như vậy, chúng ta có một sự lựa chọn về thứ tự biến để thay thế. Lấy ví dụ văn phạm $G = (\{A, B, S\}, \{a,b\}, S, P)$ với các luật sinh

1. $S \rightarrow AB$,
2. $A \rightarrow aaA$,
3. $A \rightarrow \lambda$,
4. $B \rightarrow Bb$,
5. $B \rightarrow \lambda$.

Dễ dàng thấy rằng văn phạm này sinh ra ngôn ngữ

$$L(G) = \{a^{2n}b^m : n \geq 0, m \geq 0\}$$

Bây giờ xét hai dẫn xuất

$$S \overset{1}{\Rightarrow} AB \overset{2}{\Rightarrow} aaAB \overset{3}{\Rightarrow} aaB \overset{4}{\Rightarrow} aaBb \overset{5}{\Rightarrow} aab$$

$$S \overset{1}{\Rightarrow} AB \overset{4}{\Rightarrow} ABb \overset{2}{\Rightarrow} aaABb \overset{5}{\Rightarrow} aaAb \overset{3}{\Rightarrow} aab.$$

Để trình bày dẫn xuất nào được sử dụng, chúng ta đã đánh số các luật sinh và ghi số thích hợp trên kí hiệu dẫn xuất $\Rightarrow$. Từ đây chúng ta thấy rằng hai dẫn xuất không chỉ tạo ra cùng một câu mà còn sử dụng chính xác các luật sinh giống nhau chỉ khác biệt về thứ tự mà các luật sinh được áp dụng. Để loại bỏ các yếu tố không quan trọng như thế, chúng ta thường yêu cầu rằng các biến được thay thế trong một thứ tự chỉ định. Từ đây chúng ta đưa ra định nghĩa sau:

Định nghĩa 5.2. *Một dẫn xuất được gọi là* ***trái nhất*** *(DXTN – leftmost derivation) nếu trong mỗi bước biến bên trái nhất trong dạng câu được thay thế. Nếu trong mỗi bước biến phải nhất được thay thế, chúng ta gọi là* ***dẫn xuất phải nhất*** *(DXPN – rightmost derivation).*

Ví dụ 5.5. Xét văn phạm với các luật sinh (được đánh chỉ số bên tay phải)

$$S \to aAB, \ 1$$

$$A \to bBb, \ 2$$

$$B \to A|\lambda, \ 3,4$$

thì

$$S \overset{1}{\Rightarrow} aAB \overset{2}{\Rightarrow} abBbB \overset{3}{\Rightarrow} abAbB \overset{2}{\Rightarrow} abbBbbB \overset{4}{\Rightarrow} abbbbB \overset{4}{\Rightarrow} abbbb$$

là một DXTN của chuỗi *abbbb*. Một DXPN của chuỗi này là

$$S \overset{1}{\Rightarrow} aAB \overset{4}{\Rightarrow} aA \overset{2}{\Rightarrow} abBb \overset{3}{\Rightarrow} abAb \overset{2}{\Rightarrow} abbBbb \overset{4}{\Rightarrow} abbbb$$

DXTN và DXPN có lợi điểm là trong nhiều trường hợp chúng ta không cần trình bày cụ thể dẫn xuất của một câu thì ta chỉ cần trình bày dãy số hiệu luật sinh được áp dụng lần lượt để sinh ra câu đó. Từ dãy số hiệu luật sinh này hoàn toàn có thể trình bày ra được dẫn xuất cụ thể mà không bị nhầm khi đã biết vào mỗi thời điểm dẫn xuất chính xác biến nào sẽ được thay thế và được thay thế bằng luật sinh gì. Chẳng hạn trong trường hợp trên chúng ta có thể ghi:

DXTN của *abbbb* là 123244.

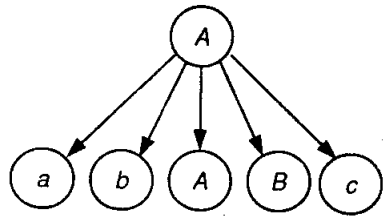
DXPN của *abbbb* là 142324.

3. Cây dẫn xuất (*derivation trees*)

Một cách thứ hai để trình bày các dẫn xuất, độc lập với thứ tự các luật sinh được áp dụng, là bằng ***cây dẫn xuất*** **(CDX).** Một CDX là một

cây có thứ tự trong đó các nốt được gán nhãn với vế trái của luật sinh còn các con của các nốt biểu diễn vế phải tương ứng của nó. Chẳng hạn, H.5.1 trình bày một phần của CDX biểu diễn luật sinh:

$$A \to abABc$$



Hình 5.1

Định nghĩa 5.3. *Cho $G = (V, T, S, P)$ là một VPPNC. Một cây có thứ tự là một cây dẫn xuất cho G nếu và chỉ nếu có các tính chất sau:*

1. *Gốc được gán nhãn là S.*
2. *Mỗi lá có một nhãn lấy từ tập $T \cup \{\lambda\}$.*
3. *Mỗi nốt bên trong (không phải là lá) có một nhãn lấy từ V.*
4. *Nếu mỗi nốt có nhãn $A \in V$, và các con của nó được gán nhãn (từ trái sang phải) $a_1, a_2, \ldots, a_n$, thì P phải chứa một luật sinh có dạng*

$$A \to a_1 a_2 \ldots a_n$$

5. *Một lá được gán nhãn λ không có anh chị em, tức là, một nốt với một con được gán nhãn λ có thể không có con nào khác.*

Một cây mà có các tính chất 3, 4 và 5, nhưng trong đó tính chất 1 không nhất thiết được giữ còn tính chất 2 được thay thế bằng 2a.

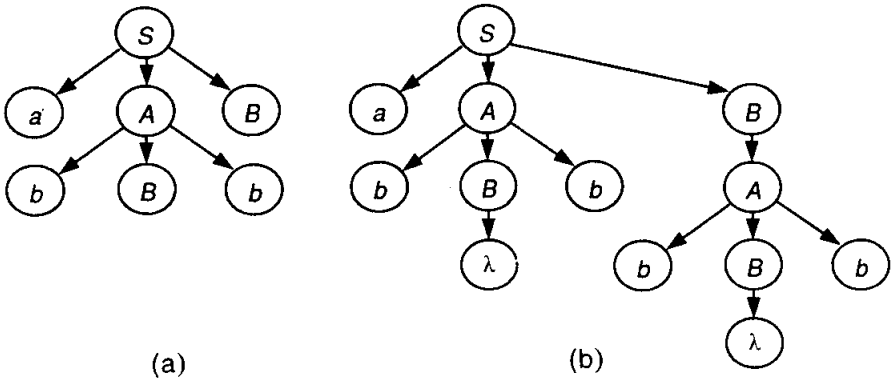
2a. Mỗi lá có một nhãn lấy từ tập $V \cup T \cup \{\lambda\}$ thì được gọi là một ***cây dẫn xuất riêng phần*** *(CDXRP).*

Chuỗi kí hiệu nhận được bằng cách đọc các nốt lá của cây từ trái sang phải, bỏ qua bất kỳ λ nào được bắt gặp, được gọi là ***kết quả*** *(yield) của cây.*

Ví dụ 5.6. Xét văn phạm G với các luật sinh sau

$$S \to aAB,$$
$$A \to bBb,$$
$$B \to A|\lambda,$$

Cây trong H.5.2(a) là một CDXRP đối với G, trong khi cây trong H.5.2(b) là một CDX. Kết quả của cây thứ nhất, chuỗi $abBbB$ là một dạng câu của G. Kết quả của cây thứ hai, chuỗi $abbbb$ là một câu của $L(G)$.



Hình 5.2

Ghi chú: đôi lúc người ta còn gọi cây dẫn xuất là ***cây cú pháp*** (*parse tree*).

4. Mối quan hệ giữa dạng câu và cây dẫn xuất

Nhận xét

CDX đưa ra một mô tả của dẫn xuất rất tường minh và dễ hiểu. Giống như ĐTCTT cho automat hữu hạn, sự tường minh là một sự giúp đỡ lớn trong việc thực hiện lý luận. Tuy vậy, đầu tiên chúng ta phải thiết lập một quan hệ giữa dẫn xuất và CDX.

Định lý 5.1. *Cho $G = (V, T, S, P)$ là một VPPNC, thì đối với mọi $w \in L(G)$, tồn tại một CDX của G mà kết quả của nó là w. Ngược lại, kết quả của một CDX bất kỳ là thuộc $L(G)$. Tương tự, nếu t_G là một CDXRP bất kỳ của G mà gốc của nó được gán nhãn là S thì kết quả của t_G là một dạng câu của G.*

Chứng minh

Sinh viên tự chứng minh bằng qui nạp.

5.2. PHÂN TÍCH CÚ PHÁP VÀ TÍNH NHẬP NHẰNG

Từ trước đến giờ chúng ta đã tập trung trên các khía cạnh sinh ra của văn phạm: cho một văn phạm G, chúng ta nghiên cứu tập các chuỗi mà có thể được dẫn xuất bằng G. Trong trường hợp các ứng dụng thực tế, chúng ta cũng cần quan tâm đến khía cạnh phân tích của văn phạm: cho một chuỗi w các kí tự kết thúc, chúng ta muốn biết w có ở trong $L(G)$ hay không. Nếu ở trong, chúng ta có thể muốn tìm một dẫn xuất của w. Một giải thuật mà có thể cho chúng ta biết w có ở trong $L(G)$ hay không là một giải thuật thành viên. Thuật ngữ ***phân tích cú pháp*** (*parsing*) (PTCP) mô tả việc tìm một dãy các luật sinh mà một chuỗi $w \in L(G)$ được dẫn xuất ra.

Phân tích cú pháp *(pasing hay syntax analysis)*

Phân tích cú pháp là quá trình xác định một chuỗi có được sinh ra bởi một văn phạm nào đó hay không, mà cụ thể hơn là quá trình tìm một CDX cho chuỗi đó.

Kết quả của quá trình phân tích cú pháp rơi vào một trong hai khả năng "yes" hoặc "no". "Yes" có nghĩa là chuỗi được sinh ra bởi văn phạm, trong trường hợp này thường kèm theo một hay một số dẫn xuất sinh ra chuỗi. "No" có nghĩa là chuỗi không được sinh ra bởi văn phạm. Trong trường hợp này người ta còn gọi là chuỗi không đúng cú pháp hay có lỗi (*error*).

Các giải thuật phân tích cú pháp thường có dạng như sau:

Input : $G = (V, T, S, P)$ và chuỗi w cần phân tích

Output: "yes" hay "no". Trong trường hợp "yes" thường có kèm theo các DXTN hay DXPN của chuỗi.

Về PTCP, thì có nhiều phương pháp PTCP, tuy nhiên chúng thường thuộc một trong hai trường phái phân tích sau.

PTCP từ trên xuống *(top–down parsing)* ***và từ dưới lên*** *(bottom–up parsing)*

PTCP từ trên–xuống là quá trình xây dựng CDX từ gốc xuống lá. Còn PTCP từ dưới–lên là quá trình xây dựng CDX từ lá lên gốc.

Bây giờ ta sẽ trình bày những vấn đề cơ bản trong việc phân tích cú pháp từ trên xuống thông qua ví dụ sau:

Ví dụ 5.7. Cho văn phạm G sau:

$$S \to aAbS|bBS|\lambda \quad (1, 2, 3)$$

$$A \to aAA|aS|b \quad (4. 5. 6)$$

$$B \to bBB|bS|a \quad (7, 8, 9)$$

Hãy PTCP từ trên xuống cho chuỗi sau:

$$w = aabbbba$$

Thông thường để thuận tiện cho việc PTCP các luật sinh đều được đánh số như ví dụ trên trình bày và các số này được gọi là các số hiệu luật sinh. Như đã nói ở trên quá trình PTCP từ trên xuống là quá trình tìm ra dẫn xuất cho chuỗi cần phân tích mà quá trình này được bắt đầu bằng kí hiệu mục tiêu S. Và quá trình tìm dẫn xuất là quá trình thay thế biến trong dạng câu để đi từ dạng này sang dạng câu khác chi tiết hơn cho đến khi đi đến được một dạng câu đặc biệt đó là chuỗi cần phân tích nếu được và lúc đó quá trình phân tích được gọi là thành công, còn nếu

ngược lại không có cách nào để đi đến được chuỗi cần phân tích thì quá trình phân tích được gọi là gặp lỗi và như vậy chuỗi cần phân tích không được sinh ra bởi văn phạm hay nói như trong ngôn ngữ tự nhiên là ***không đúng cú pháp*** so với văn phạm. Để thực hiện việc PTCP từ trên xuống chúng ta giới thiệu hai đầu đọc, một đầu đọc đọc trên chuỗi kí hiệu nhập và di chuyển từ trái sang phải. Một đầu đọc đọc trên các dạng câu lần lượt được hình thành trong quá trình phân tích và cũng di chuyển từ trái sang phải. Vào thời điểm khởi đầu, đầu đọc 1 nằm ở vị trí khởi đầu của chuỗi nhập, đầu đọc 2 nằm ở vị trí khởi đầu của dạng câu thứ nhất chính là kí hiệu mục tiêu S. Ta thể hiện mỗi đầu đọc bằng một dấu chấm · tương ứng. Theo ví dụ trên thì vào thời điểm khởi đầu chúng ta có hình ảnh:

	Khởi đầu
Chuỗi nhập	·*aabbbba*
Dạng câu	·*S*

Để phân tích cú pháp ta phải thay thế các biến trong dạng câu bằng các vế phải của chúng mà cụ thể là biến nằm ngay bên vế trái của dấu chấm nếu có. Trong ví dụ này đó là thay thế S. Và vấn đề cốt lõi của việc PTCP từ trên xuống xuất hiện đó là dựa vào các kí hiệu kết thúc bên phải dấu chấm trên chuỗi nhập mà thông thường là dựa vào kí hiệu ngay bên phải dấu chấm và dựa vào biến ngay bên phải dấu chấm dưới dạng câu (nếu có) ta phải ***quyết định chọn vế phải nào*** trong các vế phải của biến cần thay thế để thay thế cho biến đó mà ***có khả năng nhất*** để sinh ra được chuỗi nhập. Như trong ví dụ này ta thấy kí hiệu kết thúc nằm ngay bên phải đầu đọc 1 là a và biến cần được thay thế là S. Trong các vế phải của S ta thấy vế phải trong luật sinh số 1 là có khả năng nhất, hai vế phải trong hai luật sinh số 2 và 3 không có khả năng sinh ra được chuỗi nhập. Sau khi thay thế ta quá trình phân tích có dạng:

	Khởi đầu	1
Chuỗi nhập	·*aabbbba*	·*aabbbba*
Dạng câu	·*S*	·*aAbS*

Số 1 chỉ ra số hiệu luật sinh được áp dụng. Đến đây hai kí hiệu kết thúc tương ứng ở ngay bên phải hai dấu chấm so trùng nhau vì vậy hai đầu đọc cùng tiến lên một bước tức là di chuyển sang phải một kí tự. Chúng ta có hình ảnh:

	Khởi đầu	1	
Chuỗi nhập	·*aabbbba*	·*aabbbba*	*a*·*abbbba*
Dạng câu	·*S*	·*aAbS*	*a*·*AbS*

Đến đây kí hiệu kết thúc ngay bên phải của đầu đọc 1 là a còn kí hiệu ngay bên phải đầu đọc 2 là A. Trong các vế phải của A ta thấy cả hai vế phải của hai luật sinh 4 và 5 đều có khả năng. Vậy vấn đề là ta chọn vế phải nào để thay thế hay cả hai vế phải đều có khả năng để sinh ra chuỗi nhập. Đây chính là vấn đề quan trọng nhất của quá trình PTCP từ trên xuống. Để giải quyết vấn đề này tất cả các phương pháp PTCP đều thực hiện việc vét cạn tất cả các khả năng có thể, và rồi tiếp tục quá trình, cho đến một lúc nào đó sẽ biết được khả năng nào đúng đắn hơn khả năng nào hay có nhiều khả năng cùng dẫn xuất ra chuỗi nhập. Tuy tất cả các phương pháp đều vét cạn nhưng mỗi phương pháp có một kỹ thuật vét cạn khác nhau dẫn đến độ phức tạp của các phương pháp cũng khác nhau. Như ở bên dưới này chúng ta có trình bày một phương pháp PTCP vét cạn cơ bản nhất và cũng thô sơ nhất. Ở cuối chương này và ở phần Phụ lục chúng ta có trình bày hai phương pháp PTCP khác hiệu quả hơn rất nhiều mặc dù vẫn phải vét cạn mọi khả năng có thể tại mọi thời điểm PTCP. Quay trở lại ví dụ này, giả sử bằng một cách nào đó ta biết được vế phải của luật sinh số 4 có khả năng hơn vì vậy đến lúc này ta có hình ảnh của quá trình PTCP như sau:

	Khởi đầu	1		4
Chuỗi nhập	·*aabbbba*	·*aabbbba*	*a*·*abbbba*	*a*·*abbbba*
Dạng câu	·*S*	·*aAbS*	*a*·*AbS*	*a*·*aAAbS*

Tiếp tục theo khuôn mẫu này, bỏ qua chi tiết làm thế nào để lựa chọn được vế phải đúng đắn để thay thế cho biến đang cần thay thế ta có toàn bộ quá trình PTCP của chuỗi w trên diễn ra như sau:

	Khởi đầu	1		4		6
Chuỗi nhập	·*aabbbba*	·*aabbbba*	*a*·*abbbba*	*a*·*abbbba*	*aa*·*bbbba*	*aa*·*bbbba*
Dạng câu	·*S*	·*aAbS*	*a*·*AbS*	*a*·*aAAbS*	*aa*·*AAbS*	*aa*·*bAbS*
		6			2	
Chuỗi nhập	*aab*·*bbba*	*aab*·*bbba*	*aabb*·*bba*	*aabbb*·*ba*	*aabbb*·*ba*	*aabbbb*·*a*
Dạng câu	*aab*·*AbS*	*aab*·*bbS*	*aabb*·*bS*	*aabbb*·*S*	*aabbb*·*bBS*	*aabbbb*·*BS*
	9		3			
Chuỗi nhập	*aabbbb*·*a*	*aabbbba*·	*aabbbba*·			
Dạng câu	*aabbbb*·*aS*	*aabbbba*·*S*	*aabbbba*·			

Vậy kết quả của quá trình phân tích trên là dãy số hiệu luật sinh 1.4.6.6.2.9.3 cái mà mô tả dẫn xuất trái nhất của chuỗi.

Còn sau đây là một ví dụ về phân tích cú pháp từ dưới lên.

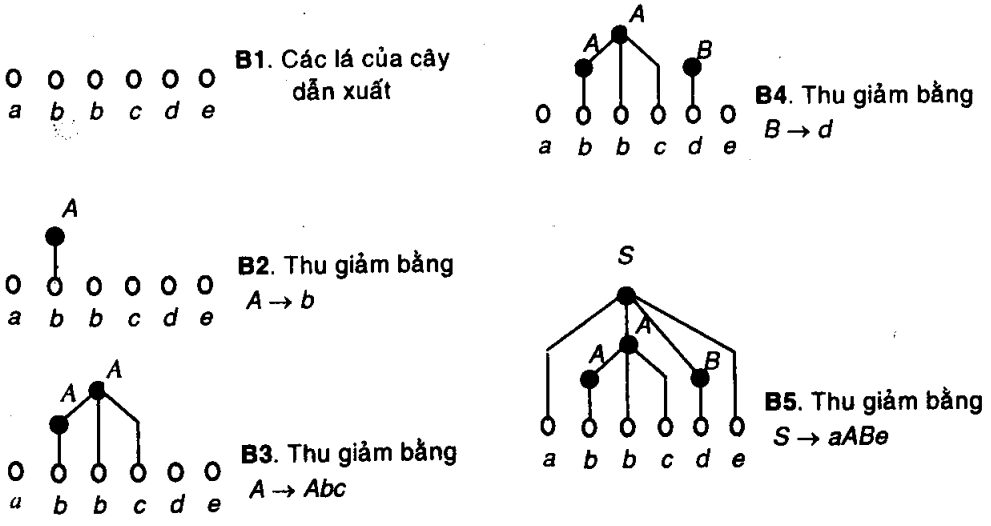
Ví dụ 5.8. Cho văn phạm G sau:

$$S \to aABe$$
$$A \to Abc|b$$
$$B \to d$$

Hãy PTCP từ dưới lên cho chuỗi sau:

$$w = abbcde.$$

Quá trình xây dựng cây dẫn xuất cho w từ dưới lên theo các bước như sau:



Hình 5.3

Kết quả ta có dãy dẫn xuất ngược của nó là

$$abbcde \Leftarrow aAbcde \Leftarrow aAde \Leftarrow aABe \Leftarrow S$$

Nếu viết theo chiều ngược lại thì đó chính là dẫn xuất phải nhất của chuỗi:

$$S \Rightarrow aABe \Rightarrow aAde \Rightarrow aAbcde \Rightarrow abbcde$$

Vì vậy người ta cũng thường nói PTCP từ dưới lên chính là quá trình tìm DXPN của chuỗi. Còn PTCP từ trên xuống là quá trình tìm DXTN của chuỗi.

Qua ví dụ trên cho ta thấy vấn đề quan trọng nhất của phương pháp PTCP từ dưới lên là trong số các chuỗi thành phần của dạng câu có khả năng để thu gọn, chọn thành phần nào có ***khả năng nhất để thu***

gọn được về thành biến mục tiêu. Dĩ nhiên để xác định được khả năng đúng đắn nhất để thu gọn người ta cũng dùng kỹ thuật vét cạn. Các phương pháp PTCP từ dưới lên không được trình bày trong sách này mà được cho trong một tài liệu khác sẽ được nêu ở bên dưới.

Hai phương pháp trên là hai phương pháp phân tích chủ yếu đối với các ngôn ngữ lập trình hiện tại. Tương ứng với hai loại phương pháp này có hai lớp văn phạm thông dụng mà thường được sử dụng trong việc định nghĩa các ngôn ngữ lập trình. Đó là văn phạm $LL(k)$ và $LR(k)$. Văn phạm $LL(k)$ thích hợp với phương pháp phân tích cú pháp từ trên xuống còn $LR(k)$ thích hợp với phương pháp phân tích cú pháp từ dưới lên. Đa số các giải thuật PTCP thuộc loại PTCP từ trên xuống, tuy nhiên phương pháp PTCP từ dưới lên có những ưu điểm vượt trội so với phương pháp PTCP từ trên xuống. Những điều này sẽ được đề cập sâu hơn và chính qui hơn trong lý thuyết về trình biên dịch và sẽ được tác giả trình bày trong một tài liệu về *CÁC PHƯƠNG PHÁP PHÂN TÍCH CÚ PHÁP* cho các lớp ngôn ngữ.

1. Phân tích cú pháp và thành viên (*membership*)

Trong phần này chúng ta sẽ giới thiệu một phương pháp PTCP cơ bản nhất nhưng cũng thô sơ nhất.

Cho một chuỗi w thuộc $L(G)$, chúng ta có thể PTCP nó trong một kiểu rõ ràng: chúng ta xây dựng một cách có hệ thống tất cả các dẫn xuất có thể (chẳng hạn, trái nhất) và xem có một dẫn xuất nào đó trong chúng so trùng với w hay không. Cụ thể, chúng ta bắt đầu bằng cách:

- ***Ở lượt*** (*round*) thứ nhất xem xét tất cả các luật sinh có dạng

$$S \to x,$$

tìm tất cả các x mà có thể được dẫn xuất từ S bởi một bước.

- Nếu không có kết quả nào trong số này so trùng với w, chúng ta sẽ đi tiếp đến lượt tiếp theo, trong đó chúng ta áp dụng tất cả các luật sinh có thể tới biến bên trái nhất của mỗi x. Thao tác này sẽ gởi ra cho ta một tập các dạng câu, một vài trong chúng có khả năng dẫn tới w.

- Trong mỗi lượt kế tiếp, chúng ta lại lấy tất cả các biến trái nhất và áp dụng tất cả các luật sinh có thể. Có thể rằng một số trong các dạng câu này có thể bị từ chối vì lý do rằng w không bao giờ có thể được dẫn xuất từ chúng, nhưng trong tổng quát, chúng ta sẽ có trên mỗi lượt một tập các dạng câu có thể. Sau lượt đầu tiên, chúng ta có các dạng câu mà có thể được dẫn xuất từ S bằng cách áp dụng một

luật sinh, sau lượt thứ hai chúng ta có các dạng câu mà có thể được dẫn xuất từ S với hai luật sinh .... Nếu $w \in L(G)$, thì nó phải có một DXTN có độ dài hữu hạn. Vì vậy phương pháp này cuối cùng sẽ tìm ra được một DXTN của w.

Chúng ta gọi phương pháp này là phương pháp ***phân tích cú pháp vét cạn*** (*exhaustive search parsing*) (PTCPVC). Nó thuộc dạng ***phân tích cú pháp từ trên-xuống*** (*top–down parsing*), cái mà chúng ta đã có giới thiệu ở trên.

Ví dụ 5.9. Xét văn phạm

$$S \to SS|aSb|bSa|\lambda \text{ 1, 2, 3,4}$$

và chuỗi $w = aabb$. Lượt thứ nhất gởi ra cho chúng ta

1. $S \Rightarrow SS$
2. $S \Rightarrow aSb$
3. $S \Rightarrow bSa$
4. $S \Rightarrow \lambda$

Hai trường hợp cuối 3 và 4 có thể loại bỏ không xét tiếp vì không thể dẫn xuất ra được w. Tiếp tục, lượt thứ hai sinh ra các dạng câu sau:

1.1 $S \Rightarrow SS \Rightarrow SSS$
1.2 $S \Rightarrow SS \Rightarrow aSbS$
1.3 $S \Rightarrow SS \Rightarrow bSaS$
1.4 $S \Rightarrow SS \Rightarrow S$
2.1 $S \Rightarrow aSb \Rightarrow aSSb$
2.2 $S \Rightarrow aSb \Rightarrow aaSbb$
2.3 $S \Rightarrow aSb \Rightarrow abSab$
2.4 $S \Rightarrow aSb \Rightarrow ab$

Một lần nữa ta thấy có thể loại bỏ các trường hợp 1.3, 2.3, 2.4. Trường hợp 1.4 cũng được loại bỏ vì quay về lại một dạng câu mà đã được xét trước đó. Trên lượt kế tiếp ta tìm thấy chuỗi đích thực sự từ dãy

2.3.4 $S \Rightarrow aSb \Rightarrow abSab \Rightarrow abab$

Vì vậy $aabb$ là ở trong ngôn ngữ được sinh ra bởi văn phạm đang xét.

Nhận xét

Phương pháp PTCPVC có các nhược điểm nghiêm trọng sau:

1. Không hiệu quả, nó không được sử dụng ở những nơi mà cần đến sự PTCP hiệu quả.

2. Nhưng tính hiệu quả thậm chí chỉ là vấn đề thứ hai, có một vấn đề khác cần nói hơn. Trong khi phương pháp này luôn phân tích ra được những chuỗi $w \in L(G)$, nó có khả năng không bao giờ kết thúc đối với các chuỗi không nằm trong $L(G)$. Chẳng hạn với $w = abb$, phương pháp này sẽ đi đến việc sinh ra vô hạn các dạng câu mà không dừng lại, trừ phi chúng ta xây dựng thêm trong nó một cách nào đó để dừng lại.

Vấn đề không kết thúc của phương pháp PTCPVC là tương đối dễ khắc phục nếu chúng ta giới hạn dạng mà văn phạm có thể có. Nếu chúng ta khảo sát ví dụ 5.9 chúng ta thấy rằng khó khăn đến từ luật sinh $S \rightarrow \lambda$; luật sinh này có thể được sử dụng để làm giảm độ dài của của dạng câu kế tiếp, vì vậy chúng ta không thể nói một cách dễ dàng khi nào thì phương pháp này dừng lại. Nếu chúng ta không có bất kỳ luật sinh nào như vậy, thì chúng ta có ít khó khăn hơn. Thực tế có hai loại luật sinh chúng ta muốn loại trừ, những luật sinh có dạng $A \rightarrow \lambda$ cũng như các luật sinh có dạng $A \rightarrow B$. Như chúng ta sẽ thấy trong chương tiếp theo, sự giới hạn này (loại trừ các luật sinh vừa nói) không ảnh hưởng đến sức mạnh của những văn phạm kết quả về bất kỳ khía cạnh nào.

Ví dụ 5.10. Văn phạm

$$S \rightarrow SS|aSb|bSa|ab|ba$$

thỏa mãn các yêu cầu đã nêu. Nó sinh ra ngôn ngữ trong ví dụ 5.9 trừ chuỗi trống.

Cho một chuỗi bất kỳ $w \in \{a, b\}^+$, phương pháp PTCPVC sẽ luôn luôn kết thúc sau không quá $(|w| - 1)$ lượt. Điều này dễ hiểu vì chiều dài của dạng câu tăng ít nhất là một kí hiệu sau mỗi lượt.

Sau lượt thứ $(|w| - 1)$ chúng ta hoặc tạo ra được sự PTCP hoặc biết $w \notin L(G)$.

Ý tưởng trong ví dụ này có thể được tổng quát hóa thành định lý sau.

Định lý 5.2. *Giả sử rằng $G = (V, T, S, P)$ là một VPPNC mà không có bất kỳ luật sinh nào có dạng:*

$$A \rightarrow \lambda,$$

hay

$$A \rightarrow B,$$

trong đó: $A, B \in V$, thì phương pháp PTCPVC có thể được hiện thực thành một giải thuật mà đối với bất kỳ $w \in \Sigma^*$, hoặc tạo ra được sự phân tích cú pháp của w, hoặc biết rằng không có sự PTCP nào là có thể cho nó.

Chứng minh

Đối với mỗi dạng câu, xét cả chiều dài lẫn số kí hiệu kết thúc. Mỗi bước trong dẫn xuất tăng tối thiểu một trong hai đại lượng trên. Vì cả chiều dài lẫn số kí hiệu kết thúc đều không vượt quá $|w|$, nên quá trình dẫn xuất sẽ không nhiều hơn $(2|w| - 1)$ lượt, tại thời điểm đó chúng ta hoặc có một sự PTCP thành công hoặc w không thể được sinh ra bởi văn phạm.

Nhận xét

Phương pháp PTCPVC này không hiệu qua ở chỗ, nếu chúng ta có $|P|$ luật sinh thì ở lượt thứ nhất có khả năng sinh ra $|P|$ dạng câu, cũng như $|P|^2$ ở lượt thứ hai .... Như vậy toàn bộ việc phân tích có thể sinh ra $\left|P^{(2|w|-1)}\right|$ dạng câu. Phương pháp này rõ ràng là quá tốn kém, vì vậy chúng ta cần tìm các phương pháp tốt hơn (nếu có) để có ý nghĩa thực tế hơn. Không may thay, các PPPTVP hiệu quả không rõ ràng và tất cả các giải thuật được biết đều khá phức tạp. Đây là một chủ đề thuộc khóa học về trình biên dịch, vì vậy chúng ta sẽ không theo đuổi nó ở đây. Chúng ta chỉ sẽ đề cập đến một vài kết quả độc lập để mang lại cho độc giả một chút hương vị của chủ đề lớn rộng này.

Định lý 5.3. *Đối với mọi VPPNC tồn tại một giải thuật mà phân tích một chuỗi w bất kỳ có* $\in L(G)$ *không trong một số bước tỉ lệ với* $|w|^3$.

Có một vài phương pháp được biết để thực hiện điều này, nhưng tất cả trong số chúng là khá phức tạp mà chúng ta không thể mô tả chúng mà không cần đến việc phát triển thêm một vài kết quả nữa. Trong phần 6.3 chúng ta sẽ trở lại với câu hỏi này một lần nữa một cách ngắn gọn.

Một lý do nữa để không theo đuổi vấn đề này ở đây trong chi tiết là vì những giải thuật này thậm chí cũng không làm chúng ta thỏa mãn. Một phương pháp mà trong đó công việc tăng theo lũy thừa bậc ba độ dài của chuỗi là khá không hiệu quả, và một trình biên dịch dựa trên đó sẽ cần một lượng thời gian quá lớn để PTCP cho thậm chí một chương trình có độ dài trung bình. Những gì mà chúng ta mong muốn là một phương pháp PTCP mà chiếm thời gian tỉ lệ với độ dài của chuỗi. Chúng ta gọi phương pháp như vậy là phương pháp PTCP ***thời gian tuyến tính***. Trong tổng quát, chúng ta không biết một phương pháp PTCP thời gian tuyến tính nào cho NNPNC, nhưng các phương pháp như thế có thể được tìm thấy cho các trường hợp đặc biệt, có giới hạn nhưng quan trọng.

Ví dụ 5.11. Ta giới thiệu ra ở đây một loại văn phạm có tính chất đặc biệt sau:

Văn phạm–s *(simple grammar)*

Là một VPPNC trong đó các luật sinh có dạng

$$A \to ax$$

trong đó: $A \in V$, $a \in T$, $x \in V^*$, *và đối với mọi cặp* (A, a) *chỉ có thể xuất hiện tối đa trên một luật sinh. Hay nói cách khác, nếu hai luật sinh bất kỳ mà có vế trái giống nhau thì vế phải của chúng phải bắt đầu bằng các kí hiệu kết thúc khác nhau.*

Với một chuỗi bất kỳ trong ngôn ngữ được sinh ra bởi văn phạm loại này có thể được phân tích không quá $|w|$ bước.

Để biết điều này, xét phương pháp PTCPVC và chuỗi $w = a_1a_2...a_n$. Vì có tối đa một luật sinh với S trên vế trái và a_1 bắt đầu bên vế phải, dẫn xuất phải bắt đầu với

$$S \Rightarrow a_1A_1A_2...A_m$$

Tiếp tục, chúng ta thay thế cho biến A_1, nhưng một lần nữa chỉ có tối đa một chọn lựa, chúng ta phải có

$$S \overset{*}{\Rightarrow} a_1a_2B_1B_2..A_2...A_m$$

Chúng ta thấy từ điều này rằng mỗi bước sinh được một kí hiệu kết thúc và vì vậy toàn bộ quá trình phải được hoàn tất không quá $|w|$ bước. Như chúng ta sẽ thấy trong phần kế tiếp, nhiều đặc điểm của các ngôn ngữ lập trình tựa Pascal có thể được biểu thị bằng văn phạm–S.

Văn phạm–S có thể mở rộng ra một chút ở x, bằng cách mở rộng $x \in (V \cup T)^*$. Điều này làm cho văn phạm không bị quá gò bó và quá trình PTCP đơn giản hơn một chút, tuy nhiên không làm thay đổi khả năng và tính chất của văn phạm.

2. Tính nhập nhằng trong văn phạm và ngôn ngữ

Khi với một chuỗi w bất kỳ nào đó $\in L(G)$ mà sự PTCPVC sinh ra được nhiều hơn một CDX thì chúng ta sẽ gọi nó là sự ***nhập nhằng*** *(ambiguity)*.

Định nghĩa 5.4

Một VPPNC G được gọi là nhập nhằng nếu tồn tại một w nào đó thuộc L(G) mà có ít nhất hai cây dẫn xuất khác nhau. Một cách phát biểu khác, sự nhập nhằng suy ra tồn tại hai hay nhiều dẫn xuất trái nhất hay phải nhất.

Ví dụ 5.12. Xét văn phạm sau $G = (V, T, E, P)$ với

$$V = \{E, I\}$$

$$T = \{a, b, c, +, *, (,)\}$$

và các luật sinh

$$E \rightarrow I$$

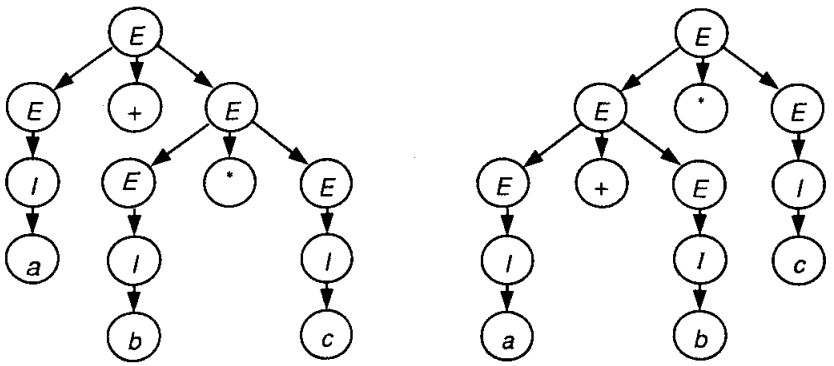
$$E \rightarrow E + E$$

$$E \rightarrow E * E$$

$$E \rightarrow (E)$$

$$I \rightarrow a \mid b \mid c$$

Dễ thấy văn phạm này sinh ra một tập con các biểu thức số học đối với các ngôn ngữ lập trình tựa Pascal. Văn phạm này là nhập nhằng vì với chuỗi $a + b * c$ thì có hai CDX khác nhau như được trình bày trong H.5.4



(a (b

Hình 5.4

Để loại bỏ tính nhập nhằng trong trường hợp này ta làm tương tự như với các biểu thức số học bằng cách qui định độ ưu tiên cho các phép toán. Chẳng hạn cho độ ưu tiên của phép * cao hơn phép +, như vậy H.5.4(a) là sẽ là sự phân tích cú pháp đúng vì nó chỉ ra $b * c$ là một biểu thức con được đánh giá trước khi thực hiện phép cộng. Tuy nhiên việc giải quyết này nằm bên ngoài văn phạm. Tốt hơn là viết lại văn phạm sao cho chỉ có một sự PTCP là có thể.

Ví dụ 5.13. Để viết lại văn phạm trong ví dụ 5.12 chúng ta đưa ra các biến mới, lấy $V = \{E, T, F, I\}$ và thay thế các luật sinh bằng

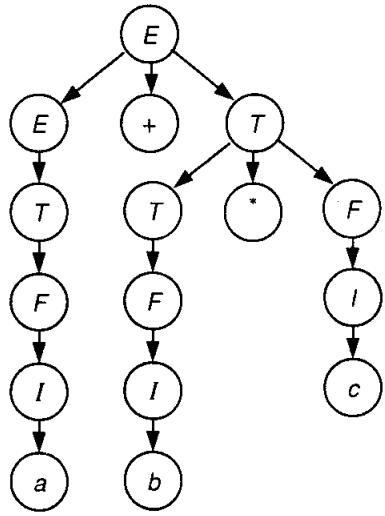
$$E \rightarrow T \mid E + T$$

$$T \rightarrow F \mid T * F$$

$$F \rightarrow I \mid (E)$$

$$I \rightarrow a \mid b \mid c$$

Một CDX của câu $a + b * c$ được trình bày trong H.5.5. Không có một CDX nào khác là có thể cho chuỗi này. Văn phạm này là không nhập nhằng và tương đương với văn phạm trong ví dụ 5.12.



Hình 5.5

Trong các ví dụ trên sự nhập nhằng đến từ văn phạm trong cảm giác rằng nó có thể được loại bỏ bằng cách tìm một văn phạm không nhập nhằng tương đương. Tuy nhiên, trong một vài trường hợp, điều này là không thể bởi sự nhập nhằng ở ngay trong bản thân ngôn ngữ.

Định nghĩa 5.5. *Nếu L là một NNPNC mà đối với nó tồn tại một văn phạm không nhập nhằng, thì L được gọi là không nhập nhằng. Nếu mọi văn phạm sinh ra L mà nhập nhằng, thì ngôn ngữ được gọi là* ***nhập nhằng cố hữu.***

Ví dụ 5.14. Ngôn ngữ $L = \{a^n b^n c^m\} \cup \{ a^n b^m c^m \}$ với n, m không âm là một NNPNC nhập nhằng cố hữu.

L là PNC là dễ chứng minh. Chú ý

$$L = L_1 \cup L_2$$

trong đó: L_1 được sinh bởi văn phạm

$$S_1 \rightarrow X_1 C$$
$$X_1 \rightarrow aX_1 b|\lambda$$
$$C \rightarrow cC|\lambda$$

còn L_2 được sinh bởi văn phạm

$$S_2 \rightarrow AX_2$$
$$X_2 \rightarrow bX_2 c|\lambda$$
$$A \rightarrow aA|\lambda$$

Vì vậy L là được sinh ra bởi sự kết hợp của hai văn phạm này với luật sinh thêm vào là

$$S \rightarrow S_1 | S_2$$

Văn phạm này là nhập nhằng vì chuỗi $a^n b^n c^n$ thuộc cả L_1 lẫn L_2 nên nó có hai dẫn xuất riêng biệt: một cái bắt đầu bằng $S \Rightarrow S_1$ và một cái bắt đầu bằng $S \Rightarrow S_2$. Điều này cũng gợi ý cho chúng ta chứng minh rằng mọi văn phạm cho L đều sẽ nhập nhằng trên chuỗi $a^n b^n c^n$ tương tự như trường hợp trên. Chúng ta có thể lý luận rằng mọi văn phạm sinh ra L, thì tập luật sinh của nó sẽ được chia làm hai phần, một phần dùng để sinh ra L_1, một phần để sinh ra L_2 và hai phần này sẽ có các luật sinh khác nhau. Vì vậy đối với những chuỗi thuộc cả L_1 lẫn L_2 như $a^n b^n c^n$ sẽ có hai dẫn xuất đi theo hai hướng khác nhau. Một chứng minh chặt chẽ đã được thực hiện trong tài liệu của Harrison 1978. Ở đây nó được để lại như bài tập cho các bạn.

5.3. VĂN PHẠM PHI NGỮ CẢNH VÀ NGÔN NGỮ LẬP TRÌNH

Một trong những sử dụng quan trọng nhất của lý thuyết ngôn ngữ hình thức là trong định nghĩa của các ngôn ngữ lập trình và trong việc xây dựng các trình thông dịch và biên dịch cho chúng.

Vấn đề cơ bản ở đây là định nghĩa một ngôn ngữ lập trình một cách chính xác và sử dụng định nghĩa này như là một điểm khởi đầu cho việc viết các chương trình dịch hiệu quả và tin cậy. Cả NNCQ lẫn PNC đều quan trọng trong việc thực hiện điều này. Như chúng ta đã thấy, NNCQ là được sử dụng trong việc nhận biết một mẫu đơn giản nào đó xảy ra trong ngôn ngữ lập trình, nhưng như chúng ta lý luận trong phần giới thiệu của chương này, chúng ta cần NNPNC để mô hình các khía cạnh phức tạp hơn.

Như với hầu hết các ngôn ngữ khác, chúng ta có thể định nghĩa một ngôn ngữ lập trình bằng một văn phạm. Theo truyền thống trong việc viết một ngôn ngữ lập trình chúng ta sử dụng dạng ký pháp *Backus–Naur* hay viết tắt là BNF. Dạng này cơ bản giống như các kí hiệu mà chúng ta đã sử dụng ở đây, tuy nhiên diện mạo có hơi khác. Trong BNF, các biến được đóng bởi các cặp dấu ngoặc góc (<>). Các kí hiệu kết thúc được viết mà không có đánh dấu gì đặc biệt. BNF cũng sử dụng các kí hiệu phụ như |, giống như cách chúng ta đã thực hiện. Vì vậy, văn phạm trong ví dụ 5.13 phải xuất hiện trong BNF như sau:

$$\langle expression \rangle \ ::= \langle term \rangle \mid \langle expression \rangle + \langle term \rangle,$$

$$\langle term \rangle \ ::= \langle factor \rangle \mid \langle term \rangle \ ^{*} \ \langle factor \rangle,$$

và ... Kí hiệu + và * là những kí hiệu kết thúc. Kí hiệu | được sử dụng như một kí hiệu thay thế như trong kí hiệu của chúng ta, còn ::= được sử dụng thay thế cho →. BNF có khuynh hướng sử dụng các danh hiệu biến tường minh hơn để làm cho ý nghĩa của luật sinh trở nên rõ ràng.

Nhiều phần trong ngôn ngữ lập trình tựa Pascal nhạy cảm đối với định nghĩa bằng các dạng giới hạn của VPPNC. Chẳng hạn, một phát biểu **if–then–else** của Pascal có thể được định nghĩa như:

<if_statement> ::= *if* < *expression* > *<then_clause> <else_clause>*

Ở đây từ khóa **if** là kí hiệu kết thúc. Tất cả các thành phần khác là các biến mà còn phải được định nghĩa. Chúng ta thấy rằng cái này giống như một luật sinh của văn phạm–*S*. Biến < *if_statement* > trên vế trái là luôn luôn gắn với kí hiệu kết thúc **if** trên vế phải. Với lý do này một phát biểu như vậy là được phân tích cú pháp một cách dễ dàng và hiệu quả. Chúng ta thấy ở đây một lý do tại sao chúng ta sử dụng từ khóa trong các ngôn ngữ lập trình. Từ khóa không chỉ cung cấp một cấu trúc trực quan có thể giúp người đọc chương trình mà còn làm cho công việc viết trình biên dịch dễ dàng hơn rất nhiều.

Không may mắn thay, không phải tất cả các đặc điểm của một ngôn ngữ lập trình điển hình đều có thể được biểu thị bằng một văn phạm–*S*. Luật cho < *expression* > ở trên là không ở trong loại này, vì vậy việc phân tích cú pháp trở nên ít rõ ràng hơn. Thế thì một câu hỏi được đặt ra là những luật văn phạm gì chúng ta có thể cho phép mà vẫn phân tích cú pháp hiệu quả. Người ta đã tìm thấy các văn phạm như vậy đó là các văn phạm *LL* và *LR*, những văn phạm mà có khả năng biểu thị các đặc điểm ít rõ ràng của một ngôn ngữ lập trình, và còn cho phép chúng ta phân tích trong thời gian tuyến tính, chúng hiện đang được sử dụng rộng rãi trong các trình biên dịch. Đây không phải là vấn đề đơn giản, và nhiều cái trong đó vượt ra ngoài phạm vi thảo luận của chúng ta. Chúng ta sẽ đề cập ngắn gọn những chủ đề này trong chương 6, nhưng đối với mục đích của chúng ta hiện tại nó là đủ để hiểu rõ rằng những văn phạm như vậy là tồn tại và đã được nghiên cứu một cách rộng rãi.

Liên quan đến vấn đề này, vấn đề về tính nhập nhằng chiếm một ý nghĩa quan trọng. Việc mô tả một ngôn ngữ lập trình phải không được nhập nhằng, nếu không một chương trình có thể tạo ra các kết quả rất khác nhau khi được xử lý bởi các trình biên dịch khác nhau hay chạy trên những hệ thống khác nhau. Như ví dụ 5.12 trình bày, một sự tiếp cận khờ khạo có thể dễ dàng tạo ra sự nhập nhằng trong văn phạm. Để tránh những lỗi như thế chúng ta phải có khả năng nhận biết và loại bỏ

tính nhập nhằng. Một câu hỏi liên quan là, một ngôn ngữ có là nhập nhằng cố hữu hay không. Những gì mà chúng ta cần cho mục đích này là các giải thuật để dò tìm và loại bỏ tính nhập nhằng trong VPPNC và để quyết định một NNPNC có nhập nhằng cố hữu hay không. Không may mắn thay, đây là những công việc vô cùng khó, không thể làm được trong trường hợp tổng quát nhất, như chúng ta sẽ thấy sau. Những khía cạnh đó của một ngôn ngữ lập trình, khía cạnh mà có thể được mô hình hóa bởi một VPPNC, thì thường được xem như là khía cạnh ***cú pháp*** (*syntax*) của nó. Tuy nhiên, thông thường có vấn đề rằng: không phải tất cả các chương trình mà đúng cú pháp theo ngữ cảnh đó lại là những chương trình chấp nhận được. Chẳng hạn đối với Pascal, định nghĩa BNF thông thường cho phép các cấu trúc sau:

var x, y : real;
x, z : integer

hay

var x : integer;
x := 3.2.

Tuy nhiên không có cái nào trong hai cấu trúc này được chấp nhận đối với một trình biên dịch Pascal, vì chúng vi phạm các ràng buộc khác, chẳng hạn như "một biến nguyên không thể được gán một giá trị thực." Những qui tắc loại này là thuộc về ***ngữ nghĩa*** (*semantics*) của ngôn ngữ lập trình, vì nó phải làm theo cách chúng ta diễn dịch ý nghĩa của một cấu trúc cụ thể.

Ngữ nghĩa của ngôn ngữ lập trình là một vấn đề phức tạp. Không đẹp đẽ và chính xác như các VPPNC tồn tại để mô tả các ngôn ngữ lập trình, và vì vậy một vài đặc điểm ngữ nghĩa có thể được định nghĩa một cách nghèo nàn hay mơ hồ. Nó là một mối quan tâm cả trong các ngôn ngữ lập trình lẫn trong lý thuyết ngôn ngữ hình thức để tìm một phương pháp hiệu quả cho việc định nghĩa ngữ nghĩa của ngôn ngữ lập trình. Một vài phương pháp đã được đề nghị, nhưng không có phương pháp nào trong chúng là được chấp nhận rộng rãi cũng như thành công trong việc định nghĩa ngữ nghĩa như NNPNC đã làm được đối với cú pháp.