PGS.TS. PHAN HUY KHÁNH

Lập trình Lôgích trong Prolog

NHÀ XUẤT BẢN ĐẠI HỌC QUỐC GIA HÀ NỘI
2004

PGS.TS. PHAN HUY KHÁNH

Lập trình Lôgích trong Prolog

Prolog là ngôn ngữ lập trình lôgich (Prolog = PROgramming in LOGic) do GS. A. Colmerauer đưa ra lần đầu tiên năm 1972 tại trường Đại học Marseille, nước Pháp. Đến năm 1980, Prolog nhanh chóng được áp dụng rộng rãi, được người Nhật chọn làm ngôn ngữ phát triển máy tính thế hệ 5. Prolog đã được cài đặt trên hầu hết các dòng máy tính Unix/Linux, Macintosh, Windows.

Prolog còn được gọi là ngôn ngữ lập trình ký hiệu (symbolic programming) tương tự lập trình hàm (functional programming), hay lập trình phi số (non-numerical programming). Nguyên lý lập trình lôgich dựa trên phép suy diễn lôgích, liên quan đến những khái niệm toán học như phép hợp nhất Herbrand, hợp giải Robinson, lôgich Horn, lôgich vị từ bậc một (first order predicate logic), v.v...

Prolog rất thích hợp để giải quyết những bài toán liên quan đến các đối tượng và mối quan hệ giữa chúng. Prolog được ứng dụng chủ yếu trong lĩnh vực trí tuệ nhân tạo (Artificial Intelligence) như công nghệ xử lý tri thức, hệ chuyên gia, máy học, xử lý ngôn ngữ, trò chơi, v.v...

Nội dung cuốn sách tập trung trình bày cơ sở lý thuyết và những kỹ thuật lập trình cơ bản trong Prolog, rất thích hợp cho sinh viên các ngành tin học và những bạn đọc muốn tìm hiểu về kỹ thuật lập trình ứng dụng trong lĩnh vực trí tuệ nhân tạo.

VỀ TÁC GIẢ :

Tốt nghiệp ngành Toán Máy tính năm 1979 tại trường Đại học Bách khoa Hà Nội. Từ 1979 đến nay giảng dạy tại khoa Công nghệ Thông tin, trường Đại học Bách khoa, Đại học Đà Nẵng. Bảo vệ tiến sĩ năm 1991 tại Pháp. Giữ chức chủ nhiệm khoa
Công nghệ Thông tin 1995-2000.

Hướng nghiên cứu chính : xử lý ngôn ngữ, xử lý đa ngữ, lý thuyết tính toán.

E-mail: khanhph@vnn.vn

LỜI NÓI ĐẦU

Cuốn sách này nhằm cung cấp cơ sở lý thuyết và những phương pháp lập trình cơ bản nhất của môn học «Lập trình lôgich» (Programming in Logic). Người đọc sẽ được làm quen với một số kỹ thuật lập trình lôgich được ứng dụng tương đối phổ biến và chủ yếu trong lĩnh vực trí tuệ nhân tạo (Artificial Intelligence) như công nghệ xử lý tri thức, máy học, hệ chuyên gia, xử lý ngôn ngữ tự nhiên, trò chơi, v.v...

Cuốn sách gồm năm chương, trong mỗi chương, tác giả đều cố gắng đưa vào nhiều ví dụ minh họa. Nội dung các chương như sau :

- *Chương 1 giới thiệu ngôn ngữ lập trình Prolog dựa trên lôgich Horn (Horn logic). Người đọc được làm quen với các kiểu dữ liệu của Prolog, khái niệm luật, sự kiện và viết được các chương trình Prolog đơn giản.*
- *Chương 2 trình bày các mức nghĩa khác nhau của một chương trình Prolog : nghĩa lôgich, nghĩa khai báo và nghĩa thủ tục, cách Prolog trả lời các câu hỏi, cách Prolog làm thoả mãn các đích.*
- *Chương 3 trình bày các phép toán số học, phép so sánh các đối tượng và định nghĩa các hàm sử dụng phép đệ quy trong Prolog.*
- *Chương 4 trình bày cấu trúc danh sách và các phép xử lý cơ bản trên danh sách của Prolog.*
- *Chương 5 trình bày kỹ thuật lập trình nâng cao với Prolog.*
- *Phần phụ lục giới thiệu ngôn ngữ lập trình SWI-Prolog, hướng dẫn cách cài đặt sử dụng phần mềm này và một số chương trình ví dụ tiêu biểu viết trong SWI Prolog đã chạy có kết quả.*

Cuốn sách này dùng làm giáo trình cho sinh viên ngành Tin học và những bạn đọc muốn tìm hiểu thêm về kỹ thuật lập trình cho lĩnh vực trí tuệ nhân tạo.

Trong quá trình biên soạn, tác giả đã nhận được từ các bạn đồng nghiệp nhiều đóng góp bổ ích về mặt chuyên môn, những động viên khích lệ về mặt tinh thần, sự giúp đỡ về biên tập để cuốn sách được ra đời. Tác giả xin được bày tỏ lòng biết ơn sâu sắc. Tác giả cũng chân thành cảm ơn mọi ý kiến phê bình đóng góp của bạn đọc gần xa về nội dung của cuốn sách này.

Đà Nẵng, ngày 27/05/2004
Tác giả.

MỤC LỤC

CHƯƠNG 1

Mở đầu về ngôn ngữ Prolog

« *A program is a theory (in some logic) and computation is deduction from the theory* »
J. A. Robinson

« *Program = data structure + algorithm* »
N. Wirth

« *Algorithm = logic + control* »
R. Kowalski

I. Giới thiệu ngôn ngữ Prolog

I.1. Prolog là ngôn ngữ lập trình lôgich

Prolog là ngôn ngữ được sử dụng phổ biến nhất trong dòng các ngôn ngữ lập trình lôgich (Prolog có nghĩa là PROgramming in LOGic). Ngôn ngữ Prolog do giáo sư người Pháp Alain Colmerauer và nhóm nghiên cứu của ông đề xuất lần đầu tiên tại trường Đại học Marseille đầu những năm 1970. Đến năm 1980, Prolog nhanh chóng được áp dụng rộng rãi ở châu Âu, được người Nhật chọn làm ngôn ngữ phát triển dòng máy tính thế hệ 5. Prolog đã được cài đặt trên các máy vi tính Apple II, IBM-PC, Macintosh.

Prolog còn được gọi là ngôn ngữ *lập trình ký hiệu* (symbolic programming) tương tự các ngôn ngữ *lập trình hàm* (functional programming), hay *lập trình phi số* (non-numerical programming). Prolog rất thích hợp để giải quyết các bài toán liên quan đến các *đối tượng* (object) và mối *quan hệ* (relation) giữa chúng.

Prolog được sử dụng phổ biến trong lĩnh vực trí tuệ nhân tạo. Nguyên lý lập trình lôgich dựa trên các mệnh đề Horn (Horn logíc). Một mệnh đề Horn biểu diễn một sự kiện hay một sự việc nào đó là đúng hoặc không đúng, xảy ra hoặc không xảy ra (có hoặc không có, v.v...).

Ví dụ I.1 : Sau đây là một số mệnh đề Horn :

1. ***Nếu*** *một người già mà (**và**) khôn ngoan **thì** người đó hạnh phúc.*
2. *Jim là người hạnh phúc.*
3. ***Nếu*** *X là cha mẹ của Y **và** Y là cha mẹ của Z **thì** X là ông của Z.*
4. *Tom là ông của Sue.*

5. Tất cả mọi người đều chết (hoặc ***Nếu*** *ai là người* ***thì*** *ai đó phải chết*).

6. Socrat là người.

Trong các mệnh đề Horn ở trên, các mệnh đề 1, 3, 5 được gọi là các *luật* (rule), các mệnh đề còn lại được gọi là các *sự kiện* (fact). Một chương trình lôgich có thể được xem như là một cơ sở dữ liệu gồm các mệnh đề Horn, hoặc dạng luật, hoặc dạng sự kiện, chẳng hạn như tất cả các sự kiện và luật từ 1 đến 6 ở trên. Người sử dụng (NSD) gọi chạy một chương trình lôgich bằng cách đặt *câu hỏi* (query/ question) truy vấn trên cơ sở dữ liệu này, chẳng hạn câu hỏi :

Socrat có chết không ?

(tương đương khẳng định *Socrat chết* đúng hay sai ?)

Một hệ thống lôgich sẽ thực hiện chương trình theo cách «suy luận»-tìm kiếm dựa trên vốn «hiểu biết» đã có là chương trình - cơ sở dữ liệu, để minh chứng câu hỏi là một khẳng định, là *đúng* (Yes) hoặc *sai* (No). Với câu hỏi trên, hệ thống tìm kiếm trong cơ sở dữ liệu khẳng định *Socrat chết* và «tìm thấy» luật 5 thoả mãn (vế ***thì***). Vận dụng luật 5, hệ thống nhận được *Socrat là người* (vế ***nếu***) chính là sự kiện 5. Từ đó, câu trả lời sẽ là :

Yes

có nghĩa sự kiện *Socrat chết* là đúng.

I.2. Cú pháp Prolog

I.2.1. Các thuật ngữ

Một chương trình Prolog là một cơ sở dữ liệu gồm các *mệnh đề* (clause). Mỗi mệnh đề được xây dựng từ các *vị từ* (predicat). Một vị từ là một phát biểu nào đó về các đối tượng có giá trị chân *đúng* (true) hoặc *sai* (fail). Một vị từ có thể có các đối là các *nguyên lôgich* (logic atom).

Mỗi nguyên tử (nói gọn) biểu diễn một quan hệ giữa các *hạng* (term). Như vậy, hạng và quan hệ giữa các hạng tạo thành mệnh đề.

Hạng được xem là những đối tượng "dữ liệu" trong một trình Prolog. Hạng có thể là *hạng sơ cấp* (elementary term) gồm *hằng* (constant), *biến* (variable) và các *hạng phức hợp* (compound term).

Các hạng phức hợp biểu diễn các đối tượng phức tạp của bài toán cần giải quyết thuộc lĩnh vực đang xét. Hạng phức hợp là một *hàm tử* (functor) có chứa các *đối* (argument), có dạng

Tên_hàm_tử(Đối_1, ..., Đối_n)

Tên hàm tử là một chuỗi chữ cái và/hoặc chũ số được bắt đầu bởi một chữ cái thường. Các đối có thể là biến, hạng sơ cấp, hoặc hạng phức hợp. Trong Prolog,

hàm tử đặc biệt "." (dấu chấm) biểu diễn cấu trúc *danh sách* (list). Kiểu dữ liệu hàm tử tương tự kiểu bản ghi (record) và *danh sách* (list) tương tự kiểu mảng (array) trong các ngôn ngữ lập trình mệnh lệnh (C, Pascal...).

Ví dụ I.2 :

```
f(5, a, b).
student(robert, 1975, info, 2,
                  address(6, 'mal juin', 'Caen')).
[a, b, c]
```

Mệnh đề có thể là một *sự kiện*, một *luật* (hay *quy tắc*), hay một *câu hỏi*.

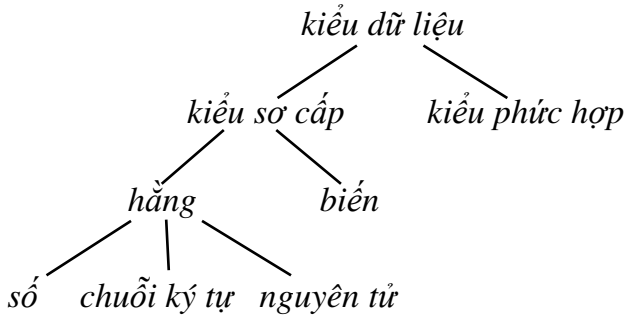
Prolog quy ước viết sau mỗi mệnh đề một dấu chấm để kết thúc như sau :

- Sự kiện : **< ... >.**
  (tương ứng với luật **< ... > :- true.**)
- Luật : **< ... > :- < ... >.**
- Câu hỏi **?- < ... >.**
  (ở chế độ tương tác có dấu nhắc lệnh)

I.2.2. Các kiểu dữ liệu Prolog

Hình 1.1. biểu diễn một sự phân lớp các kiểu dữ liệu trong Prolog gồm kiểu dữ liệu sơ cấp và kiểu dữ liệu có cấu trúc. Sự phân lớp này nhận biết kiểu của một đối tượng nhờ bề ngoài cú pháp.

Cú pháp của Prolog quy định mỗi kiểu đối tượng có một dạng khác nhau. Prolog không cần cung cấp một thông tin nào khác để nhận biết kiểu của một đối tượng. Trong Prolog, NSD không cần khai báo kiểu dữ liệu.



Hình I.1. Các kiểu dữ liệu trong Prolog

Các kiểu dữ liệu Prolog được xây dựng từ các ký tự ASCII :

- Các chữ cái in hoa A, B, ..., Z và chữ cái in thường a, b, ..., z.
- Các chữ số 0, 1, ..., 9.

- Các ký tự đặc biệt, chẳng hạn + - * / < > = : . & _ ~.

I.2.3. Chú thích

Trong một chương trình Prolog, c*hú thích* (comment) được đặt giữa hai cặp ký hiệu **/*** và ***/** (tương tự ngôn ngữ C). Ví dụ :

```
/************************/
/*** Đây là một chú thích ***/
/************************/
```

Trong trường hợp muốn đặt một chú thích ngắn sau mỗi phần khai báo Prolog cho đến hết dòng, có thể đặt trước một ký hiệu **%**.

Ví dụ :

```
%%%%%%%%%%%%%%%%%%%%
% Đây cũng là một chú thích
%%%%%%%%%%%%%%%%%%%%
```

Prolog sẽ bỏ qua tất cả các phần chú thích trong thủ tục.

II. Các kiểu dữ liệu sơ cấp của Prolog

II.1. Các kiểu hằng (trực kiện)

II.1.1. Kiểu hằng số

Prolog sử dụng cả số nguyên và số thực. Cú pháp của các số nguyên và số thực rất đơn giản, chẳng hạn như các ví dụ sau :

```
1       1515     0      -97
3.14    -0.0035  100.2
```

Tuỳ theo phiên bản cài đặt, Prolog có thể xử lý các miền số nguyên và miền số thực khác nhau. Ví dụ trong phiên bản Turbo Prolog, miền số nguyên cho phép từ `-32768` đến `32767`, miền số thực cho phép từ **±**`1e-307` đến **±**`1e+308`. Các số thực rất khi được sử dụng trong Prolog. Lý do chủ yếu ở chỗ Prolog là ngôn ngữ lập trình ký hiệu, phi số.

Các số nguyên thường chỉ được sử dụng khi cần đếm số lượng các phần tử hiện diện trong một danh sách Prolog dạng $[a_1, a_2, \ldots, a_n]$.

II.1.2. Kiểu hằng lôgich

Prolog sử dụng hai hằng lôgich có giá trị là `true` và `fail`. Thông thường các hằng lôgich không được dùng như tham số mà được dùng như các mệnh đề. Hằng `fail` thường được dùng để tạo sinh lời giải bài toán.

II.1.3. Kiểu hằng chuỗi ký tự

Các hằng là chuỗi (string) các ký tự được đặt giữa hai dấu nháy kép.

`"Toto \#\{@ tata"`	chuỗi có tuỳ ý ký tự
`""`	chuỗi rỗng (empty string)
`"\""`	chuỗi chỉ có một dấu nháy kép.

II.1.4. Kiểu hằng nguyên tử

Các hằng nguyên tử Prolog là chuỗi ký tự ở một trong ba dạng như sau :

(1) Chuỗi gồm chữ cái, chữ số và ký tự _ luôn luôn được *bắt đầu bằng một chữ cái in thường*.

```
newyork                    a_
nil                        x__y
x25                        tom_cruise
```

(2) Chuỗi các ký tự đặc biệt :

```
<--->                      .:.
======>                    ::==
...
```

(3) chuỗi đặt giữa hai dấu nháy đơn (quote) *được bắt đầu bằng chữ in hoa*, dùng phân biệt với các tên biến :

```
'Jerry'                    'Tom SMITH'
```

II.2. Biến

Tên biến là một chuỗi ký tự gồm chữ cái, chữ số, *bắt đầu bởi chữ hoa hoặc dấu gạch dưới dòng :*

```
X, Y, A
Result, List_of_members
_x23, _X, _, ...
```

III. Sự kiện và luật trong Prolog

III.1. Xây dựng sự kiện

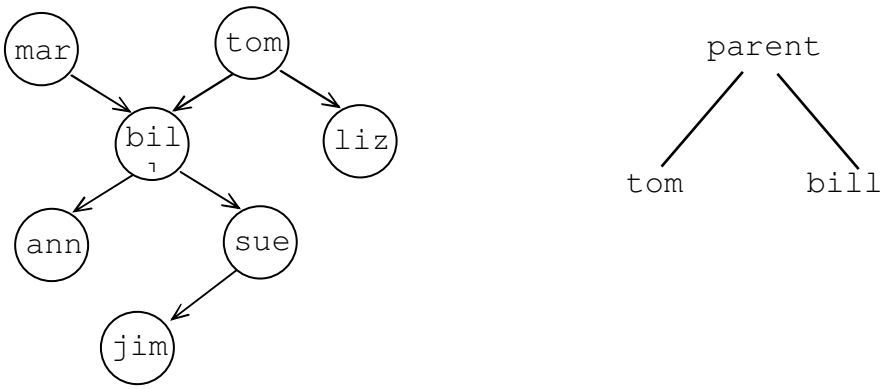
Ví dụ III.1 : Quan hệ gia đình

Để xây dựng các sự kiện trong một chương trình Prolog, ta lấy một ví dụ về. Ta xây dựng một cây gia hệ như Hình III.1.

Trong cây gia hệ (a), các nút chỉ người, còn các mũi tên chỉ quan hệ *cha mẹ của* (parent of). Sự kiện *Tom là cha mẹ của Bill* được viết thành một vị từ Prolog như sau (chú ý mệnh đề được kết thúc bởi một dấu chấm) :

```
parent(tom, bill).      % Chú ý không có dấu cách trước dấu mở ngoặc
```

Ở đây, vị từ `parent` có hai đối là `tom` và `bill`. Người ta có thể biểu diễn vị từ này bởi một cây như trong Hình III.1 (b) : nút gốc là tên của vị từ, còn các nút lá là các đối.



(a) (b)

Hình III.1.Cây gia hệ.

Từ cây gia hệ trên đây, có thể tiếp tục viết các vị từ khác để nhận được một chương trình Prolog gồm 6 vị từ như sau :

```
parent(mary, bill).
parent(tom, bill).
parent(tom, liz).
parent(bill, ann).
parent(bill, sue).
parent(sue, jim).
```

Sau khi hệ thống Prolog nhận được chương trình này, thực chất là một cơ sở dữ liệu, người ta có thể đặt ra các câu hỏi liên quan đến quan hệ `parent`. Ví dụ

câu hỏi *Bill có phải là cha mẹ của Sue* được gõ vào trong hệ thống đối thoại Prolog (dấu nhắc **?-_**) như sau :

```
?- parent(bill, sue).
```

Sau khi tìm thấy sự kiện này trong chương trình, Prolog trả lời :

```
Yes
```

Ta tiếp tục đặt câu hỏi khác :

```
?- parent(liz, sue).
No
```

Bởi vì Prolog không tìm thấy sự kiện Liz là người mẹ của Sue trong chương trình. Tương tự, Prolog trả lời `No` cho sự kiện :

```
?- parent(tom, ben).
```

Vì tên `ben` chưa được đưa vào trong chương trình. Ta có thể tiếp tục đặt ra các câu hỏi thú vị khác. Chẳng hạn, ai là cha (hay mẹ) của Liz ?

```
?- parent(X, liz).
```

Lần này, Prolog không trả lời `Yes` hoặc `No`, mà đưa ra một giá trị của X làm thoả mãn câu hỏi trên đây :

```
X = tom
```

Để biết được ai là con của Bill, ta chỉ cần viết :

```
?- parent(bill, X).
```

Với câu hỏi này, Prolog sẽ có hai câu trả lời, đầu tiên là :

```
X = ann ->;
```

Để biết được câu trả lời tiếp theo, trong hầu hết các cài đặt của Prolog, NSD phải gõ vào một dấu chấm phẩy (`;`) sau `->` (Arity Prolog) :

```
X = sue
```

Nếu đã hết phương án trả lời mà vẫn tiếp tục yêu cầu (`;`), Prolog trả lời `No`.

NSD có thể đặt các câu hỏi tổng quát hơn, chẳng hạn : *ai là cha mẹ của ai* ? Nói cách khác, cần tìm X và Y sao cho X là cha mẹ của Y. Ta viết như sau :

```
?- parent(X, Y).
```

Sau khi hiển thị câu trả lời đầu tiên, Prolog sẽ lần lượt tìm kiếm những cặp cha mẹ – con thoả mãn và lần lượt hiển thị kết quả nếu chừng nào NSD còn yêu cầu cho đến khi không còn kết quả lời giải nào nữa (kết thúc bởi `Yes`) :

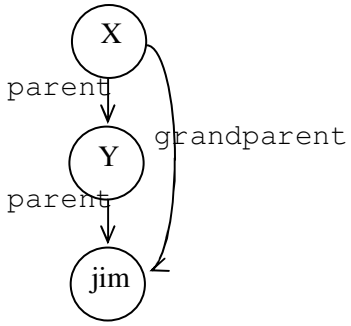
```
X = mary
Y = bill ->;
X = tom
Y = bill ->;
```

```
X = tom
Y = liz ->;
X = bill
Y = ann ->;
X = bill
Y = sue ->;
X = sue
Y = jim
Yes
```

Tuỳ theo cài đặt Prolog, NSD có thể gõ vào một dấu chấm (.) hoặc Enter để chấm dứt giữa chừng luồng trả lời.

Ta có thể tiếp tục đưa ra những câu hỏi phức tạp hơn khác, chẳng hạn *ai là ông (bà) của Jim ?* Thực tế, quan hệ ông – bà (`grandparent`) chưa được định nghĩa, cần phải phân tách câu hỏi này thành hai phần sơ cấp hơn :

1. Ai là cha (mẹ) của Jim ? Giả sử có tên là Y.
2. Ai là cha (mẹ) của Y ? Giả sử có tên là X.



Hình III.2. Quan hệ ông bà được hợp thành từ hai quan hệ cha mẹ.

Lúc này, có thể viết trong Prolog như sau :

```
?- parent(Y, jim), parent(X, Y).
```

Prolog trả lời :

```
Y = sue
X = bill
Yes
```

Câu hỏi trên đây tương ứng với câu hỏi : tìm X và Y thoả mãn :

```
parent(Y, jim)
```

và

```
parent(X, Y).
```

Nếu thay đổi thứ tự hai thành phần câu hỏi, thì nghĩa lôgich vẫn không thay đổi và Prolog trả lời cùng kết quả (có thể thay đổi về thứ tự), nghĩa là ta có thể đặt câu hỏi như sau :

```
?- parent(X, Y), parent(Y, jim).
X = bill
Y = đường dẫn
Yes
```

Bây giờ ta đặt câu hỏi *ai là cháu của Tom* ?

```
?- parent(tom, X), parent(X, Y).
X = bill
Y = ann->;
X = bill
Y = sue ->;
No
```

Một câu hỏi khác có thể như sau : *Ann và Sue có cùng người ông không ?* nghĩa là ta diễn đạt thành hai giai đoạn :

1. Tìm X là cha mẹ của Ann.
2. X tìm thấy có cùng là cha mẹ của Sue không ?

Câu hỏi và trả lời trong Prolog như sau :

```
?- parent(X, ann), parent(X, sue).
X = bill
```

Trong Prolog, câu hỏi còn được gọi là *đích* (goal) cần phải được *thoả mãn* (satisfy). Mỗi câu hỏi đặt ra đối với cơ sở dữ liệu có thể tương ứng với một hoặc nhiều đích. Chẳng hạn dãy các đích :

```
parent(X, ann), parent(X, sue).
```

tương ứng với câu hỏi là *phép hội* (conjunction) của 2 mệnh đề :

X là một cha mẹ của Ann, *và*
X là một cha mẹ của Sue.

Nếu câu trả lời là `Yes`, thì có nghĩa đích đã được thoả mãn, hay đã *thành công*. Trong trường hợp ngược lại, câu trả lời là `No`, có nghĩa đích *không được thoả mãn*, hay đã *thất bại*.

Nếu có nhiều câu trả lời cho một câu hỏi, Prolog sẽ đưa ra câu trả lời đầu tiên và chờ yêu cầu của NSD tiếp tục.

III.2. Xây dựng luật

III.2.1. Định nghĩa luật

Từ chương trình gia hệ trên đây, ta có thể dễ dàng bổ sung các thông tin khác, chẳng hạn bổ sung các sự kiện về giới tính (nam, nữ) của những người đã nêu tên trong quan hệ `parent` như sau :

```
woman(mary).
man(tom).
man(bill).
woman(liz).
woman(sue).
woman(ann).
man(jim).
```

Ta đã định nghĩa các quan hệ *đơn* (unary) `woman` và `man` vì chúng chỉ liên quan đến một đối tượng duy nhất. Còn quan hệ `parent` là nhị phân, vì liên quan đến một cặp đối tượng. Như vậy, các quan hệ đơn dùng để thiết lập một thuộc tính của một đối tượng. Mệnh đề :

```
woman(mary).
```

được giải thích : *Mary là nữ*. Tuy nhiên, ta cũng có thể sử dụng quan hệ nhị phân để định nghĩa giới tính :

```
sex(mary, female).
sex(tom, male).
sex(bill, male).
...
```

Bây giờ ta đưa vào một quan hệ mới `child`, đối ngược với `parent` như sau :

```
child(liz, tom).
```

Từ đó, ta định nghĩa luật mới như sau :

```
child(Y, X) :- parent(X, Y).
```

Luật trên được hiểu là :

Với mọi X và Y,
Y là con của X nếu
X là cha (hay mẹ) của Y.

hay

Với mọi X và Y,
nếu X là cha (hay mẹ) của Y thì
Y là con của X.

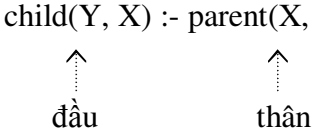
Có sự khác nhau cơ bản giữa sự kiện và luật. Một sự kiện, chẳng hạn :

```
parent(tom, liz).
```

là một điều gì đó luôn đúng, không có điều kiện gì ràng buộc. Trong khi đó, các luật liên quan đến các thuộc tính chỉ được thoả mãn nếu một số điều kiện nào đó được thoả mãn. Mỗi luật bao gồm hai phần:

- phần bên phải (RHS: Right Hand Side) chỉ *điều kiện*, còn được gọi là *thân* (body) của luật, và
- phần bên trái (LH: Left Hand Side S) chỉ *kết luận*, còn được gọi là *đầu* (head) của luật.

Nếu điều kiện `parent(X, Y)` là đúng, thì `child(Y, X)` cũng đúng và là hậu quả lôgich của phép suy luận (inference).



Câu hỏi sau đây giải thích cách Prolog sử dụng các luật : Liz có phải là con của Tom không ?

```
?- child(liz, tom)
```

Thực tế, trong chương trình không có sự kiện nào liên quan đến con, mà ta phải tìm cách áp dụng các luật. Luật trên đây ở dạng tổng quát với các đối tượng X và Y bất kỳ, mà ta lại cần các đối tượng cụ thể `liz` và `tom`.

Ta cần sử dụng *phép thế* (substitution) bằng cách gán giá trị `liz` cho biến Y và `tom` cho X. Người ta nói rằng các biến X và Y đã được *ràng buộc* (bound) :

```
X = tom
```

và

```
Y = liz
```

Lúc này, phần điều kiện có giá trị `parent(tom, liz)` và trở thành *đích con* (sub-goal) để Prolog thay thế cho đích `child(liz, tom)`. Tuy nhiên, đích này thoả mãn và có giá trị `Yes` vì chính là sự kiện đã thiết lập trong chương trình.

Sau đây, ta tiếp tục bổ sung các quan hệ mới. Quan hệ mẹ `mother` được định nghĩa như sau (chú ý dấu phẩy chỉ phép *hội* hay phép *và* lôgich) :

```
mother(X, Y) :- parent(X, Y), woman(X).
```

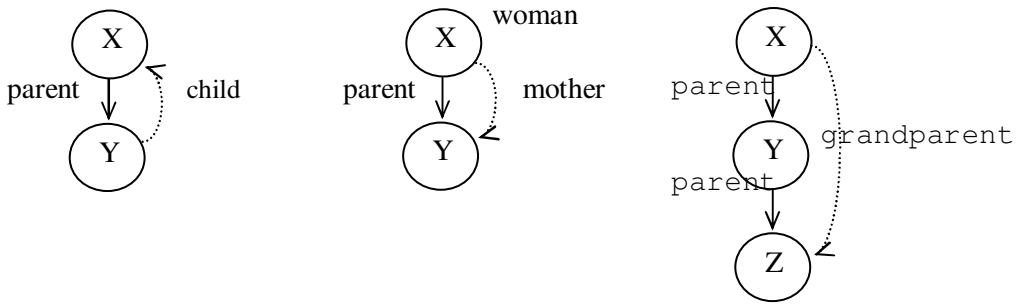
được hiểu là :

Với mọi X và Y, X là mẹ của Y nếu
X là cha (hay mẹ) của Y và X là nữ.

Đồ thị sau đây minh hoạ việc định nghĩa các quan hệ `child`, `mother` và `grandparent` sử dụng một quan hệ khác :

Trong đồ thị, người ta quy ước rằng : các nút tương ứng với các đối tượng (là các đối của một quan hệ). Các cung nối các nút tương ứng với các quan hệ nhị phân, được định hướng từ đối thứ nhất đến đối thứ hai của quan hệ.

Một quan hệ đơn được biểu diễn bởi tên quan hệ tương ứng với nhãn của đối tượng đó. Các quan hệ cần định nghĩa sẽ được biểu diễn bởi các cung có nét đứt. Mỗi đồ thị được giải thích như sau : nếu các quan hệ được chỉ bởi các cung có nét liền được thoả mãn, thì quan hệ biểu diễn bởi cung có nét đứt cũng được thoả mãn.



Hình III.3. Định nghĩa quan hệ con, mẹ và ông bà sử dụng một quan hệ khác.

Như vậy, quan hệ ông–bà `grandparent` được viết như sau :

```
grandparent(X, Z) :- parent(X, Y), parent(Y, Z).
```

Để thuận tiện cho việc đọc chương trình Prolog, ta có thể viết một luật trên nhiều dòng, dòng đầu tiên là phần đầu của luật, các dòng tiếp theo là phần thân của luật, mỗi đích trên một dòng phân biệt. Bây giờ quan hệ `grandparent` được viết lại như sau :

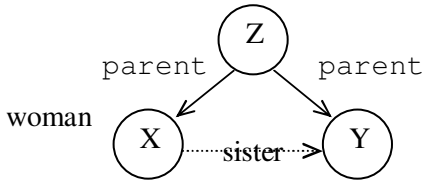
```
grandparent(X, Z) :-
  parent(X, Y),
  parent(Y, Z).
```

Ta tiếp tục định nghĩa quan hệ chị em gái `sister` như sau :

Với mọi X và Y, X là một chị (em) gái của Y nếu
(1) *X và Y có cùng cha (cùng mẹ), và*
(2) *X là nữ*.

```
sister(X, Y) :-
```

```
   parent(Z, X),
   parent(Z, Y),
   woman(X).
```



Hình III.4. Định nghĩa quan hệ chị (em) gái.

Chú ý cách giải thích điều kiện *X và Y có cùng cha mẹ* : một Z nào đó phải là một cha mẹ của X, và cũng Z đó phải là một cha mẹ của Y.

Hay nói một cách khác là : Z1 là một cha mẹ của X, Z2 là một cha mẹ của Y, và Z1 đồng nhất với Z2.

An là nữ, Ann và Sue cùng cha mẹ nên Ann là chị em gái của Sue, ta có :

```
?- sister(ann, sue).
Yes
```

Ta cũng có thể hỏi ai là chị em gái của Sue như sau :

```
?- sister(X, sue).
```

Prolog sẽ lần lượt đưa ra hai câu trả lời :

```
X = ann ->;
X = sue ->.
Yes
```

Vậy thì Sue lại là em gái của chính mình ?! Điều này sai vì ta chưa giải thích rõ trong định nghĩa chị em gái. Nếu chỉ dựa vào định nghĩa trên đây thì câu trả lời của Prolog là hoàn toàn hợp lý. Prolog suy luận rằng X và Y có thể đồng nhất với nhau, mỗi người đàn bà có cùng cha mẹ sẽ là em gái của chính mình. Ta cần sửa lại định nghĩa bằng cách thêm vào điều kiện X và Y khác nhau. Như sẽ thấy sau này, Prolog có nhiều cách để giải quyết, tuy nhiên lúc này ta giả sử rằng quan hệ :

```
different(X, Y)
```

đã được Prolog nhận biết và được thoả mãn nếu và chỉ nếu X và Y không bằng nhau. Định nghĩa chị (em) gái mới như sau :

```
sister(X, Y) :-
   parent(Z, X),
   parent(Z, Y),
   woman(X).
   different(X, Y).
```

Ví dụ III.2 : Ta lấy lại ví dụ cổ điển sử dụng hai tiên đề sau đây :

Tất cả mọi người đều chết.
Socrate là một người.

Ta viết trong Prolog như sau :

```
mortal(X) :- man(X).
man(socrate).
```

Một định lý được suy luận một cách lôgich từ hai tiên đề này là *Socrate phải chết.* Ta đặt các câu hỏi như sau :

```
?- mortal(socrate).
Yes
```

Ví dụ III.3 :

Để chỉ Paul cũng là người, còn Bonzo là con vật, ta viết các sự kiện :

```
man(paul).
animal(bonzo).
```

Con người có thể nói và không phải là loại vật, ta viết luật :

```
speak(X) :- man(X), not(animal(bonzo)).
```

Ta đặt các câu hỏi như sau :

```
?- speak(bonzo).
No
?- speak(paul).
Yes
```

Ví dụ III.4 :

Ta đã xây dựng các sự kiện và các luật có dạng vị từ chứa tham đối, sau đây, ta lấy một ví dụ khác về sự kiện và luật không chứa tham đối :

```
'It is sunny'.
'It is summer'.
'It is hot' :-
   'It is summer', 'It is sunny'.
'It is cold' :-
   'It is winter', 'It is snowing'.
```

Từ chương trình trên, ta có thể đặt câu hỏi :

```
?- 'It is hot'.
Yes
```

Câu trả lời `'It is hot'` là đúng vì đã có các sự kiện `'It is sunny'` và `'It is summer'` trong chương trình. Còn câu hỏi « `?- 'It is cold.'` » có câu trả lời sai.

III.2.2. Định nghĩa luật đệ quy

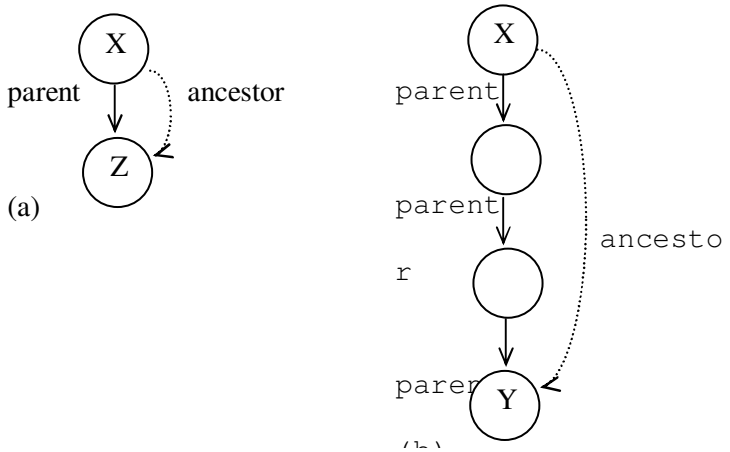
Bây giờ ta tiếp tục thêm một quan hệ mới vào chương trình. Quan hệ này chỉ sử dụng quan hệ `parent`, và chỉ có hai luật. Luật thứ nhất định nghĩa các tổ tiên trực tiếp, luật thứ hai định nghĩa các tổ tiên gián tiếp.

Ta nói rằng X là một tổ tiên gián tiếp của Z nếu tồn tại một liên hệ cha mẹ (ông bà) giữa X và Z :

Trong cây gia hệ ở *Hình III.1*, Tom là tổ tiên trực tiếp của Liz, và tổ tiên gián tiếp của Sue. Ta định nghĩa luật 1 (tổ tiên trực tiếp) như sau :

Với mọi X và Z,
X là một tổ tiên của Z nếu
X là cha mẹ của Z.

```
ancestor(X, Z) :-
  parent(X, Z).
```



Hình III.5. Quan hệ tổ tiên : (a) X là tổ tiên trực tiếp của Z,
(b) X là tổ tiên gián tiếp của Z.

Định nghĩa luật 2 (tổ tiên gián tiếp) phức tạp hơn, trình Prolog trở nên dài dòng hơn, mỗi khi càng mở rộng mức tổ tiên hậu duệ như chỉ ra trong *Hình III.6*.

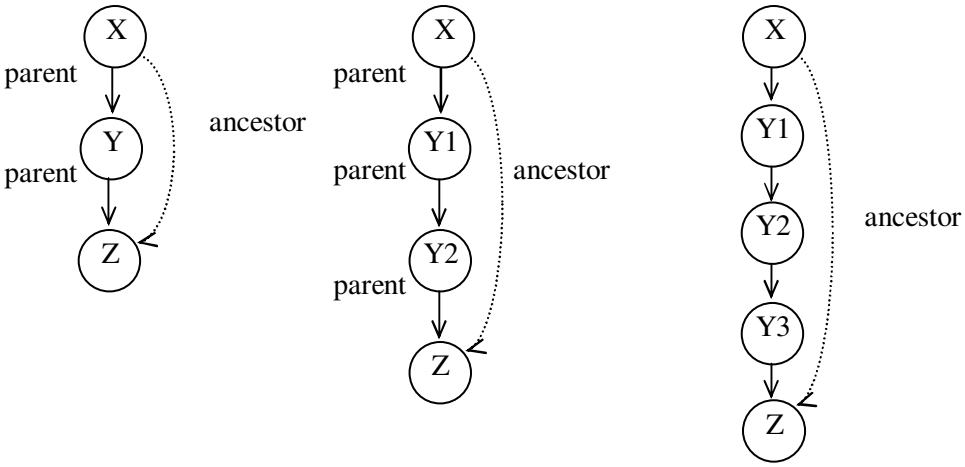
Kể cả luật 1, ta có quan hệ tổ tiên được định nghĩa như sau :

```
ancestor(X, Z) :-          % luật 1 định nghĩa tổ tiên trực tiếp
   parent(X, Z).
ancestor(X, Z) :-          % luật 2 : tổ tiên gián tiếp là ông bà (tam đại)
   parent(X, Y),
   parent(Y, Z).
ancestor(X, Z) :-          % tổ tiên gián tiếp là cố ông cố bà (tứ đại)
   parent(X, Y1),
   parent(Y1, Y2),
   parent(Y2, Z).
```

```
ancestor(X, Z) :-          % ngũ đại đồng đường
   parent(X, Y1),
   parent(Y1, Y2),
   parent(Y2, Y3),
   parent(Y3, Z).
...
```

Tuy nhiên, tồn tại một cách định nghĩa tổ tiên gián tiếp ở mức bất kỳ nhờ phép *đệ quy* (recursive) như sau :

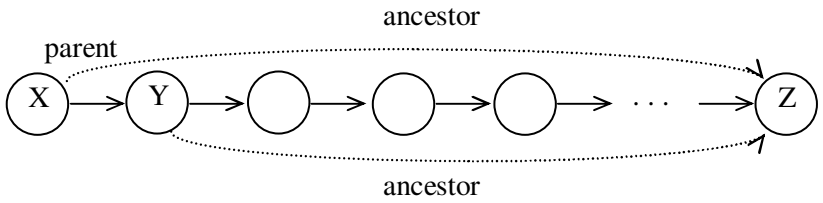
Với mọi X và Z,
X là một tổ tiên của Z nếu
tồn tại Y sao cho
(1) X là cha mẹ của Y và
(2) Y là một tổ tiên của Z.



Hình III.6. Các cặp tổ tiên hậu duệ gián tiếp ở các mức khác nhau.

```
ancestor(X, Z) :-
   parent(X, Z).

ancestor(X, Z) :-
   parent(X, Y),
   ancestor(Y, Z).

?- ancestor(mary, X).

X = jim ->;
X = ann ->;
X = sue ->;
X = bill
Yes
```

Trong Prolog, hầu hết các chương trình phức tạp đều sử dụng đệ quy, đệ quy là một khả năng mạnh của Prolog.



Hình III.7.Dạng đệ quy của quan hệ tổ tiên (được quay ngang cho thuận tiện).

Cho đến lúc này, ta đã định nghĩa nhiều quan hệ (`parent`, `woman`, `man`, `grandparent`, `child`, `sister`, `mother` và `ancestor`). Ta thấy mỗi quan hệ chỉ tương ứng với một mệnh đề, tuy nhiên, quan hệ `ancestor` lại có hai mệnh đề.

Người ta nói rằng những mệnh đề này *liên quan* (concern) đến quan hệ `ancestor`. Trong trường hợp tất cả các mệnh đề đều liên quan đến một quan hệ, người ta nhận được một *thủ tục* (procedure).

III.2.3. Sử dụng biến trong Prolog

Khi tính toán, NSD có thể thay thế một biến trong một mệnh đề bởi một đối tượng khác. Lúc này ta nói biến đã bị *ràng buộc*.

Các biến xuất hiện trong một mệnh đề được gọi là biến tự do. Người ta giả thiết rằng các biến là được lượng tử toàn thể và được đọc là «*với mọi*». Tuy hiên có nhiều cách giải thích khác nhau trong trường hợp các biến chỉ xuất hiện trong phần bên phải của luật. Ví dụ :

```
haveachil(X) :- parent(X, Y).
```

có thể được đọc như sau :

(a) *Với mọi* X và Y,
nếu X là cha (hay mẹ) của Y thì X có một người con.

(b) *Với mọi* X,
X có một người con nếu tồn tại *một* Y sao cho X là cha (hay mẹ) của Y.

Khi một biến chỉ xuất hiện một lần trong một mệnh đề thì không cần đặt tên cho nó. Prolog cho phép sử dụng các *biến nặc danh* (anonymous variable) là các biến có tên chỉ là một dấu gạch dưới dòng _. Ta xét ví dụ sau :

```
have_a_child(X) :- parent(X, Y).
```

Luật trên nêu lên rằng với mọi X, X có một con nếu X là cha của một Y nào đó. Ta thấy đích `have_a_child` không phụ thuộc gì vào tên của con, vì vậy có thể sử dụng biến nặc danh như sau :

```
have_a_child(X) :- parent(X, _).
```

Mỗi vị trí xuất hiện dấu gạch dưới dòng _ trong một mệnh đề tương ứng với một biến nặc danh mới. Ví dụ nếu ta muốn thể hiện tồn tại một người nào đó có con nếu tồn tại hai đối tượng sao cho một đối tượng này là cha của đối tượng kia, thì ta có thể viết :

```
someone_has_a_child :- parent(_, _).
```

Mệnh đề này tương đương với :

```
someone_has_a_child :- parent(X, Y).
```

nhưng hoàn toàn khác với :

```
someone_has_a_child :- parent(X, X).
```

Nếu biến nặc danh xuất hiện trong một câu hỏi, thì Prolog sẽ không hiển thị giá trị của biến này trong kết quả trả về. Nếu ta muốn tìm kiếm những người có con, mà không quan tâm đến tên con là gì, thì chỉ cần viết :

```
?- parent(X, _).
```

hoặc tìm kiếm những người con, mà không quan tâm đến cha mẹ là gì :

```
?- parent(_ , X).
```

Tầm vực từ vựng (lexical scope) của các biến trong một mệnh đề không vượt ra khỏi mệnh đề đó. Có nghĩa là nếu, ví dụ, biến X15 xuất hiện trong hai mệnh đề khác nhau, thì sẽ tương ứng với hai biến phân biệt nhau. Trong cùng một mệnh đề, X15 luôn luôn chỉ biểu diễn một biến. Tuy nhiên đối với các hằng thì tình huống lại khác : một nguyên tử thể hiện một đối tượng trong tất cả các mệnh đề, có nghĩa là trong tất cả chương trình.

IV. Kiểu dữ liệu cấu trúc của Prolog

IV.1. Định nghĩa kiểu cấu trúc của Prolog

Kiểu dữ liệu có cấu trúc, tương tự cấu trúc bản ghi, là đối tượng có nhiều thành phần, mỗi thành phần lại có thể là một cấu trúc. Prolog xem mỗi thành phần như là một đối tượng khi xử lý các cấu trúc. Để tổ hợp các thành phần thành một đối tượng duy nhất, Prolog sử dụng các hàm tử.

Ví dụ IV.1 :

Cấu trúc gồm các thành phần ngày tháng năm tạo ra hàm tử `date`.

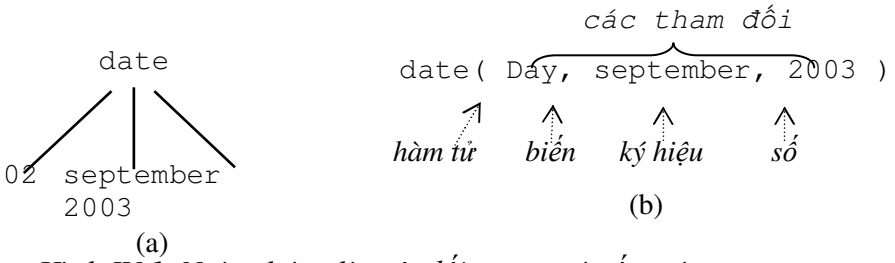
Ngày 2/9/1952 sẽ được viết như sau : `date(2, september, 1952)`

Mọi thành phần trong hàm tử `date` đều là hằng (hai số nguyên và một nguyên tử). Tuy nhiên ta có thể thay thế mỗi thành phần bằng một biến hay một cấu trúc khác. Chẳng hạn ta có thể thay thế thành phần thứ nhất bằng biến `Day` (chú ý tên biến bắt đầu bởi chữ hoa) thể hiện bất kỳ ngày nào của tháng 9 :

```
date(Day, may, 1890)
```

Chú ý rằng `Day` là một biến, có thể được ràng buộc khi xử lý sau đó.

Trong Prolog, về mặt cú pháp, các đối tượng là những hạng. Trong ví dụ trên, `may` và `date(Day, september, 2003)` đều là những hạng.



Hình IV.1. Ngày tháng là một đối tượng có cấu trúc :
(a) biểu diễn dạng cây của cấu trúc ; (b) giải thích cách viết trong Prolog

Mọi đối tượng có cấu trúc đều có thể được biểu diễn hình học dưới dạng *cây* (tree), với hàm tử là gốc, còn các thành phần tham đối là các nhánh của cây. Nếu một trong các thành phần là một cấu trúc, thì thành phần đó tạo thành một cây con của cây ban đầu. Hai hạng là có cùng cấu trúc nếu có cùng cây biểu diễn và có cùng thành phần (pattern of variables). Hàm tử của gốc được gọi là *hàm tử chính* của hạng.

Ví dụ IV.2 :

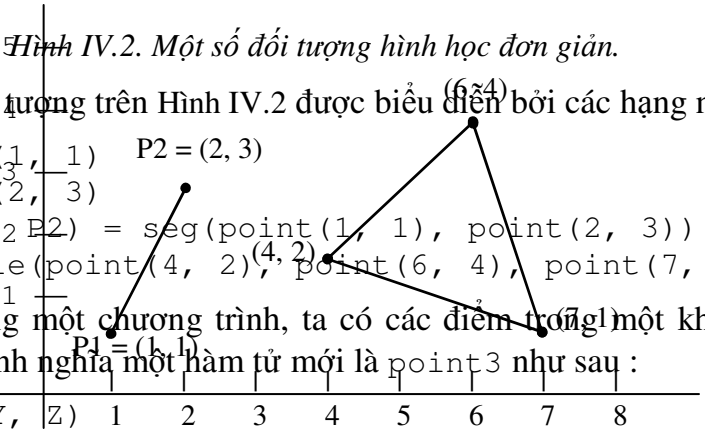
Cấu trúc (đơn giản) của một cuốn sách gồm ba thành phần *tiêu đề* và *tác giả* cũng là các cấu trúc (con), *năm xuất bản* là một biến :

```
book(title(Name), author(Author), Year)
```

Ví dụ IV.3 :

Xây dựng các đối tượng hình học đơn giản trong không gian hai chiều. Mỗi điểm được xác định bởi hai toạ độ, hai điểm tạo thành một đường thẳng, ba điểm tạo thành một tam giác. Ta xây dựng các hàm tử sau đây :

`point`	biểu diễn điểm,
`seg`	biểu diễn một đoạn thẳng (segment),
`triangle`	biểu diễn một tam giác.



Hình IV.2. Một số đối tượng hình học đơn giản.

Từ đó, các đối tượng trên Hình IV.2 được biểu diễn bởi các hạng như sau :

```
P1 = point(1, 1)
P2 = point(2, 3)
S = seg(P1, P2) = seg(point(1, 1), point(2, 3))
T = triangle(point(4, 2), point(6, 4), point(7, 1))
```

Nếu trong cùng một chương trình, ta có các điểm trong một không gian ba chiều, ta có thể định nghĩa một hàm tử mới là `point3` như sau :

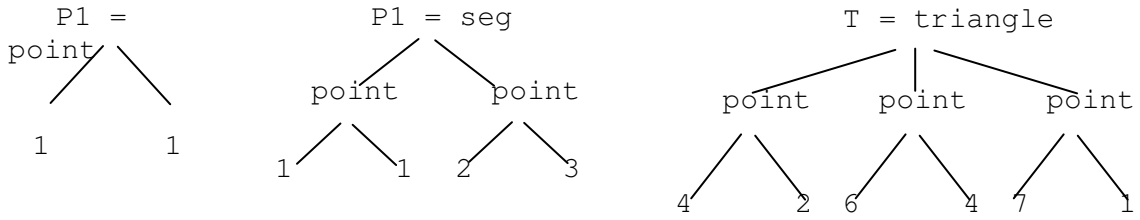
```
point3(X, Y, Z)
```

Prolog cho phép sử dụng cùng tên hai cấu trúc khác nhau. Ví dụ :

`point(X1, Y1)` và `point(X, Y, Z)`

là hai cấu trúc khác nhau.

Trong cùng một chương trình, nếu một tên đóng hai vai trò khác nhau, như trường hợp `point` ở trên, thì Prolog sẽ căn cứ vào số lượng đối số để phân biệt. Cùng một tên này sẽ tương ứng với hai hàm tử, một hàm tử có hai đối số và một hàm tử có ba đối số. Như vậy, một hàm tử được định nghĩa bởi hai yếu tố :

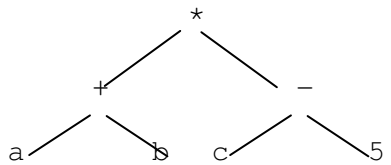


Hình IV.3. Biểu diễn dạng cây của các đối tượng.

(1) Tên hàm tử có cú pháp là cú pháp của các nguyên tử.

(2) Kích thước của hàm tử là số các đối số của nó.

Biểu diễn dạng cây của các đối tượng điểm, đoạn thẳng và tam giác trên đây được cho trong *Hình IV.3.* Như đã trình bày, mọi đối tượng cấu trúc của Prolog đều được biểu diễn dưới dạng cây, xuất hiện trong một chương trình dưới dạng các hạng.



*Hình IV.4. Cấu trúc cây của biểu thức (a + b) * (c – 5)*

Ví dụ biểu thức số học : (*a* + *b*) * (*c* – *5*)
có dạng cây, có thể viết dưới dạng biểu thức tiền tố gồm các hàm tử `*`, `+` và – :

```
*(+(a, b), –(c, 5))
```

IV.2. So sánh và hợp nhất các hạng

Ta vừa xét cách biểu diễn các cấu trúc dữ liệu sử dụng hạng. Bây giờ ta sẽ xét phép toán quan trọng nhất liên quan đến các hạng là phép *so khớp* (matching), thực chất là phép so sánh (comparison operators) trên các hạng và các vị từ.

Trong Prolog, việc so khớp tương ứng với việc *hợp nhất* (unification) được nghiên cứu trong lý thuyết lôgich. Cho hai hạng, người ta nói rằng chúng là hợp nhất được với nhau, nếu :

(1) chúng là giống hệt nhau, hoặc
(2) các biến xuất hiện trong hai hạng có thể được ràng buộc sao cho các hạng của mỗi đối tượng trở nên giống hệt nhau.

Thứ tự chuẩn (standard order) trên các hạng được định nghĩa như sau :

1. Biến < Nguyên tử < Chuỗi < Số < Hạng
2. Biến cũ < Biến mới
3. Nguyên tử được so sánh theo thứ tự ABC (alphabetically).
4. Chuỗi được so sánh theo thứ tự ABC.
5. Số được so sánh theo giá trị (by value). Số nguyên và số thực được xử lý như nhau (treated identically).
6. Các hạng phức hợp (compound terms) được so sánh bậc hay số lượng tham đối (arity) trước, sau đó so sánh tên hàm tử (functor-name) theo thứ tự ABC và cuối cùng so sánh một cách đệ quy (recursively) lần lượt các tham đối từ trái qua phải (leftmost argument first).

Ví dụ hai hạng `date(D, M, 1890)` và `date(D1, May, Y1)` là có thể với nhau nhờ ràng buộc sau :

- `D` được ràng buộc với `D1`
- `M` được ràng buộc với `May`
- `Y1`được ràng buộc với `1890`

Trong Prolog, ta có thể viết :

```
D = D1
M = May
Y1 = 1890
```

Tuy nhiên, ta không thể ràng buộc hai hạng `date(D, M, 1890)` và `date(D1, May, 2000)`, hay `date(X, Y, Z)` và `point(X, Y, Z)`.

Cấu trúc `book(title(Name), author(Author))` được so khớp với :

```
book(title(lord_of_the_rings), author(tolkien))
```

nhờ phép thế :

```
Name = lord_of_the_rings
Author = tolkien
```

Thuật toán hợp nhất Herbrand so khớp hai hạng S và T :

(1) Nếu S và T là các hằng, thì S và T chỉ có thể khớp nhau nếu và chỉ nếu chúng có cùng giá trị (chỉ là một đối tượng).

(2) Nếu S là một biến, T là một đối tượng nào đó bất kỳ, thì S và T khớp nhau, với S được ràng buộc với T. Tương tự, nếu T là một biến, thì T được ràng buộc với S.

(3) Nếu S và T là các cấu trúc, thì S và T khớp nhau nếu và chỉ nếu :

   (a) S và T có cùng một hàm tử chính, và

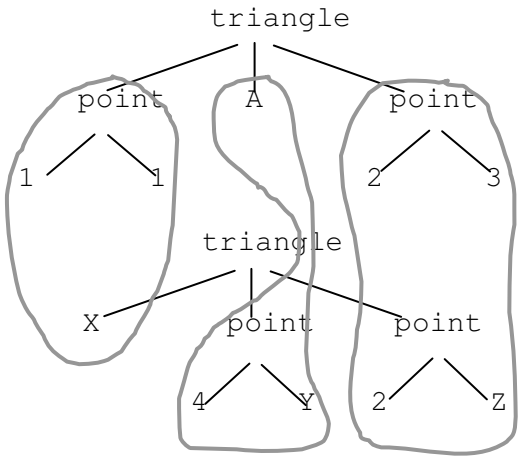
   (b) tất cả các thành phần là khớp nhau từng đôi một.

   Như vậy, sự ràng buộc được xác định bởi sự ràng buộc của các thành phần.

Ta có thể quan sát luật thứ ba ở cách biểu diễn các hạng dưới dạng cây trong Hình IV.5 dưới đây. Quá trình so khớp được bắt đầu từ gốc (hàm tử chính). Nếu hai hàm tử là giống nhau, thì quá trình sẽ được tiếp tục với từng cặp tham đối của chúng. Mọi quá trình so khớp được xem như một dãy các phép tính đơn giản hơn như sau :

```
triangle = triangle
point(1, 1) = X
A = point(4, Y)
point(2, 3) = point(2, Z)
```

Mọi quá trình so khớp là *tích cực* (positive), nếu tất cả các quá trình so khớp bổ trợ là tích cực.



Hình IV.5. Kết quả so khớp :
triangle(point(1, 1), A, point(2, 3)))= triangle(X, point(4, Y), point(2, Z))).

Sự ràng buộc nhận được như sau :

```
X = point(1, 1)
A = point(4, Y)
Z = 3
```

Sau đây là một ví dụ minh hoạ sử dụng kỹ thuật so khớp để nhận biết một đoạn thẳng đã cho là nằm ngang hay thẳng đứng.

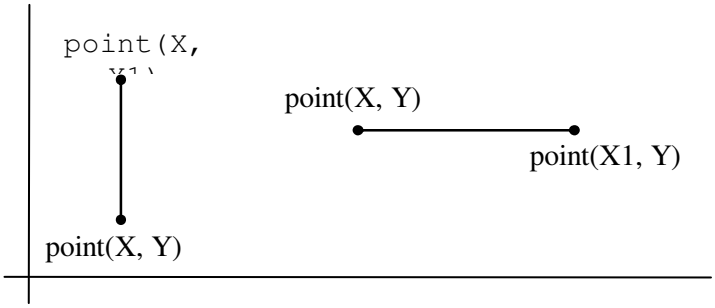
Một đoạn thẳng là thẳng đứng nếu hoành độ (abscissa) của hai mút là bằng nhau, tương tự, là nằm ngang nếu tung độ (ordinate) của hai mút là bằng nhau.

Ta sử dụng quan hệ đơn phân để biểu diễn các tính chất này như sau :

```
vertical(seg(point(X, Y), point(X, Y1))).
horizontal(seg(point(X, Y), point(X1, Y))).
```

Ta có :

```
?- vertical(seg(point(1, 1), point(1, 2))).
Yes
?- vertical(seg(point(1, 1), point(2, Y))).
No
?- horizontal(seg(point(1, 1), point(2, Y))).
Y = 1
Yes
```



Hình IV.6. Minh hoạ các đoạn thẳng nằm ngang và thẳng đứng.

Với câu hỏi thứ nhất, Prolog trả lời `Yes` vì các sự kiện được so khớp đúng. Với câu hỏi thứ hai, vì không có sự kiện nào được so khớp nên Prolog trả lời `No`. Với câu hỏi thứ ba, Prolog cho `Y` giá trị *1* để được so khớp đúng.

Ta có thể đặt một câu hỏi tổng quát hơn như sau : *Cho biết những đoạn thẳng thẳng đứng có một mút cho trước là (2, 3)* ?

```
?- vertical(seg(point(2, 3), P)).
P = point(2, _0104)
Yes
```

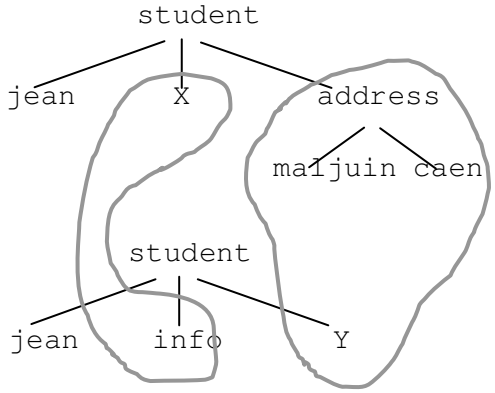
Câu trả lời có nghĩa là mọi đường thẳng có phương trình X = 2 là thẳng đứng. Chú ý rằng ở đây, ta không nhận được tên biến như mong muốn (là Y), mà tuỳ theo phiên bản cài đặt cụ thể, Prolog sẽ tạo ra một tên biến khi thực hiện chương trình, `_0104` trong ví dụ trên, nhằm tránh đặt lại tên biến của NSD với hai lý do như sau. Thứ nhất, cùng một tên biến nhưng xuất hiện trong các mệnh đề khác nhau thì sẽ biểu diễn những biến khác nhau. Thứ hai, do khi áp dụng liên tiếp cùng một mệnh đề, chính bản «copy» được sử dụng với một bộ biến khác.

Bây giờ ta đặt tiếp một câu hỏi thú vị như sau : *Có tồn tại một đoạn thẳng vừa thẳng đứng vừa nằm ngang hay không* ?

```
?- vertical(S), horizontal(S).
S = seg(point(_00E8, _00EC), point(_00E8, _00EC))
Yes
```

Câu trả lời có nghĩa là mọi đoạn thẳng khi suy biến thành một điểm thì vừa thẳng đứng, lại vừa nằm ngang. Ta thấy rằng kết quả nhận được là nhờ so khớp. Ở đây, các tên biến `_00E8` và `_00EC`, tương ứng với X và Y, đã được tạo ra bởi Prolog.

Sau đây là một ví dụ khác minh hoạ hai cấu trúc được số khớp với nhau.



Hình IV.7. Kết quả so khớp :
student(jean, X, address (maljuin, caen)) = student(jean,info, Y).

Tóm tắt chương 1

Chương 1 đã trình bày những yếu tố sơ cấp của Prolog, rất gần gũi với lôgich hình thức. Những điểm quan trọng mà ta có được là :

- Những đối tượng sơ cấp của Prolog là *nguyên tử*, *biến* và *số*. Các đối tượng có cấu trúc, hay *cấu trúc*, dùng để biểu diễn các đối tượng có nhiều thành phần.
- Các *hàm tử* dùng để xây dựng các cấu trúc. Mỗi hàm tử được định nghĩa bởi tên và thứ nguyên (dimension).
- Kiểu của một đối tượng được định nghĩa hoàn toàn nhờ vào sự xuất hiện về mặt cú pháp của nó.
- *Tầm vực từ vựng* (lexical scope) của một biến là duy nhất mệnh đề mà biến xuất hiện. Cùng một tên biến xuất hiện trong hai mệnh đề sẽ tương ứng với hai biến khác nhau.
- Các cấu trúc được biểu diễn rất đơn giản bởi các cây. Prolog được xem như là một ngôn ngữ xử lý cây.
- Phép toán *so khớp* so sánh hai phần tử (term) và tìm cách đồng nhất chúng bởi các ràng buộc của chúng.
- Nếu so khớp thành công, Prolog đưa ra ràng buộc các biến *tổng quát nhất*.
- Những khái niệm đã trình bày là :
  mệnh đề, sự kiện, luật, câu hỏi,
  nguyên tử, biến, biến ràng buộc,
  phần đầu và phần thân của của một mệnh đề,

luật đệ quy, định nghĩa đệ quy,
đích,
đối tượng : nguyên tử, số, biến, hạng
cấu trúc
hàm tử, thứ nguyên của một hàm tử
hàm tử chính của một hạng
so khớp các hạng
ràng buộc tổng quát nhất

Bài tập chương 1

1. Tìm các đối tượng Prolog đúng đắn về mặt cú pháp trong số đối tượng được cho dưới đây. Cho biết kiểu của chúng (là nguyên tử, số, biến hay cấu trúc) ?

```
a) Diane
b) diane
c) 'Diane'
d) _diane
e) 'Diane va en vacances'
f) va( diane, vacances )
g) 45
h) 5(X , Y)
i) +( nord , owest )
j) three( small( cats ) )
```

2. Hãy tìm một biểu diễn dạng đối tượng cấu trúc cho các hình chữ nhật, hình vuông và hình tròn. Xem hình 2.4 để có cách giải quyết. Sử dụng các biểu diễn cho các hình cụ thể để minh họa.
3. Chương trình sau nói rằng hai người là có quan hệ dòng họ với nhau nếu :
   a) một là tổ tiên của người kia, hoặc,
   b) hai người có chung tổ tiên, hoặc,
   c) hai người có cùng con cháu.

```
kindred( X, Y) :-
     ancestor(X , Y).
kindred(X , Y) :-
     ancestor(X , Y).
kindred(X , Y) :-   % X và Y có cùng tổ tiên
     ancestor( Z, X),
     ancestor(Z , Y).
```

```
kindred(X , Y) :-    % X và Y có cùng con cháu
     ancestor (X , Z),
     ancestor(Y , Z).
```

Hãy cho biết có thể làm ngắn chương trình trên bằng cách sử dụng dấu chấm phẩy ; được không ?

4. Hãy tìm hiểu một định nghĩa mới về quan hệ `ancestor` :

```
ancestor(X Z) :-
     parent(X Z) .
ancestor(X Z) :-
     parent(Y , Z),
     ancestor( X, Y).
```

Định nghĩa này có đúng hay không ? Có thể thay đổi lại sơ đồ đã cho trong hình 1.7 để tương ứng với định nghĩa mới này ?

5. Ngoài các định nghĩa quan hệ gia đình đã có trong phần lý thuyết và bài tập, hãy định nghĩa các quan hệ khác theo tập quán Việt Nam (cô, dì, chú, bác...) ?
6. Hãy định nghĩa các quan hệ trong thế giới sinh vật (động vật, thực vật) ?
7. Cho biết các hạng Prolog hợp thức sau đây (valid Prolog terms) :

```
23                              +(fred, jim)
foo(X, bar(+(3, 4)))            1+2.
Foo(x)                          Alison Cawsey
```

8. Cho quan hệ `parent` được định nghĩa trong phần lý thuyết cho biết kết quả của các câu hỏi sau :

```
a) ?- parent(jim , X).
b) ?- parent( X , jim).
c) ?- parent(mary , X) , parent( X , part).
d) ?- parent(mary , X) , parent( X , y ) , parent(y ,
   jim).
```

9. Viết các mệnh đề Prolog diễn tả các câu hỏi liên quan đến quan hệ `parent` :
   a) Ai là cha mẹ của Sue ?
   b) Liz có con không ?
   c) Ai là ông bà (grandparent) của Sue ?
10. Viết trong Prolog các mệnh đề sau :
    a) Ai có một đứa trẻ người đó là hạnh phúc.
       Hướng dẫn : Xây dựng quan hệ một ngôi `happy`.
    b) Với mọi X, nếu X có một con mà người con này có một chị em gái, thì X có hai con (xây dựng quan hệ `have_two_children`).
11. Định nghĩa quan hệ `grandchild` bằng cách sử dụng quan hệ `parent`.
    Hướng dẫn : tìm hiểu quan hệ `grandparent`.

12. Định nghĩa quan hệ `aunt( X, Y )` bằng cách sử dụng quan hệ `parent`. Để thuận tiện, có thể vẽ sơ đồ minh họa.
13. Các phép so khớp dưới đây có đúng không ? Nếu đúng, cho biết các ràng buộc biến tương ứng ?
    a) `point( A , B ) = point( 1 , 2 )`
    b) `point( A , B ) = point( X , Y, Z )`
    c) `addition( 2 , 2 ) = 4`
    d) `+( 2 , D ) = +( E , 2 )`
    e) `triangle( point( -1 , 0 ) , P2, P3 ) = triangle( P1, point( 1, 0 ), point( 0, Y ) )`

    Các ràng buộc ở đây đã định nghĩa một lớp các tam giác. Làm cách nào để mô tả lớp này ?
14. Sử dụng mô tả các đường thẳng đã cho trong bài học, tìm một hạng biểu diễn mọi đường thẳng đứng `X = 5`.
15. Cho hình chữ nhật được biểu diễn bởi hạng: `rectangle(P1, P2, P3, P4)`.

    Với `Pi` là các đỉnh của hình chữ nhật theo chiều dương, hãy định nghĩa quan hệ :

    `regular( R )`

    là đúng nếu `R` là một hình chữ nhật có các cạnh thẳng đứng và nằm ngang (song song với các trục tọa độ).

CHƯƠNG 2

Ngữ nghĩa của chương trình Prolog

I. Quan hệ giữa Prolog và lôgich toán học

Prolog có quan hệ chặt chẽ với lôgich toán học. Dựa vào lôgich toán học, người ta có thể diễn tả cú pháp và nghĩa của Prolog một cách ngắn gọn và súc tích. Tuy nhiên không vì vậy mà những người học lập trình Prolog cần phải biết một số khái niệm về lôgich toán học. Thật may mắn là những khái niệm về lôgich toán học không thực sự cần thiết để có thể hiểu và sử dụng Prolog như là một công cụ lập trình. Sau đây là một số quan hệ giữa Prolog và lôgich toán học.

Prolog có cú pháp là những công thức *lôgich vị từ bậc một* (first order predicate logic), được viết dưới dạng các *mệnh đề* (các lượng tử ∀ và ∃ không xuất hiện một cách tường minh), nhưng hạn chế chỉ đơn thuần ở dạng *mệnh đề Horn*, là những mệnh đề chỉ có ít nhất một trực kiện dương (positive literals). Năm 1981, Clocksin và Mellish đã đưa ra một chương trình Prolog chuyển các công thức tính vị từ bậc một thành dạng các mệnh đề.

Cách Prolog diễn giải chương trình là theo kiểu Toán học : Prolog xem các sự kiện và các luật như là các tiên đề, xem câu hỏi của NSD như là một định lý cần phỏng đoán. Prolog sẽ tìm cách chứng minh định lý này, nghĩa là chỉ ra rằng định lý có thể được suy luận một cách lôgich từ các tiên đề.

Về mặt thủ tục, Prolog sử dụng phương pháp *suy diễn quay lui* (back chaining) để *hợp giải* (resolution) bài toán, được gọi là chiến lược hợp giải SLD (Selected, Linear, Definite : **L**inear resolution with a **S**election function for **D**efinite sentences) do J. Herbrand và A. Robinson đề xuất năm 1995.

Có thể tóm tắt như sau : để chứng minh `P(a)`, người ta tìm sự kiện

```
P(t)
```

hoặc một luật :

```
P(t) :- L1, L2, …, Ln
```

sao cho `a` có thể hợp nhất (unifiable) được với `t` nhờ so khớp. Nếu tìm được `P(t)` là sự kiện như vậy, việc chứng minh kết thúc. Còn nếu tìm được `P(t)` là luật, cần lần lượt chứng minh vế bên phải `L1`, `L2`, ..., `Ln` của nó.

Trong Prolog, câu hỏi luôn luôn là một dãy từ một đến nhiều đích. Prolog trả lời một câu hỏi bằng cách tìm kiếm để *xoá* (erase) tất cả các đích. Xoá một đích nghĩa là chứng minh rằng đích này được thoả mãn, với giả thiết rằng các quan hệ của chương trình là đúng. Nói cách khác, xoá một đích có nghĩa là đích này được suy ra một cách lôgich bởi các sự kiện và luật chứa trong chương trình.

Nếu có các biến trong câu hỏi, Prolog tìm các đối tượng để thay thế vào các biến, sao cho đích được thoả mãn. Sự ràng buộc giá trị của các biến tương ứng với việc hiển thị các đối tượng này. Nếu Prolog không thể tìm được ràng buộc cho các biến sao cho đích được suy ra từ chương trình thì nó sẽ trả lời `No`.

II. Các mức nghĩa của chương trình Prolog

Cho đến lúc này, qua các ví dụ minh hoạ, ta mới chỉ hiểu được tính đúng đắn về kết quả của một chương trình Prolog, mà chưa hiểu được làm cách nào để hệ thống tìm được lời giải. Một chương trình Prolog có thể được hiểu theo *nghĩa khai báo* (declarative signification) hoặc theo *nghĩa thủ tục* (procedural signification). Vấn đề là cần phân biệt hai mức nghĩa của một chương trình Prolog, là *nghĩa khai báo* và *nghĩa thủ tục*. Người ta còn phân biệt mức nghĩa thứ ba của một chương trình Prolog là *nghĩa lôgich* (logical semantic).

Trước khi định nghĩa một cách hình thức hai mức ngữ nghĩa khai báo và thủ tục, ta cần phân biệt sự khác nhau giữa chúng. Cho mệnh đề :

```
P :- Q, R.
```

với `P`, `Q`, và `R` là các hạng nào đó.

Theo nghĩa khai báo, ta đọc chúng theo hai cách như sau :

- `P` là đúng nếu cả `Q` và `R` đều đúng.
- `Q` và `R` dẫn ra `P`.

Theo nghĩa thủ tục, ta cũng đọc chúng theo hai cách như sau :

- Để giải bài toán `P`, *đầu tiên*, giải bài toán con `Q`, *sau đó* giải bài toán con `R`.
- Để xoá `P`, *đầu tiên*, xoá `Q`, *sau đó* xoá `R`.

Sự khác nhau giữa nghĩa khai báo và nghĩa thủ tục là ở chỗ, nghĩa thủ tục không định nghĩa các quan hệ lôgich giữa phần đầu của mệnh đề và các đích của thân, mà chỉ định nghĩa *thứ tự* xử lý các đích.

II.1. Nghĩa khai báo của chương trình Prolog

Về mặt hình thức, nghĩa khai báo, hay *ngữ nghĩa chủ ý* (intentional semantic) xác định các mối quan hệ đã *được định nghĩa* trong chương trình. Nghĩa khai báo xác định những gì là kết quả (đích) mà chương trình phải tính toán, phải tạo ra.

Nghĩa khai báo của chương trình xác định nếu một đích là đúng, và trong trường hợp này, xác định giá trị của các biến. Ta đưa vào khái niệm *thể nghiệm* (instance) của một mệnh đề C là mệnh đề C mà mỗi một biến của nó đã được thay thế bởi một hạng. Một *biến thể* (variant) của một mệnh đề C là mệnh đề C sao cho mỗi một biến của nó đã được thay thế bởi một biến khác.

Ví dụ II.1 : Cho mệnh đề :

```
hasachild(X) :-
        parent(X, Y).
```

Hai biến thể của mệnh đề này là :

```
hasachild(A) :-
        parent(A, B).

hasachild(X1) :-
        parent(X1, X2).
```

Các thể nghiệm của mệnh đề này là :

```
hasachild(tom) :-
        parent(tom, Z).

hasachild(jafa) :-
        parent(jafa, small(iago)).
```

Cho trước một chương trình và một đích G, nghĩa khai báo nói rằng :

Một đích G là đúng (thoả mãn, hay suy ra được từ chương trình một cách logic) nếu và chỉ nếu

(1) tồn tại một mệnh đề C của chương trình sao cho

(2) tồn tại một thể nghiệm I của mệnh đề C sao cho:

  (a) phần đầu của I là giống hệt G, và

  (b) mọi đích của phần thân của I là đúng.

Định nghĩa trên đây áp dụng được cho các câu hỏi Prolog. Câu hỏi là một danh sách các đích ngăn cách nhau bởi các dấu phẩy. Một danh sách các đích là đúng nếu *tất cả* các đích của danh sách là đúng cho *cùng một* ràng buộc của các biến. Các giá trị của các biến là những giá trị ràng buộc tổng quát nhất.

II.2. Khái niệm về gói mệnh đề

Một *gói* hay *bó mệnh đề* (packages of clauses) là tập hợp các mệnh đề có cùng tên hạng tử chính (cùng tên, cùng số lượng tham đối). Ví dụ sau đây là một gói mệnh đề :

```
a(X) :- b(X, _).
a(X) :- c(X), e(X).
a(X) :- f(X, Y).
```

Gói mệnh đề trên có ba mệnh đề có cùng hạng là `a(X)`. Mỗi mệnh đề của gói là một phương án giải quyết bài toán đã cho.

Prolog quy ước :

- mỗi *dấu phẩy* (comma) đặt giữa các mệnh đề, hay các đích, đóng vai trò *phép hội* (conjunction). Về mặt lôgich, chúng phải đúng *tất cả*.
- mỗi *dấu chấm phẩy* (semicolon) đặt giữa các mệnh đề, hay các đích, đóng vai trò *phép tuyển* (disjunction). Lúc này chỉ cần một trong các đích của danh sách là đúng.

Ví dụ II.2 :

```
P :- Q; R.
```

được đọc là : `P` đúng nếu `Q` đúng hoặc `R` đúng. Người ta cũng có thể viết tách ra thành hai mệnh đề :

```
P :- Q.
P :- R.
```

Trong Prolog, dấu phẩy (phép hội) có mức độ ưu tiên cao hơn dấu chấm phẩy (phép tuyển). Ví dụ :

```
P :- Q, R; S, T, U.
```

được hiểu là :

```
P :- (Q, R); (S, T, U).
```

và có thể được viết thành hai mệnh đề :

```
P :- (Q, R).
P :- (S, T, U).
```

Hai mệnh đề trên được đọc là : `P` đúng nếu hoặc cả `Q` và `R` đều đúng, hoặc cả `S`, `T` và `U` đều đúng.

Về nguyên tắc, thứ tự thực hiện các mệnh đề trong một gói là không quan trọng, tuy nhiên trong thực tế, cần chú ý tôn trọng thứ tự của các mệnh đề. Prolog sẽ lần lượt thực hiện theo thứ tự xuất hiện các mệnh đề trong gói và trong chương trình theo mô hình tuần tự bằng cách thử quay lui mà ta sẽ xét sau đây.

II.3. Nghĩa lôgich của các mệnh đề

Nghĩa lôgich thể hiện mối liên hệ giữa đặc tả lôgich (logical specification) của bài toán cần giải với bản thân chương trình.

1. Các mệnh đề không chứa biến

Mệnh đề	*Nghĩa lôgich*	*Ký hiệu Toán học*
`P(a).`	`P(X)` đúng **nễu** `X = a`	`P(X)` ⇔ `X = a`
`P(a).` `P(b).`	`P(X)` đúng **nễu** `X = a` hoặc `X = b`	`P(X)` ⇔ `(X = a) ∨ (X = b)`
`P(a) :-` `Q(c).`	`P(X)` đúng **nễu** `X = a` và `Q(c)` đúng	`P(X)` ⇔ `X = a ∧ Q(c)`

`P(a) :-`
`Q(c).`
`P(b).`
	`P(X)` đúng **nễu hoặc** (`X = a` và `Q(c)` đúng) **hoặc** `X = b`	`P(X)` ⇔ `(X = a ∧ Q(c)) ∨ (X = b)`

Quy ước : **nễu** = nếu và chỉ nếu.

2. Các mệnh đề có chứa biến

Mệnh đề	*Nghĩa lôgich*	*Ký hiệu Toán học*
`P(X).`	Với mọi giá trị của `X`, `P(X)` đúng	`∀X P(X)`
`P(X) :-` `Q(X).`	Với mọi giá trị của `X`, `P(X)` đúng **nễu** `Q(X)` đúng	`P(X)` ⇔ `Q(X)`
`P(X) :-` `Q(X, Y).`	Với mọi giá trị của `X`, `P(X)` đúng **nễu** tồn tại `Y` là một biến cục bộ sao cho `Q(X, Y)` đúng	`P(X)` ⇔ `∃Y Q(X, Y)`
`P(X) :-` `Q(X, _).`	Với mọi giá trị của `X`, `P(X)` đúng **nễu** tồn tại một giá trị nào đó của `Y` sao cho `Q(X, Y)` đúng (không quan tâm đến giá trị của `Y` như thế nào)	`P(X)` ⇔ `∃Y Q(X, Y)`

`P(X) :-`
`Q(X, Y),`
`R(Y).`
	Với mọi giá trị của `X`, `P(X)` đúng **nễu** tồn tại `Y` sao cho `Q(X, Y)` và `R(Y)` đúng	`P(X)` ⇔ `∃Y Q(X, Y) ∧ R(Y)`

`P(X) :-`
`Q(X, Y),`
`R(Y).`
`p(a).`
	Với mọi giá trị của `X`, `P(X)` đúng **nễu hoặc** tồn tại `Y` sao cho `Q(X, Y)` và `R(Y)` đúng, **hoặc** `X = a`	`P(X)` ⇔ `(∃Y Q(X, Y) ∧ R(Y)) ∨ (X = a)`

3. Nghĩa lôgich của các đích

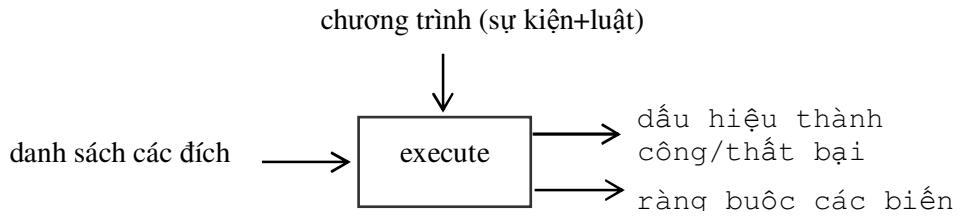
Đích	*Nghĩa lôgich*
`p(a).`	Có phải `p(a)` đúng (thoả mãn) ?
`p(a), Q(b).`	Có phải cả `p(a)` và `Q(b)` đều đúng ?
`P(X).`	Cho biết giá trị của `X` để `P(X)` là đúng ?
`P(X), Q(X, Y).`	Cho biết các giá trị của `X` và của `Y` để `P(X)` và `Q(X, Y)` đều là đúng ?

II.4. Nghĩa thủ tục của Prolog

Nghĩa thủ tục, hay *ngữ nghĩa thao tác* (operational semantic), lại xác định *làm cách nào* để nhận được kết quả, nghĩa là làm cách nào để các quan hệ *được xử lý* thực sự bởi hệ thống Prolog.

Nghĩa thủ tục tương ứng với cách Prolog trả lời các câu hỏi *như thế nào* (how) hay lập luận trên các tri thức. Trả lời một câu hỏi có nghĩa là tìm cách xóa một danh sách. Điều này chỉ có thể thực hiện được nếu các biến xuất hiện trong các đích này được ràng buộc sao cho chúng được suy ra một cách lôgich từ chương trình (hay từ các tri thức đã ghi nhận).

Prolog có nhiệm vụ thực hiện lần lượt từng đích trong một danh sách các đích từ một chương trình đã cho. «Thực hiện một đích» có nghĩa là tìm cách thoả mãn hay xoá đích đó khỏi danh sách các đích đó.



Hình II.1. Mô hình vào/ra của một thủ tục thực hiện một danh sách các đích.

Gọi thủ tục này là `execute` (thực hiện), cái vào và cái ra của nó như sau :

Cái vào : một chương trình và một danh sách các đích
Cái ra : một dấu hiệu thành công/thất bại và một ràng buộc các biến

Nghĩa của hai cái ra như sau :

(1) Dấu hiệu thành công/thất bại là `Yes` nếu các đích được thoả mãn (thành công), là `No` nếu ngược lại (thất bại).

(2) Sự ràng buộc các biến chỉ xảy ra nếu chương trình được thực hiện.

Ví dụ II.3 :

Minh hoạ cách Prolog trả lời câu hỏi cho ví dụ chương trình gia hệ trước đây như sau :

Đích cần tìm là :

```
?- ancestor(tom, sue)
```

Ta biết rằng `parent(bill, sue)` là một sự kiện. Để sử dụng sự kiện này và luật 1 (về tổ tiên trực tiếp), ta có thể kết luận rằng `ancestor(bill, sue)`. Đây là một sự kiện kéo theo : sự kiện này không có mặt trong chương trình, nhưng có thể được suy ra từ các luật và sự kiện khác. Ta có thể viết gọn sự suy diễn này như sau :

`parent(bill, sue)` ⇒ `ancestor(bill, sue)`

Nghĩa là `parent(bill, sue)` kéo theo `ancestor(bill, sue)` bởi luật 1. Ta lại biết rằng `parent(tom, bill)` cũng là một sự kiện. Mặt khác, từ sự kiện được suy diễn `ancestor(bill, sue)`, luật 2 (về tổ tiên gián tiếp) cho phép kết luận rằng `ancestor(tom, sue)`. Quá trình suy diễn hai giai đoạn này được viết :

`parent(bill, sue)` ⇒ `ancestor(bill, sue)`

`parent(tom, bill)` *và* `ancestor(bill, sue)` ⇒ `ancestor(tom, sue)`

Ta vừa chỉ ra các giai đoạn để xoá một đích, gọi là một chứng minh. Tuy nhiên, ta chưa chỉ ra *làm cách nào* Prolog nhận được một chứng minh như vậy.

Prolog nhận được phép chứng minh này theo thứ tự ngược lại những gì đã trình bày. Thay vì xuất phát từ các sự kiện chứa trong chương trình, Prolog bắt đầu bởi các đích và, bằng cách sử dụng các luật, nó thay thế các đích này bởi các đích mới, cho đến khi nhận được các sự kiện sơ cấp.

Để xoá đích :

```
?- ancestor(tom, sue).
```

Prolog tìm kiếm một mệnh đề trong chương trình mà đích này được suy diễn ngay lập tức. Rõ ràng chỉ có hai mệnh đề thoả mãn yêu cầu này là luật 1 và luật 2, liên quan đến quan hệ `ancestor`. Ta nói rằng phần đầu của các luật này *tương ứng* với đích.

Hai mệnh đề này biểu diễn hai khả năng mà Prolog phải khai thác xử lý. Prolog bắt đầu chọn xử lý mệnh đề thứ nhất xuất hiện trong chương trình :

```
ancestor(X, Z) :- parent(X, Z).
```

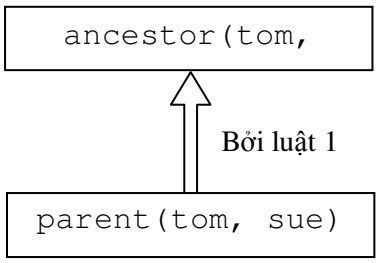
Do đích là `ancestor(tom, sue)`, các biến phải được ràng buộc như sau :

```
X = tom, Z = sue
```

Lúc này, đích ban đầu trở thành :

```
parent(tom, sue)
```

Hình dưới đây biểu diễn giai đoạn chuyển một đích thành đích mới sử dụng một luật. Thất bại xảy ra khi không có phần đầu nào trong các mệnh đề của chương trình tương ứng với đích `parent(tom, sue).`



Hình II.2. Xử lý bước đầu tiên :
Đích phía trên được thoả mãn nếu Prolog có thể xoá đích ở phía dưới.

Lúc này Prolog phải tiến hành *quay lui* (backtracking) trở lại đích ban đầu, để tiếp tục xử lý mệnh đề khác là luật thứ hai :

```
ancestor(X, Z) :- parent(X, Y), ancestor(Y, Z).
```

Tương tự bước xử lý thứ nhất, các biến X và Z được ràng buộc như sau :

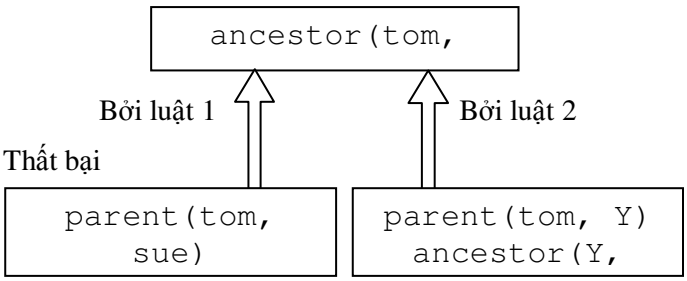
```
X = tom, Z = sue
```

Đích phía trên `ancestor(tom, sue)` được thay thế bởi hai đích là :

```
parent(tom, Y), ancestor(Y, sue).
```

Nhưng lúc này, Y chưa có giá trị. Lúc này cần xoá hai đích. Prolog sẽ tiến hành xoá theo thứ tự xuất hiện của chúng trong chương trình. Đối với đích thứ nhất, việc xoá rất dễ dàng vì đó là một trong các sự kiện của chương trình. Sự tương ứng sự kiện dẫn đến Y được ràng buộc bởi giá trị `bill`.

Các giai đoạn thực hiện được mô tả bởi cây hợp giải sau đây :



Hình II.3. Các giai đoạn thực hiện xử lý xoá đích.

Sau khi đích thứ nhất `parent(tom, bill)` thoả mãn, còn lại đích thứ hai :

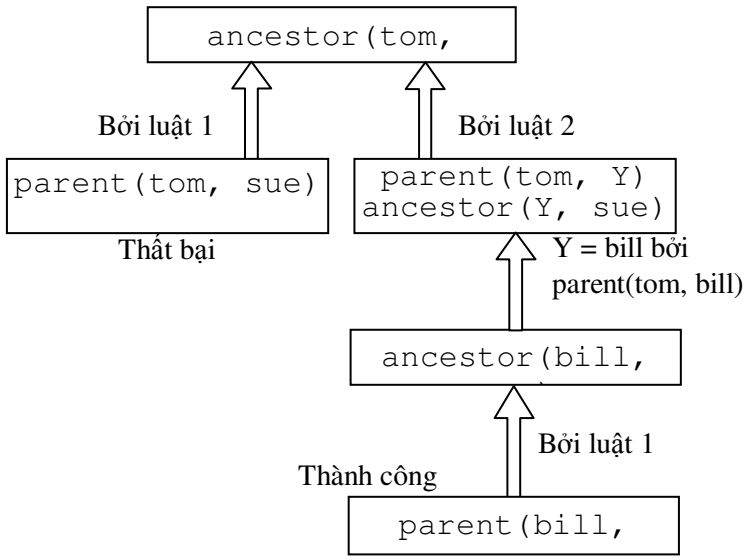
```
ancestor(bill, sue)
```

cũng phải được thoả mãn Một lần nữa, luật 1 được sử dụng. Chú ý rằng việc áp dụng lần thứ hai cùng luật này không liên quan gì đến lần áp dụng thứ nhất. Prolog sử dụng các biến mới mỗi lần luật được gọi đến. Luật 1 bây giờ có thể được đặt tên lại như sau :

```
ancestor(X', Z') :- parent(X', Z').
```

Phần đầu phải tương ứng với đích thứ nhất, `ancestor(bill, sue)`, tức là :

```
X' = bill, Z' = sue
```



Hình II.4. Quá trình thực hiện xoá đích `ancestor(tom, sue)`.

Từ đó đích (trong phần thân) phải thay thế bởi :

```
parent(bill, sue)
```

Đích này được thoả mãn ngay lập tức, vì chính là một sự kiện trong chương trình. Quá trình xử lý được minh hoạ lại đầy đủ trong Hình II.4.

Hình 2.4. có dạng một cây. Mỗi nút tương ứng với một đích, hay một danh sách các đích cần thoả mãn. Mỗi cung nối hai nút tương ứng với việc áp dụng một luật trong chương trình. Việc áp dụng một luật cho phép chuyển các đích của một nút thành các đích mới của một nút khác. Đích trên cùng (gốc của cây) được xoá khi tìm được một con đường đi từ gốc đến lá có nhãn là *thành công*. Một nút lá có nhãn là *thành công* khi trong nút là một sự kiện của chương trình. Việc thực thi một chương trình Prolog là việc tìm kiếm những con đường như vậy.

Nhánh bên phải chứng tỏ rằng có thể xoá đích.

Trong quá trình tìm kiếm, có thể xảy ra khả năng là Prolog đi trên một con đường không tốt. Khi gặp nút chứa một sự kiện không tồn tại trong chương trình, xem như thất bại, nút được gắn nhãn *thất bại*, ngay lập tức Prolog tự động quay lui lên nút phía trên, chọn áp dụng một mệnh đề tiếp theo có mặt trong nút này để tiếp tục con đường mới, chừng nào thành công.

Ví dụ trên đây, ta đã giải thích một cách không hình thức cách Prolog trả lời câu hỏi. Thủ tục `execute` dưới đây mô tả hình thức và có hệ thống hơn về quá trình này.

Để thực hiện danh sách các đích :

$G_1, G_2, ..., G_m$

thủ tục `execute` tiến hành như sau :

- Nếu danh sách các đích là rỗng, thủ tục *thành công* và dừng.
- Nếu danh sách các đích khác rỗng, thủ tục duyệt `scrutinize` sau đây được thực hiện

Thủ tục `scrutinize` :

Duyệt các mệnh đề trong chương trình bắt đầu từ mệnh đề đầu tiên, cho đến khi nhận được mệnh đề `C` có phần đầu trùng khớp với phần đầu của đích đầu tiên G_1.

Nếu không tìm thấy một mệnh đề nào như vậy, thủ tục rơi vào tình trạng *thất bại*.

Nếu mệnh đề `C` được tìm thấy, và có dạng :

H :- $D_1, ..., D_n$

khi đó, các biến của `C` được đặt tên lại để nhận được một biến thể `C'` không có biến nào chung với danh sách $G_1, G_2, ..., G_m$.

Mệnh đề `C'` như sau :

H' :- $D'_1, ..., D'_n$

Giả sử S là ràng buộc của các biến từ việc so khớp giữa G_1 và H′, Prolog thay thế G_1 bởi D'_1, ..., D'_n trong danh sách các đích để nhận được một danh sách mới :

D_1', ..., D_n', G_2, ..., G_m

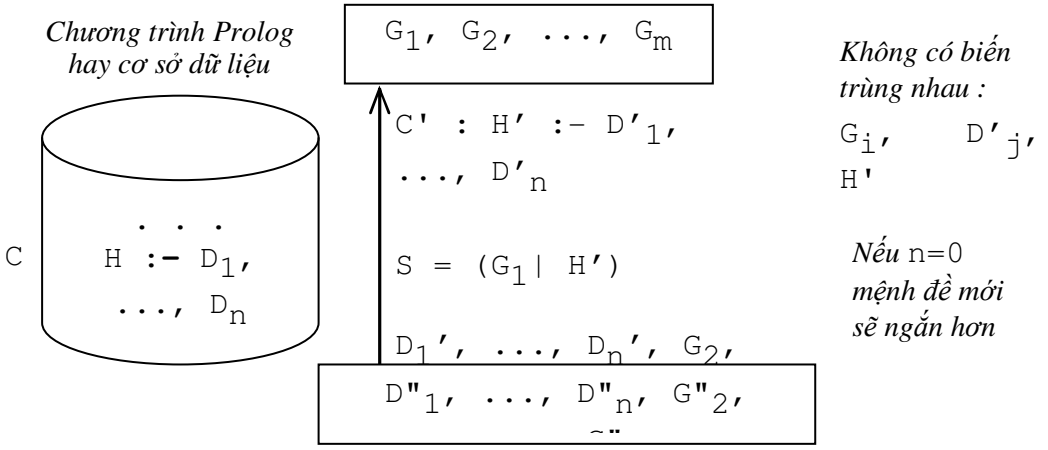
Chú ý rằng nếu C là một sự kiện, khi đó, n=0 và danh sách mới sẽ ngắn hơn danh sách cũ. Trường hợp danh sách mới rỗng, kết quả thành công.

Thay thế các biến của danh sách mới này bởi các giá trị mới chỉ định bởi ràng buộc S, ta nhận được một danh sách các đích mới :

D"$_1$, ..., D"$_n$, G"$_2$, ..., G"$_m$

Thực hiện thủ tục một cách đệ quy cho danh sách các đích mới này. Nếu kết thúc thành công, tiếp tục thực hiện danh sách ban đầu. Trong trường hợp ngược lại, Prolog bỏ qua danh sách các đích để quay lui lại thủ tục `scrutinize`. Quá trình tìm kiếm các mệnh đề trong chương trình được bắt đầu lại từ sau mệnh đề C, với một mệnh đề mới.

Quá trình thực hiện thủ tục `execute` được mô tả như sau :



Hình II.5. Quá trình thực hiện `execute`.

Sau đây là thủ tục `execute` được viết bằng giả ngữ Pascal.

```
Procedure execute(program, goallist, success);
{ Tham đối vào :
  program      danh sách các mệnh đề
  goallist     danh sách các đích
  Tham đối ra :
  success      kiểu Boolean, là true nếu goallist là true đối với tham
               đối program
  Các biến cục bộ :
  goal         đích
  othergoals   danh sách các đích
  satisfied    kiểu Boolean
  matchOK      kiểu Boolean
  process      ràng buộc của các biến
  H, H', D1, D1', ..., Dn, Dn'    các đích
  Các hàm phụ :
  empty(L)     có giá trị true nếu L là danh sách rỗng
  head(L)      trả về phần tử đầu tiên của danh sách L
  tail(L)      trả về danh sách L sau khi đã bỏ đi phần tử đầu tiên
  add(L1, L2)  ghép danh sách L2 vào sau danh sách L1
  match(T1, T2, matchOK, process)
               so khớp các hạng T1 và T2, nếu thành công,
               biến matchOK có giá trị true, và process chứa các ràng
               buộc tương ứng với các biến.
```

```
  substitute(process, goals)
                thay thế các biến của goals bởi giá trị ràng buộc tương ứng
                trong process.
}
begin { execute_main }
 if empty(goallist) then success:= true
 else begin
  goal:= head(goallist);
  othergoals:= tail(goallist);
  satisfied:= false;
  while not satisfied and there_are_again_some_terms do begin
     Let the following clause of program is:
     H :- D1, ..., Dn
     constructing a variant of this clause:
     H' :- D1', ..., Dn'
     match(goal, H', matchOK, process)
      if matchOK then begin
         newgoals:= add([ D1', ..., Dn' ], othergoals);
         newgoals:= substitute(process, newgoals);
         execute(program, newgoals, satisfied)
      end { if }
  end; { while }
  satisfied:= satisfied
 end
end; { execute_main }
```

Từ thủ tục `execute` trên đây, ta có một số nhận xét sau. Trước hết, thủ tục không mô tả làm cách nào để nhận được ràng buộc cuối cùng cho các biến. Chính ràng buộc S đã dẫn đến thành công nhờ các lời gọi đệ quy.

Mỗi lần lời gọi đệ quy `execute` thất bại (tương ứng với mệnh đề `C`), thủ tục `scrutinize` tìm kiếm mệnh đề tiếp theo ngay sau mệnh đề `C`. Quá trình thực thi là hiệu quả, vì Prolog bỏ qua những phần vô ích để rẽ sang nhánh khác.

Lúc này, mọi ràng buộc cho biến thuộc nhánh vô ích bị loại bỏ hoàn toàn. Prolog sẽ lần lượt duyệt hết tất cả các con đường có thể để đến thành công.

Ta cũng đã thấy rằng ngay sau khi có một kết quả tích cực, NSD có thể yêu cầu hệ thống quay lui để tìm kiếm một kết quả mới. Chi tiết này đã không được xử lý trong thủ tục `execute`. Trong các cài đặt Prolog hiện nay, nhiều khả năng mới đã được thêm vào nhằm đạt hiệu quả tối ưu. Không phải mọi mệnh đề trong

chương trình đều được duyệt đến, mà chỉ duyệt những mệnh đề có liên quan đến đích hiện hành.

Ví dụ II.4 :

Cho chương trình :

```
thick(bear).          % clause 1
thick(elephant).      % clause 2
small(cat).           % clause 3
brown(bear).          % clause 4
grey(elephant).       % clause 5
black(cat).           % clause 6
dark(Z) :- black(Z). % clause 7: all this who is black is
dark
dark(Z) :- brown(Z). % clause 8: all this who is brown is
dark
```

Câu hỏi :

```
?- dark(X), thick(X).% who is thick and dark ?
X = bear
Yes
```

(1) Danh sách ban đầu của các đích là : `dark(X), thick(X).`

(2) Tìm kiếm (duyệt) từ đầu đến cuối chương trình một mệnh đề có phần đầu tương ứng với đích đầu tiên `dark(X)`. Prolog tìm được mệnh đề 7 :

```
dark(Z) :- black(Z).
```

Thay thế đích đầu tiên bởi phần thân của mệnh đề 7 sau khi đã được ràng buộc (thế biến Z bởi X) để nhận được một danh sách các đích mới :

```
black(X), thick(X).
```

(3) Tìm kiếm trong chương trình một mệnh đề sao cho đích con `black(X)` được so khớp : tìm được mệnh đề 6 là sự kiện `black(cat)`. Lúc này, ràng buộc thành công, danh sách các đích thu gọn thành :

```
thick(cat)
```

(4) Tìm kiếm đích con `thick(cat)`. Do không tìm thấy mệnh đề nào thoả mãn, Prolog quay lui lại giai đoạn (3). Ràng buộc `X=cat` bị loại bỏ. Danh sách các đích trở thành :

```
black(X), thick(X).
```

Tiếp tục tìm kiếm trong chương trình bắt đầu từ mệnh đề 6. Do không tìm thấy mệnh đề nào thoả mãn, Prolog quay lui lại giai đoạn (2) để tiếp tục tìm kiếm bắt đầu từ mệnh đề 7. Kết quả tìm được mệnh đề 8 :

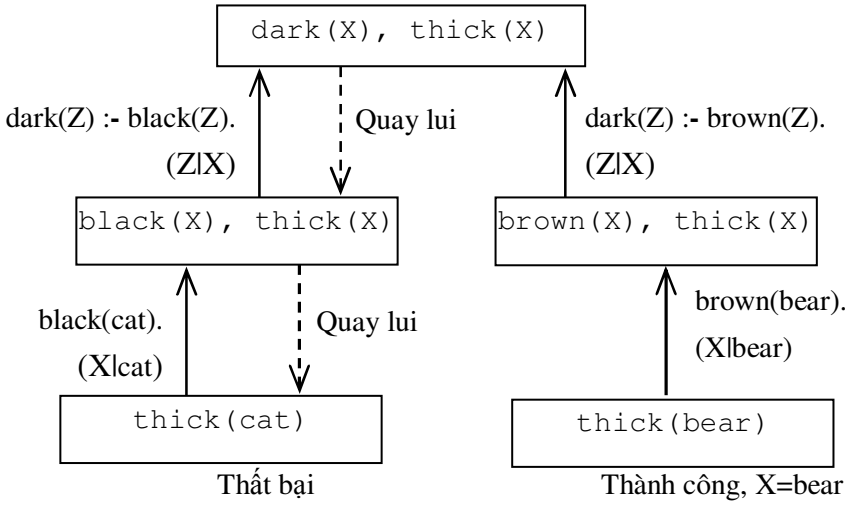
```
dark(Z) :- brown(Z).
```

Sau khi thay thế bởi `brown(X)` trong danh sách các đích, ta nhận được :
`brown(X), thick(X)`

(5) Tìm kiếm cho ràng buộc `brown(X)`, được kết quả là `brown(bear)`. Mệnh đề này là một sự kiện, danh sách các đích thu gọn lại thành :
`thick(bear)`

(6) Việc tìm kiếm trong chương trình dẫn đến kết quả `thick(bear)`. Do đây là một sự kiện, danh sách các đích trở nên rỗng. Điều này chứng tỏ chương trình đã thực hiện thành công, sự ràng buộc cho biến là :
`X = bear`

Quá trình thực hiện được giải thích trong hình dưới đây.



Hình II.6. Quá trình thực hiện xoá đích `dark(X), thick(X)`.

II.5. Tổ hợp các yếu tố khai báo và thủ tục

Người ta thường quan tâm đến tính ưu việt của Prolog là khả năng tự quản lý các chi tiết thủ tục. Điều này cho phép NLT (NLT) dự kiến trước được nghĩa khai báo của một chương trình một cách độc lập với nghĩa thủ tục của nó. Về nguyên tắc, do kết quả thực hiện của một chương trình phụ thuộc vào phần khai báo, nên phải khai báo đầy đủ các sự kiện, luật và quan hệ khi lập trình. Điều này mang tính thực tiễn, vì luôn luôn các yếu tố khai báo của một chương trình dễ hiểu hơn so với các chi tiết thủ tục.

Để tận dụng được khả năng tự quản lý các chi tiết thủ tục của Prolog, NLT phải tập trung đặc biệt vào yếu tố khai báo, tránh nhầm lẫn trong chừng mực có thể bởi các chi tiết thực hiện chương trình. Cần để cho Prolog tự giải quyết các chi tiết mang tính thủ tục này.

Nhờ tiếp cận khai báo, lập trình trên Prolog luôn luôn thuận tiện hơn so với các ngôn ngữ thủ tục khác như Pascal. Tuy nhiên, tiếp cận khai báo không phải luôn luôn đầy đủ. Như sẽ thấy sau này, đối với các chương trình lớn, không thể loại bỏ hoàn toàn tính tiếp cận thủ tục, do tính hiệu quả thực tiễn của nó khi thực hiện chương trình.

Vì vậy, tuỳ theo chương trình Prolog mà sử dụng hoàn toàn yếu tố khai báo, loại bỏ yếu tố thủ tục khi ràng buộc thực tiễn cho phép.

Như sẽ thấy trong chương sau rằng việc sắp đặt thứ tự các mệnh đề và các đích cũng như thứ tự các hạng trong mỗi mệnh đề có vai trò quan trọng trong việc tìm ra kết quả. Mặt khác, một số chương trình tuy đúng đắn về mặt khai báo nhưng lại không chạy được trong thực tế. Ranh giới giữa yếu tố thủ tục và yếu tố khai báo rất khó suy xét. Mệnh đề sau đây là một minh chứng về việc khai báo đúng, nhưng lại hoàn toàn vô ích về mặt chạy chương trình :

```
ancestor(X, Z) :- ancestor(X, Z).
```

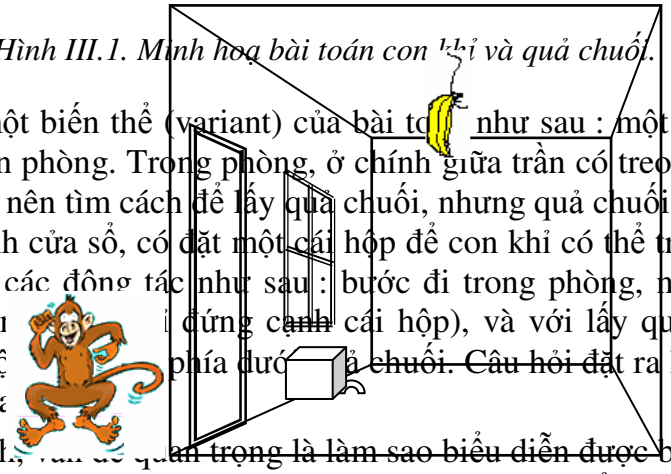
Do những tiến bộ của kỹ thuật lập trình, người ta quan tâm đến nghĩa khai báo để bỏ qua những chi tiết thủ tục, tận dụng những chi tiết khai báo làm lời giải đơn giản hơn và dễ hiểu hơn. Không phải là NLT, mà chính hệ thống phải quản lý những chi tiết thủ tục. Prolog là ngôn ngữ nhằm vào mục đích này. Như ta đã thấy, Prolog chỉ giúp quản lý đúng đắn một phần những chi tiết thủ tục, mà không thể quản lý được tất cả.

Một yếu tố thực tế nữa là người ta dễ dàng chấp nhận một chương trình chạy được (đúng nghĩa thủ tục) hơn là một chương trình chỉ đúng đắn về mặt khai báo mà chưa chạy được. Vì vậy, để giải quyết một bài toán nào đó một cách có lợi, người ta tập trung giải quyết những yếu tố khai báo, tiến hành chạy thử chương trình trên máy, rồi sắp đặt lại các mệnh đề và các đích nếu nó vẫn chưa chạy đúng về mặt thủ tục.

III. Ví dụ : con khỉ và quả chuối

III.1. Phát biểu bài toán

Trong trí tuệ nhân tạo, người ta thường lấy đề tài *con khỉ và quả chuối* (monkey and banana problem) để minh hoạ việc hợp giải bài toán. Sau đây, ta sẽ trình bày làm cách nào để vận dụng so khớp và quay lui cho những ứng dụng như vậy. Ta sẽ triển khai một cách phi thủ tục, sau đó nghiên cứu tính thủ tục một cách chi tiết.



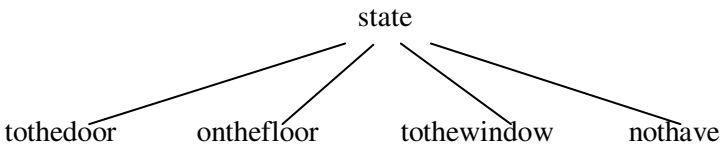
Hình III.1. Minh hoạ bài toán con khỉ và quả chuối.

Ta sử dụng một biến thể (variant) của bài toán như sau : một con khỉ đang ở trước cửa một căn phòng. Trong phòng, ở chính giữa trần có treo một quả chuối. Con khỉ đang đói nên tìm cách để lấy quả chuối, nhưng quả chuối lại treo quá cao đối với nó. Ở cạnh cửa sổ, có đặt một cái hộp để con khỉ có thể trèo lên. Con khỉ có thể thực hiện các động tác như sau : bước đi trong phòng, nhảy lên hộp, di chuyển cái hộp (nếu nó đang đứng cạnh cái hộp), và với lấy quả chuối nếu nó đang đứng trên hộp ngay phía dưới quả chuối. Câu hỏi đặt ra là con khỉ có ăn được quả chuối hay không ?

Trong lập trình, vấn đề quan trọng là làm sao biểu diễn được bài toán phù hợp với ngôn ngữ đang sử dụng. Trường hợp của chúng ta có thể nghĩ đến *trạng thái* của con khỉ có thể biến đổi theo thời gian. Trạng thái hiện hành được xác định bởi vị trí của các đối tượng. Chẳng hạn, trạng thái ban đầu của con khỉ được xác định bởi :

(1) Con khỉ đang ở trước cửa (to the door).
(2) Con khỉ đang ở trên sàn nhà (on the floor).
(3) Cái hộp đang ở cạnh cửa sổ (to the window).
(4) Con khỉ chưa lấy được quả chuối (not have).

Ta có thể nhóm bốn thông tin trên thành một đối tượng có cấu trúc duy nhất. Gọi `state` là hàm tử mà ta lựa chọn để nhóm các thành phần của đối tượng. Hình 2.9. trình bày cách biểu diễn trạng thái đầu là một tượng có cấu trúc.



Hình III.2. Trạng thái đầu của con khỉ là một đối tượng có cấu trúc gồm bốn thành phần : vị trí nằm ngang, vị trí thẳng đứng của con khỉ, vị trí của cái hộp và một chỉ dẫn cho biết con khỉ đã lấy được quả chuối chưa.

III.2. Giải bài toán với Prolog

Bài toán con khỉ và quả chuối được xem như một trò chơi chỉ có một người chơi. Ta hình thức hoá bài toán như sau : đầu tiên, đích của trò chơi là tình huống con khỉ lấy được quả chuối, nghĩa là một trạng thái `state` bốn thành phần, thành phần thứ tư là `possessing` (chiếm hữu) :

```
state(_, _, _, possessing)
```

Tiếp theo, ta tìm các động tác của con khỉ để chuyển từ một trạng thái này sang một trạng thái khác. Có bốn kiểu động tác (movement) như sau :

(1) Nắm lấy quả chuối (grab).

(2) Trèo lên hộp (climbing).

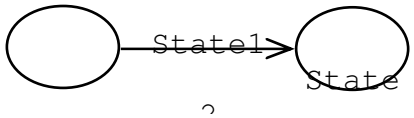
(3) Đẩy cái hộp (pushing).

(4) Di chuyển (walking).

Tuỳ theo trạng thái hiện hành, không phải tất cả mọi động tác đều có thể sử dụng. Chẳng hạn, động tác «nắm lấy quả chuối chỉ» có thể xảy ra khi con khỉ đã đứng trên cái hộp, ở đúng vị trí phía dưới quả chuối (ở chính giữa phòng), và nó chưa nắm lấy quả chuối. Quy tắc Prolog `displacement` dưới đây có ba đối số mô tả di chuyển của con khỉ như sau :

```
displacement(State1, Movement, State2).
```

Vai trò của các đối số dùng thể hiện di chuyển là :



Hình III.3. Di chuyển trạng thái.

Quy ước `state1` là trạng thái trước khi di chuyển, `M` là di chuyển đã thực hiện, và `state2` là trạng thái sau khi di chuyển. Động tác «nắm lấy quả chuối» với điều kiện đầu cần thiết được định nghĩa bởi mệnh đề có nội dung : «sau khi di chuyển, con khỉ đã lấy được quả chuối, và nó đang đứng trên cái hộp, ở giữa căn phòng». Mệnh đề được viết trong Prolog như sau :

```
displacement(
   state(tothecenter, onthebox, tothecenter, nothave),
                                          % trước khi di
   chuyển
   grab,                                  % di chuyển
   state(tothecenter, onthebox, tothecenter,
   possessing).                           % sau khi di chuyển
```

Một cách tương tự, ta có thể diễn tả di chuyển của con khỉ trên sàn nhà từ một vị trí nằm ngang P1 bất kỳ nào đó đến một vị trí mới P2. Việc di chuyển của con khỉ là độc lập với vị trí của cái hộp, và độc lập với sự kiện con khỉ đã lấy được quả chuối hay là chưa :

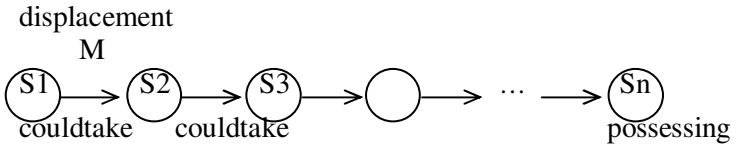
```
displacement(
   state(P1, onthefloor, G, H),
   walking(P1, P2),              % di chuyển từ P1 đến P2
   state(P2, onthefloor, G, H).
```

Mệnh đề trên đây có rất nhiều nghĩa :

- Di chuyển đã thực hiện là «đi từ P1 đến P2».
- Con khỉ ở trên sàn nhà trước và sau khi di chuyển.
- Vị trí hộp là G không thay đổi sau khi di chuyển.
- Quả chuối vẫn ở vị trí cũ trước và sau khi di chuyển (chưa bị con khỉ lấy đi).

Mệnh đề này đặc trưng cho một tập hợp đầy đủ các động tác vì có thể áp dụng cho bất kỳ một tình huống nào tương ứng với trạng thái đã chỉ ra trước khi di chuyển. Người ta gọi các mệnh đề kiểu này là một *sơ đồ* di chuyển.

Hai kiểu hành động khác là «đẩy» và «trèo» cũng được đặc trưng một cách tương tự.



Hình III.4. Dạng đệ quy của vị từ `couldtake`.

Câu hỏi đặt ra cho bài toán sẽ là «Xuất phát từ vị trí đầu S, con khỉ có thể lấy được quả chuối không ?» với vị từ sau đây :

```
couldtake(S)
```

với tham đối S là một trạng thái chỉ vị trí của con khỉ. Chương trình xây dựng cho vị từ này dựa trên hai quan sát sau đây :

(1) Với mỗi trạng thái S mà con khỉ đã lấy được quả chuối, vị từ `couldtake` có giá trị `true`, không cần một di chuyển nào khác nữa. Điều này tương ứng với sự kiện :

```
couldtake(state(_, _, _, possessing)).
```

(2) Trong các trường hợp khác, cần thực hiện một hoặc nhiều di chuyển. Xuất phát từ một trạng thái S1, con khỉ có thể lấy được quả chuối nếu tồn tại một số lần di chuyển M nào đó từ S1 đến một trạng thái S2 sao cho trong trạng thái S2, con khỉ có thể lấy được quả chuối.

Ta có mệnh đề sau :

```
couldtake(S1) :-
    displacement(S1, M, S2),
    couldtake(S2).
```

Vị từ `couldtake` có dạng đệ quy, tương tự với quan hệ `ancestor` đã xét ở đầu chương.

Chương trình Prolog đầy đủ như sau :

```
displacement(
    state(tothecenter, onthebox, tothecenter, nothave),
    grab,                                   % với lấy quả chuối
    state(tothecenter, onthebox, tothecenter,
    possessing)).
displacement(
    state(P, onthefloor, P, H),
    climbing,                               % trèo lên hộp
    state(P, onthebox, P, H)).
displacement(
    state(P1, onthefloor, P1, H),
    pushing(P1, P2),          % đẩy cái hộp từ P1 đến P2
    state(P2, onthefloor, P2, H)).
displacement(
    state(P1, onthefloor, G, H),
    walking(P1, P2),          % di chuyển từ P1 đến P2
    state(P2, onthefloor, G, H)).
pushing(tothewindow, tothecenter).
walking(tothedoor, tothewindow).
% couldtake(state) : con khỉ có thể lấy được quả chuối trong state
couldtake(state(_, _, _, possessing)).      % trường hợp 1 :
con khỉ đã có quả chuối
couldtake(State1) :-          % trường hợp 2 : cần phải hành động
     displacement(State1, Move, State2),    % hành động
     couldtake(State2).                     % lấy quả chuối
```

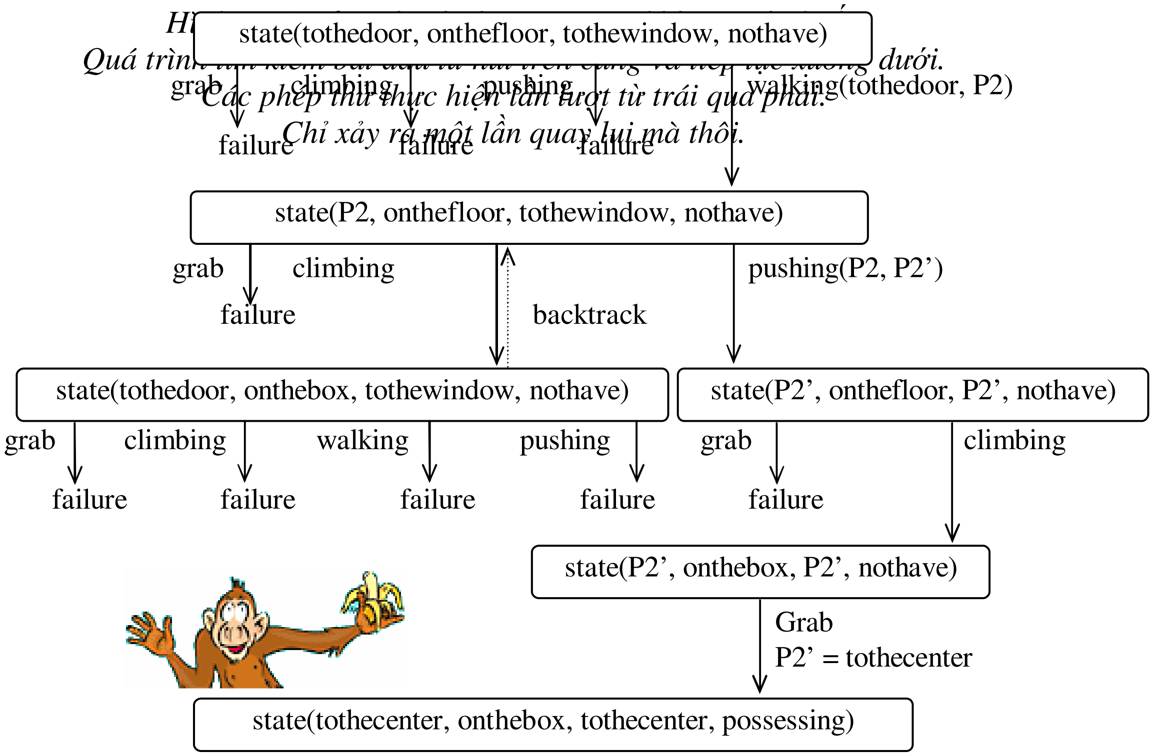
Chương trình trên đây đã được phát triển một cách phi thủ tục. Để xét tính thủ tục của chương trình, ta đặt ra câu hỏi sau đây :

```
?- couldtake(state(tothedoor, onthefloor, tothewindow,
nothave)).
Yes
```

Để có câu trả lời, Prolog phải thoả mãn một danh sách các đích theo ngữ nghĩa thủ tục. Đó là quá trình tìm kiếm cho con khỉ một di chuyển hợp lý trong mọi di chuyển có thể. Đôi khi, quá trình này sẽ dẫn đến một ngõ cụt, để thoát ra, cần phải *quay lui*. Câu hỏi trên cần quay lui một lần. Các di chuyển hợp lý tiếp theo được tìm thấy ngay do các mệnh đề liên quan đến quan hệ `displacement` có mặt trong chương trình, phù hợp với tình huống.

Tuy nhiên, vẫn có thể xảy ra khả năng các di chuyển không hợp lý. Con khỉ đi tới đi lui mãi mà không chạm được cái hộp, hoặc không có đích thực sự. Trong ví dụ trên, ta đã ưu tiên quá trình so khớp các mệnh đề để dẫn đến thành công.

Quá trìn… dưới.
Các phép thử thực hiện lần lượt từ trái qua phải.
Chỉ xảy ra một lần quay lui mà thôi.



III.3. Sắp đặt thứ tự các mệnh đề và các đích

III.3.1. Nguy cơ gặp các vòng lặp vô hạn

Xét mệnh đề sau đây :

```
p :- p
```

Nghĩa của mệnh đề là « `p` đúng nếu `p` đúng». Về mặt khai báo, mệnh đề hoàn toàn đúng đắn. Tuy nhiên, về mặt thủ tục, mệnh đề không dùng để làm gì. Trong Prolog, mệnh đề này gây ra rắc rối. Ta xét câu hỏi :

```
?- p.
```

Sử dụng mệnh đề trên, đích `p` được thay thế bởi chính đích `p`, rồi lại được thay thế bởi p, và cứ thế tiếp tục. Prolog bị rơi vào tình trạng quẩn vô hạn.

Ví dụ này làm phương tiện thực hiện các vòng lặp của Prolog. Trở lại ví dụ con khỉ và quả chuối trên đây, ta có thể thay đổi thứ tự các đích bên trong của các mệnh đề. Chẳng hạn các mệnh đề thuộc về quan hệ `displacement` đã được sắp xếp như sau :

```
grab, climbing, pushing, walking
```

(ta có thể bổ sung thêm mệnh đề `descending` nếu muốn trọn vẹn).

Các mệnh đề này nói rằng con khỉ có thể nắm lấy quả chuối (grab), trèo lên hộp (climbing), v.v... Về mặt ngữ nghĩa thủ tục, thứ tự các mệnh đề nói rằng trước con khỉ với lấy được quả chuối, nó phải trèo lên hộp, trước khi trèo lên hộp, nó phải đẩy cái hộp, v.v... Với thứ tự này, con khỉ lấy được quả chuối (giải quyết được bài toán). Bây giờ nếu ta thay đổi thứ tự thì điều gì sẽ xảy ra ? Giả thiết rằng mệnh đề `walking` xuất hiện đầu tiên. Lúc này, việc thực hiện đích đã đặt ra trên đây :

```
?- couldtake(state(tothedoor, onthefloor, tothewindow,
nothave)).
```

sẽ tạo ra một quá trình thực thi khác.

Bốn danh sách đích đầu tiên như cũ (các tên biến được đặt lại) :

```
(1)    couldtake(state(tothedoor,          onthefloor,
   tothewindow, nothave))
```

Sau khi mệnh đề thứ hai được áp dụng, ta có :

```
(2)    displacement(state(tothedoor, onthefloor,
   tothewindow, nothave), M', S2'),
   couldtake(S2')
```

Với chuyển động `walking(tothedoor, P2')`, ta nhận được :

```
(3)    couldtake(state(P2’,   onthefloor,   tothewindow,
   nothave))
```

Áp dụng lần nữa mệnh đề thứ hai của `couldtake` :

```
(4)    displacement(state(P2’, onthefloor, tothewindow,
   nothave), M’’, S2’’), couldtake(S2’’)
```

Từ thời điểm này, sự khác nhau xuất hiện. Mệnh đề đầu tiên có phần đầu có thể so khớp với đích đầu tiên trên đây bây giờ sẽ là `walking` (mà không phải `climbing` như trước).

Ràng buộc là `S2’’ = state(P2’’, onthefloor, tothewindow, nothave)`. Danh sách các đích trở thành :

```
(5)    couldtake(state(P2’’, onthefloor, tothewindow,
   nothave))
```

Bằng cách áp dụng mệnh đề thứ hai `couldtake`, ta nhận được

```
(6)    displacement(state(P2’’, onthefloor, tothewindow,
   nothave), M’’’, S2’’’), couldtake(S2’’’)
```

Tiếp tục áp dụng mệnh đề `walking` cho mệnh đề thứ nhất và ta có :

```
(7)    couldtake(state(P2’’’, onthefloor, tothewindow,
   nothave))
```

Bây giờ ta so sánh các đích (3), (5) và (7). Chúng gần như giống hệt nhau, trừ các biến `P2’`, `P2’’` và `P2’’’`. Như ta đã thấy, sự thành công của một đích không phụ thuộc vào tên các biến trong đích. Điều này có nghĩa rằng kể từ danh sách các đích (3), quá trình thực hiện không có sự tiến triển nào.

Thực tế, ta nhận thấy rằng mệnh đề thứ hai của `couldtake` và `walking` đã được sử dụng qua lại. Con khỉ đi loanh quanh trong phòng mà không bao giờ có ý định sử dụng cái hộp. Do không có sự tiến triển nào, nên về mặt lý thuyết, quá trình tìm đến quả chuối sẽ diễn ra một cách vô hạn. Prolog sẽ không xử lý những tình huống vô ích như vậy.

Ví dụ này minh hoạ Prolog đang thử giải một bài toán mà không bao giờ đạt được lời giải, dẫu rằng lời giải tồn tại. Những tình huống như vậy không phải là hiếm khi lập trình Prolog. Người ta cũng hay gặp những vòng lặp quẩn vô hạn trong các ngôn ngữ lập trình khác. Tuy nhiên, điều không bình thường so với các ngôn ngữ lập trình khác là chương trình Prolog đúng đắn về mặt ngữ nghĩa khai báo, nhưng lại không đúng đắn về mặt thủ tục, nghĩa là không có câu trả lời đối với câu hỏi cho trước.

Trong những trường hợp như vậy, Prolog không thể xoá một đích vì Prolog cố gắng đưa ra một câu trả lời trong khi đang đi theo một con đường xấu (không dẫn đến thành công).

Câu hỏi chúng ta muốn đặt ra là : liệu chúng ta có thể thay đổi chương trình sao cho có thể dự phòng trước nguy cơ bị quẩn ? Có phải chúng ta luôn luôn bị phụ thuộc vào sự sắp đặt thứ tự đúng đắn của các mệnh đề và các đích ? Rõ ràng rằng các chương trình lớn sẽ trở nên dễ sai sót nếu phải dựa trên một thứ tự nào đó của các mệnh đề và các đích. Tồn tại nhiều phương pháp khác cho phép loại bỏ các vòng lặp vô hạn, tổng quát hơn và đáng tin cậy hơn so với phương pháp sắp đặt thứ tự. Sau đây, chúng ta sẽ sử dụng thường xuyên những phương pháp này trong việc tìm kiếm các con đường, hợp giải các bài toán và duyệt các đồ thị.

III.3.2. Thay đổi thứ tự mệnh đề và đích trong chương trình

Ngay các ví dụ ở đầu chương, ta đã thấy nguy cơ xảy ra các vòng lặp vô hạn. Chương trình mô tả quan hệ tổ tiên :

```
ancestor(X, Z) :-
   parent(X, Z).

ancestor(X, Z) :-
   parent(X, Y),
   ancestor(Y, Z).
```

Ta hãy xét một số biến thể của chương trình này. Về mặt khai báo, tất cả các chương trình là tương đương, nhưng về mặt thủ tục, chúng sẽ khác nhau. Tham khảo ngữ nghĩa khai báo của Prolog, không ảnh hưởng đến nghĩa khai báo, ta có thể thay đổi như sau :

(1) Thứ tự các mệnh đề trong một chương trình, và

(2) Thứ tự các đích bên trong thân của các mệnh đề.

Thủ tục `ancestor` trên đây gồm hai mệnh đề, đuôi mệnh đề thứ nhất có một đích con và đuôi mệnh đề thứ hai có hai đích con. Như vậy chương trình sẽ có bốn biến thể (=1×2×2) mà cả bốn đều có cùng nghĩa khai báo. Ta nhận được như sau :

(1) Đảo thứ tự các mệnh đề, và

(2) Đảo thứ tự các đích cho mỗi sắp đặt thứ tự các mệnh đề.

Hình dưới đây mô tả bốn thủ tục `anc1`, `anc2`, `anc3`, `anc4` :

```
% Thủ tục gốc
anc1(X, Z) :-
   parent(X, Z).
anc1 (X, Z) :-
   parent(X, Y),
   anc1 (Y, Z).

% Biến thể a : hoán đổi các mệnh đề
```

```
anc2 (X, Z) :-
   parent(X, Y),
   anc2 (Y, Z).
anc2(X, Z) :-
   parent(X, Z).
```

% Biến thể b : hoán đổi các đích của mệnh đề thứ hai

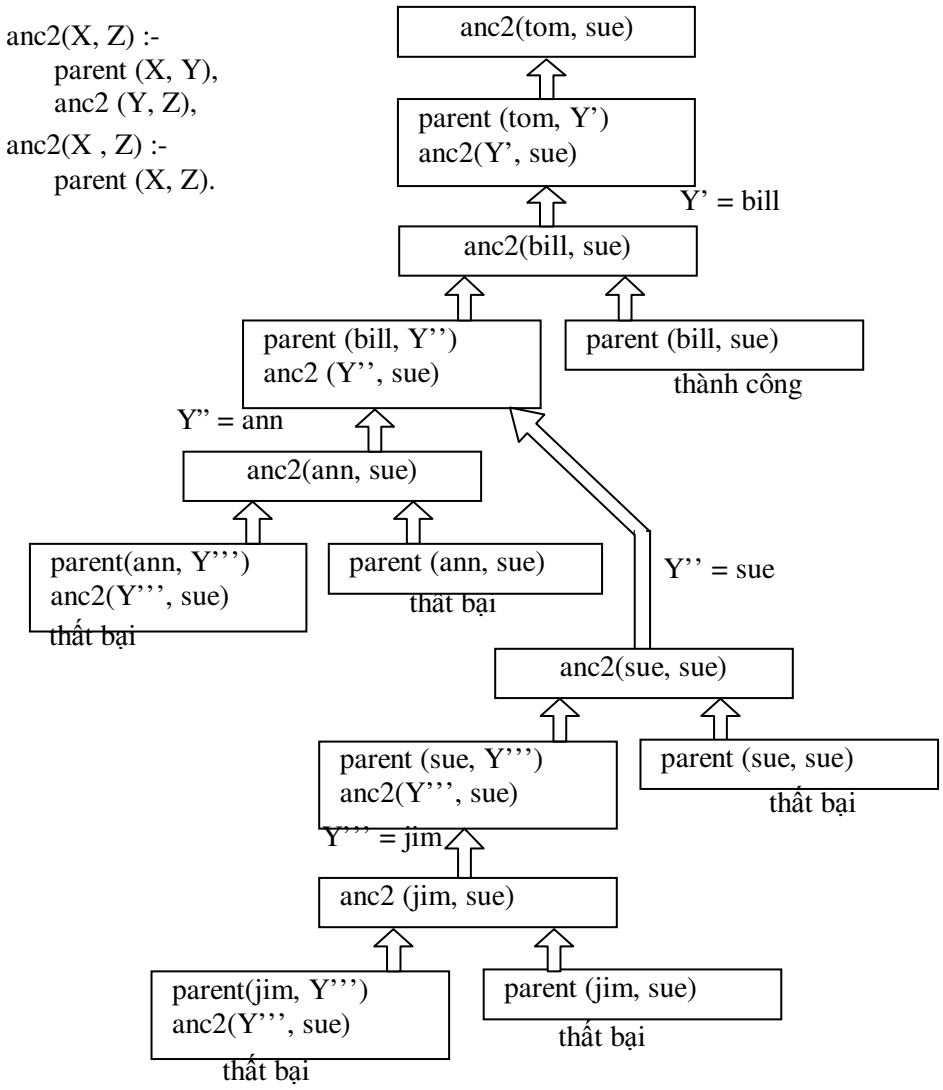
```
anc3(X, Z) :-
   parent(X, Z).
anc3 (X, Z) :-
   anc3 (X, Y),
   parent(Y, Z).
```

% Biến thể c : hoán đổi các đích và các mệnh đề

```
anc4 (X, Z) :-
   anc4 (X, Y),
   parent(Y, Z).
anc4(X, Z) :-
   parent(X, Z).
```

% Các câu hỏi được đặt ra lần lượt như sau :

```
?- anc1(tom, sue).
 -> Yes
?- anc2(tom, sue).
 -> Yes
?- anc3(tom, sue).
 -> Yes
?- anc4(tom, sue).
ERR 211 Not enough local stack
```



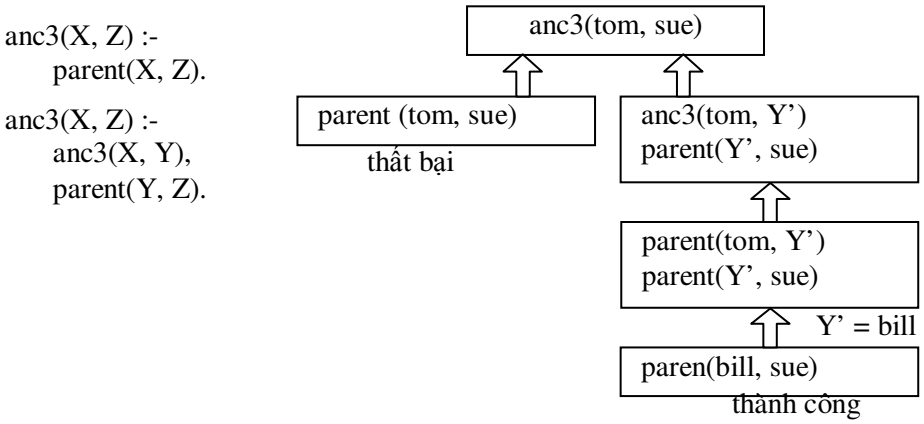
Hình III.6. Biến thể a của quan hệ tổ tiên trả lời câu hỏi "Tom có phải là một tổ tiên của Sue ?"

Trong trường hợp cuối cùng, Prolog không thể tìm ra câu trả lời. Do bị quẩn vô hạn nên Prolog thông báo "không đủ bộ nhớ". Hình 2.4. mô tả quá trình thực hiện của `anc1` (trước đây là `ancestor`) cho cùng một câu hỏi.

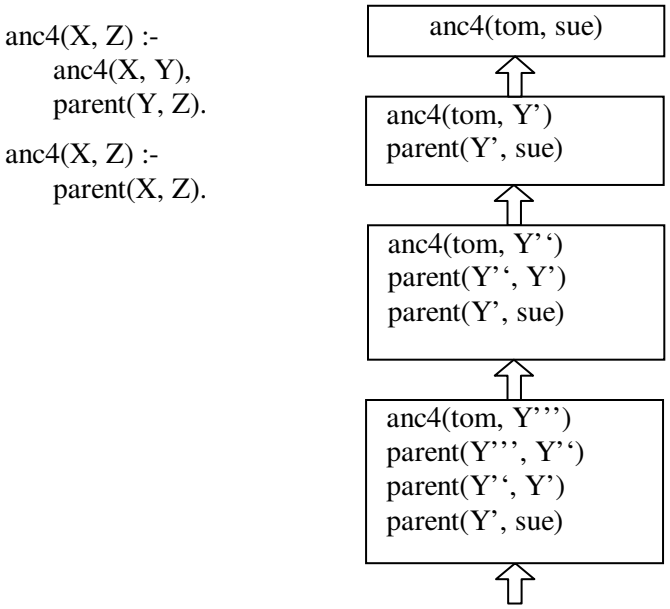
Hình 2.13 (a, b, c) mô tả quá trình thực hiện của `anc2`, `anc3` và `anc4`. Ta thấy `anc4` không có hy vọng và `anc2` kém hiệu quả hơn so với `anc1` do thực hiện nhiều lần tìm kiếm và quay lui hơn trong cây.

So sánh các quá trình so khớp trong các hình vẽ, ta thấy rằng cần khai báo các mệnh đề đủ đơn giản khi giải các bài toán. Đối với bài toán quan hệ tổ tiên, cả bốn biến thể đều dựa trên hai ý :

- kiểm tra nếu hai tham đối của quan hệ tổ tiên thoả mãn quan hệ `parent`.
- giai đoạn phức tạp nhất là tìm ai đó "giữa" những người là `parent` hay `ancestor`.



Hình III.7. Biến thể b của quan hệ tổ tiên trả lời câu hỏi "Tom có phải là một tổ tiên của Sue ?"



Hình III.8. Biến thể c của quan hệ tổ tiên trả lời câu hỏi "Tom có phải là một tổ tiên của Sue ?"

Trong số bốn biến thể của quan hệ `ancestor`, chỉ có `anc1` là thực hiện quá trình so khớp đơn giản nhất. Trong khi đó, `anc4` bắt đầu quá trình khó khăn nhất. Còn `anc2` và `anc3` nằm giữa hai thái cực này. Dù ta có xem xét chi tiết các quá trình thực hiện thế nào đi chăng nữa, thì `anc1` vẫn là luật đơn giản nhất. Người ta khuyên nên sử dụng cách này khi lập trình.

Ta không cần so sánh bốn biến thể mà xem xét với kiểu câu hỏi nào thì mỗi biến thể dẫn đến thành công hay thất bại. Ta dễ nhận thấy rằng cả hai thủ tục `anc1` và `anc2` đều có khả năng đưa ra câu trả lời cho mọi kiểu câu hỏi. Còn `anc3` thì không chắc chắn. Chẳng hạn câu hỏi sau đây gây ra thất bại :

```
anc3(liz, jim)
ERR 212 Not enough global stack
```

vì dẫn đến những lời gọi đệ quy vô hạn. Như vậy, ta không thể xem `anc3` là đúng đắn về mặt thủ tục.

Tóm tắt chương 2

- Những khái niệm đã được giới thiệu :
  đích thoả mãn, thành công
  đích không thoả mãn/bị thất bại,
  đích bị xoá,
  nghĩa khai báo, nghĩa thủ tục, nghĩa lôgich,
  quay lui.
  thể nghiệm của một mệnh đề, biến thể của một mệnh đề
- Lập trình trên ngôn ngữ Prolog là *định nghĩa các quan hệ* và *đặt câu hỏi* trên các quan hệ này.
- Một chương trình Prolog bao gồm các *mệnh đề*. Có ba kiểu mệnh đề : *sự kiện*, *luật* và *câu hỏi*.
- Một quan hệ có thể được đặc tả bởi các sự kiện, bằng cách ghi nhận bộ–n đối tượng thoả mãn quan hệ, hay thiết lập các luật liên quan đến quan hệ.
- Một thủ tục là một tập hợp các mệnh đề liên quan đến cùng một quan hệ.
- Việc đặt các câu hỏi trên các quan hệ tương tự việc vấn tin một cơ sở dữ liệu. Prolog trả lời câu hỏi bằng cách liệt kê tập hợp các đối tượng làm thoả mãn câu hỏi này.
- Trong Prolog, khi một đối tượng làm thoả mãn một câu hỏi thì việc trả lời câu hỏi luôn luôn là một quá trình phức tạp sử dụng suy diễn lôgich, khai thác các khả năng khác nhau, và cơ chế quay lui. Prolog tiến hành tự động quá trình này và về nguyên tắc, NSD có thể hiểu được.
- Người ta phân biệt nghĩa khai báo và nghĩa thủ tục khi lập trình. Một chương trình Prolog thường có nghĩa khai báo là chủ yếu. Tuy nhiên, người ta vẫn tìm thấy nghĩa thủ tục trong một số chương trình Prolog.
- Theo nghĩa thủ tục, chương trình Prolog thực hiện quá trình tìm kiếm, so khớp và quay lui.

- *Ngữ nghĩa khai báo* của Prolog xác định nếu một đích là đúng đối với một chương trình đã cho, tương ứng với ràng buộc của các biến.
- Người ta quy ước viết *phép giao* (and) của hai đích bằng cách đặt một dấu phẩy ở giữa chúng, *phép hoặc* (or) bởi một dấu chấm phẩy.
- *Ngữ nghĩa thủ tục* của Prolog được thể hiện bởi một thủ tục tìm kiếm làm thoả mãn một danh sách các đích từ một chương trình đã cho. Nếu tìm kiếm thoả mãn, Prolog trả về các ràng buộc các biến tương ứng. Nếu tại một bước nào đó bị thất bại, thủ tục này cho phép tự động *quay lui* (backtracking) để tìm kiếm tìm các khả năng khác có thể dẫn đến thành công.
- Nghĩa khai báo của các chương trình thuần Prolog không phụ thuộc sự sắp đặt các mệnh đề, cũng như không phụ thuộc sự sắp đặt các đích bên trong các mệnh đề.
- Nghĩa thủ tục phụ thuộc thứ tự các đích và các mệnh đề. Thứ tự sắp đặt này có thể ảnh hưởng đến tính hiệu quả chạy chương trình, và có thể dẫn đến những lời gọi đệ quy vô hạn.
- Cho trước một khai báo đúng, có khả năng làm tối ưu hiệu quả vận hành của hệ thống bằng cách thay đổi thứ tự các mệnh đề, mà vẫn đảm bảo tính đúng đắn về mặt khai báo. Sự sắp đặt lại thứ tự các đích và các mệnh đề là một trong những phương pháp nhằm tránh các vòng lặp quẩn vô hạn.
- Còn có những kỹ thuật khác tổng quát hơn để tránh các vòng lặp quẩn vô hạn, và làm cho chương trình vận hành đáng tin cậy hơn.

Bài tập chương 2

1. Từ chương trình Prolog dưới đây:

```
aeroplane(concorde).
aeroplane(jumbo).
on(fred, concorde).
on(jim, No_18_bus).
bird(percy).
animal(leo).
animal(tweety).
animal(peter).
has_feathers(tweety).
has_feathers(peter).

flies(X) :- bird(X).
flies(X) :- aeroplane(X).
flies(X) :- on(X, Y), aeroplane(Y).
```

```
bird(X) :- animal(X), has_feathers(X).
```

Hãy cho biết các kết quả nhận được từ câu hỏi :

```
?- flies(X).
```

bằng cách liệt kê theo thứ tự :

```
X=kq1;
X=kq2; v.v...
```

2. Sử dụng sơ đồ đã cho trong phần lý thuyết, hãy tìm hiểu cách Prolog tìm ra các câu trả lời đối với các câu hỏi dưới đây. Vẽ sơ đồ minh họa tương ứng theo kiểu sơ đồ đã cho. Có khả năng Prolog quay lui không ?

   a) `?- parent(mary , bill).`

   b) `?- mother(mary , bill).`

   c) ?- grand parent(mary, ann).

   d) ?- grand parent(bill , jim).

3. Viết lại chương trình dưới đây, nhưng không sử dụng dấu chấm hỏi :

```
translate (Number, word) :-
     Number = 1, Word = one;
     Number = 2, Word = two;
     Number = 3, Word = three.
```

4. Từ chương trình `execute` trong lí thuyết, hãy vẽ sơ đồ quá trình thực hiện của Prolog từ câu hỏi sau :

```
?- thick( X ) , dack( X ).
```

   Hãy so sánh các quá trình mô phỏng câu hỏi trên và các đích dưới đây :

```
?- dack( X ), thick( X ).
```

5. Điều gì xảy ra khi yêu cầu Prolog trả lời câu hỏi sau đây :

```
?- X = f( X ).
```

   So khớp có thành công không ?
   Giải thích vì sao một số hệ thống Prolog trả lời :

```
X = f(f(f(f(f(f(f(f(f( ... ))))))))))
Yes
```

6. Tìm các phép thay thế hợp thức và tìm kết quả (nếu có) của các phép so khớp sau đây :

```
a(1, 2) = a(X, X).
a(X, 3) = a(4, Y).
a(a(3, X)) = a(Y).
```

```
1+2 = 3.

X = 1+2.

a(X, Y) = a(1, X).

a(X, 2) = a(1, X).
```

7. Cho trước chương trình dưới đây :

```
f( 1, one ).
f( s(1) , two ).
f( s( s( 1 ) ) , three ).
f( s( s( s( X ) ) ), N ) :-
        f(X , N+3).
```

Hãy cho biết cách Prolog trả lời các câu hỏi sau đây (khi có nhiều câu trả lời có thể, hãy đưa ra ít nhất hai câu trả lời) ?

```
a) ?- f( s( 1 ) , A ).

b) ?- f( s( s( 1 ) ) , two ).

c) ?- f( s( s( s( s( s( 1 ) ) ) ) ), C ).

d) ?- f( D, three ).
```

8. Cho các vị từ `p`, `a1`, `a2`, `a3`, `a4` được định nghĩa bởi các mệnh đề sau đây :

```
p(a, b).

p(b, c b).

a1(X, Y) :- p(X, Y).

a1(X, Y) :- p(X, Z), a1(Z, Y).

a2(X, Y) :- p(X, Y).

a2(X, Y) :- a2(Z, Y), p(X, Z).

a3(X, Y) :- p(X, Z), a3(Z, Y).

a3(X, Y) :- p(X, Y).

a4(X, Y) :- a4(Z, Y), p(X, Z).

a4(X, Y) :- p(X, Y).
```

   a) Vẽ cây hợp giải SLD có các gốc là `a1(a, X)`, `a2(a, X)`, `a3(a, X)`, `a4(a, X)` ?

   b) So sánh nghĩa lôgich của các vị từ `a1`, `a2`, `a3`, `a4` ?

9. Viết gói các mệnh đề định nghĩa các hàm sau :

   a) `greathan(X, N)` trả về giá trị `X` nếu `X > N`, trả về `N` nếu không phải.

b) `sum_diff(X, Y, X)` trả về trong `Z` giá trị tổng `X + Y` nếu `X > Y`, trả về hiệu `X - Y` nếu không phải.

10. Viết chương trình Prolog từ biểu diễn lôgich sau đây :

$\forall X$: *pet(X)* $\wedge$ *small(X)* $\rightarrow$ *apartmentpet(X)*
$\forall X$: *cat(X)* $\vee$ *dog(X)* $\rightarrow$ *pet(X)*
$\forall X$: *poodle(X)* $\rightarrow$ *dog(X)* $\wedge$ *small(X)*
poodle(fluffy)

Tự đặt câu hỏi Prolog và vẽ sơ đồ quá trình thực hiện.

CHƯƠNG 3

Các phép toán và số học

Chương này trình bày số học sơ cấp, các phép toán và một số vị từ chuẩn được sử dụng trong các chương trình Prolog.

I. Số học

I.1. Các phép toán số học

Như đã biết, Prolog là ngôn ngữ chủ yếu dùng để xử lý ký hiệu, không thích hợp để tính toán số. Do vậy, các phương tiện tính toán trong hầu hết các hệ thống Prolog đều rất hạn chế. Sau đây là bảng các phép toán số học chuẩn (standard arithmetic operations) của Prolog :

Ký hiệu	*Phép toán*
`+`	Cộng (addition)
`-`	Trừ (subtraction)
`*`	Nhân (multiplication)
`/`	Chia số thực (real division)
`//`	Chia số nguyên (integer division)
`mod`	Chia lấy phần dư (modulus)
`**`	Luỹ thừa (power)

I.2. Biểu thức số học

Biểu thức số học (arithmetic expressions) được xây dựng nhờ vị từ **`is`**. Vị từ này là một *phép toán tiền tố* (infix operator) có dạng :

```
Number is Expr
```

Tham đối bên trái phép toán `is` là một đối tượng sơ cấp. Tham đối bên phải là một biểu thức số học được hợp thành từ các phép toán số học, các số và các biến. Vì phép toán `is` sẽ khởi động việc tính toán, cho nên khi thực hiện đích

này, tất cả các biến cần phải được ràng buộc với các giá trị số. Prolog so khớp thành công nếu `Number` khớp được với `Expr`. Nếu `Expr` là kiểu thực (float) thì được xem như một số nguyên.

Ví dụ I.1 :

```
?- X is 3*4.
X = 12
Yes
?- is(X, 40+50).
X = 90
Yes
?- 1.0 is sin(pi/2).
No                % sai do sin(pi/2) được làm tròn thành 1
?- 1.0 is float(sin(pi/2)).
Yes
```

Trong Prolog, các phép toán số học kéo theo sự tính toán trên các dữ liệu. Để thực hiện các phép toán số học, cần biết cách gọi dùng theo *kiểu Prolog* mà không thể gọi trực tiếp ngay được như trong các ngôn ngữ lập trình mệnh lệnh.

Chẳng hạn, nếu NSD cần cộng hai số 1 và 2 mà lại viết như sau :

```
?- X = 1 + 2
```

thì Prolog sẽ trả lời theo kiểu của Prolog :

```
X = 1 + 2
```

mà không phải là `X = 3` như mong muốn. Lý do rất đơn giản : biểu thức `X = 1 + 2` chỉ là một hạng của Prolog mà hàm tử chính là `+`, còn `1` và `2` là các tham đối của nó. Không có gì trong đích trước nó để Prolog tiến hành phép cộng. Sau đây là một số ví dụ :

```
?- X = 1 + 1 + 1.
X = 1 + 1 + 1 (ou X = +(+(1, 1), 1)).
```

Để Prolog tiến hành tính toán trên các phép toán số học, sử dụng phép toán `is` như sau :

```
?- X is 1 + 2.
X = 3
```

Phép cộng thực hiện được là nhờ một thủ tục đặc biệt kết hợp với phép toán `+`. Những thủ tục như vậy được gọi là *thủ tục thường trú* (built-in procedures).

```
?- X = 1 + 1 + 1, Y is X.
X = 1 + 1 + 1, Y = 3.

?- X is 1 + 1 + a.
```

```
ERROR: Arithmetic: `a/0' is not a function
```
(sai do `a` không phải là hàm số)

```
?- X is 1 + 1 + Z.
ERROR: Arguments are not sufficiently instantiated
```
(sai do `a` không phải là số)

```
?- Z = 2, X is 1 + 1 + Z.
Z = 2
X = 4
```

Độ ưu tiên của các phép toán số học tiền định của Prolog cũng là độ ưu tiên thoả mãn tính chất kết hợp trong toán học. Các cặp dấu ngoặc có thể làm thay đổi thứ tự độ ưu tiên giữa các phép toán. Chú ý rằng `+`, `-`, `*`, `/` và `//` được định nghĩa như là `yfx`, có nghĩa là việc tính toán được thực hiện từ trái sang phải. Ví dụ, biểu thức :

```
X is 5 -2 - 1
```
được giải thích như là :
```
X is ( 5 -2 ) - 1
```
Do đó :
```
?- X is 5 -2 - 1.
X = 2
Yes
?- X = 5 -2 - 1.
X = 5-2-1
Yes
```

Các phép *so sánh* giá trị số học trong Prolog được thực hiện theo nghĩa Toán học thông thường. Chẳng hạn, ta cần so sánh nếu tích của 277 với 37 là lớn hơn 10000 với đích sau :

```
?- 277 * 37 > 10000.
Yes
```

Bây giờ giả sử ta có quan hệ `birth`, cho phép liên hệ một người với ngày tháng năm sinh của người đó. Ta có thể tìm được tên của những người sinh ra giữa năm 1950 và năm 1960 (kể cả hai năm này) bằng cách đặt câu hỏi :

```
?- birth( Name, Year ), Year >= 1950, Year <= 1960.
% kết quả trả về là tên những người sinh ra trong khoảng 1950 - 1960
Yes
```

Prolog có sẵn các hàm số học như : `sin`, `cos`, `tan`, `atan`, `sqrt`, `pi`, `e`, `exp`, `log`, ...

Ví dụ I.2 :

```
?- X is exp(10).
X = 22026.5
Yes
?- X is sqrt(9).
X = 3
Yes
7 ?- X is abs(1.99).
X = 1.99
Yes
?- X is pi.
X = 3.14159
Yes
```

I.3. Định nghĩa các phép toán trong Prolog

Biểu thức toán học thường được viết dưới dạng *trung tố* (infix) như sau :

```
2 * a + b * c
```

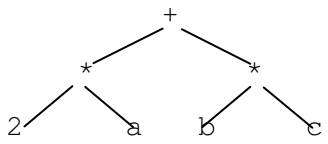
với `+` và `*` là các *phép toán* (operator), còn `a`, `b` và `c` là các *toán hạng* (operand), hay tham đối (argument). Biểu thức trên còn được viết dưới dạng *tiền tố* (prefix) nhờ các hàm tử `+` và `*` như sau :

```
+( *(2, a), *(b, c) )
```

hoặc dạng *hậu tố* (postfix) như sau :

```
( (2, a) *, (b, c) * )+
```

Do thói quen, người ta thích viết các biểu thức ở dạng trung tố để dễ đọc hơn. Prolog cho phép viết các biểu thức dưới dạng trung tố, là biểu diễn bên ngoài, nhưng thực chất, các biểu thức được biểu diễn bên trong vẫn ở dạng tiền tố, theo quy ước viết các hạng trong một mệnh đề.



Hình I.1. Biểu diễn dạng cây của biểu thức `2 * a + b * c`

Khi viết `a + b`, Prolog hiểu rằng đó là biểu thức `+(a, b)`. Để Prolog có thể hiểu được đúng đắn các biểu thức như là `a + b * c`, cần cho Prolog biết rằng phép nhân `*` có ưu tiên cao hơn phép cộng `+`. Khi đó biểu thức này phải được viết dưới dạng :

```
+( a, *(b, c) )
```

mà không phải là :

```
*( + (a, b), c)
```

Prolog quy ước phép toán có độ ưu tiên cao nhất là hàm tử chính của hạng. Nếu các biểu thức chứa + và * tuân theo những quy ước thông thường, thì cách viết `a + b * c` và `a + (b * c)` chỉ là một. Còn nếu muốn thay đổi thứ tự ưu tiên, thì cần viết rõ ràng bằng cách sử dụng các cặp dấu ngoặc `(a + b) * c` :

Mỗi NLT có thể định nghĩa các phép toán riêng của mình, chẳng hạn định nghĩa các nguyên tử `is` và `support` như là những phép toán trung tố để viết các sự kiện trong một chương trình. Chẳng hạn :

```
tom bald
wall support ceiling
```

là những sự kiện được viết trong Prolog :

```
is( tom, bald ).
support( wall, ceiling ).
```

Mỗi phép toán là một nguyên tử có độ ưu tiên là một giá trị số, tuỳ thuộc phiên bản Prolog, thông thường nằm trong khoảng giữa 1 và 1200. Các phép toán được đặc tả bởi hỗn hợp tên phép toán `f` và các biến (tham đối) `x` và `y`. Mỗi đặc tả cho biết cách kết hợp (associative) phép toán đó và được chọn sao cho phản ánh được cấu trúc của biểu thức. Một phép toán trung tố được ký hiệu bởi một `f` đặt giữa hai tham đối dạng `xfy`. Còn các phép toán tiền tố và hậu tố chỉ có một tham đối được đặt trước (hoặc đặt sau tương ứng) dấu phép toán `f`.

Có ba nhóm kiểu phép toán trong Prolog như sau :

Các phép toán	*Không kết hợp*	*Kết hợp phải*	*Kết hợp trái*
Trung tố	*xfx*	*xfy*	*yfx*
Tiền tố	*fx*	*fy*	
Hậu tố	*xf*		*yf*

Hình I.2. Ba nhóm kiểu phép toán trong Prolog.

Có sự khác nhau giữa `x` và `y`. Để giải thích, ta đưa vào khái niệm *độ ưu tiên của tham đối*. Nếu các dấu ngoặc bao quanh một tham đối, hay tham đối này là một đối tượng không có cấu trúc, thì độ ưu tiên của nó bằng 0.

Độ ưu tiên của một cấu trúc là độ ưu tiên của hàm tử chính. Do `x` là một tham đối nên độ ưu tiên của `x` phải thấp hơn hẳn độ ưu tiên của phép toán `f`, còn tham đối `y` có độ ưu tiên thấp hơn hoặc bằng độ ưu tiên của phép toán `f`.

Khi gặp một biểu thức chứa phép toán `op` dạng :

```
a op b op c
```

Tính kết hợp xác định vị trí dấu ngoặc theo thứ tự ưu tiên như sau :

- Nếu là kết hợp trái, ta có : **(**`a op b`**)** `op c`

- Nếu là kết hợp phải, ta có : `a op (b op c)`

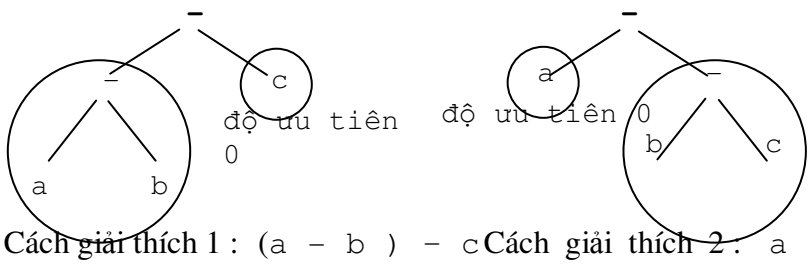
Các quy tắc trên cho phép loại bỏ tính nhập nhằng của các biểu thức chứa các phép toán có cùng độ ưu tiên. Chẳng hạn :

```
a – b – c
```

sẽ được hiểu là `(a – b ) – c`, mà không phải `a – (b – c)`. Ở đây, phép trừ «–» có kiểu trung tố `yfx`. Xem Hình I.3 dưới đây.

Bây giờ ta lấy một ví dụ khác về phép toán tiền tố một ngôi `not`. Nếu `not` được xếp kiểu `fy`, thì biểu thức sau đây viết đúng :

```
not not p
```



Hình I.3. Hai cách giải thích cho biểu thức `a - b - c` *với giả thiết rằng phép trừ «-» có độ ưu tiên là 500. Nếu «-» là* `yfx`*, thì cách giải thích 2 là sai vì độ ưu tiên của* `b - c` *không thấp hơn độ ưu tiên của «-».*

Trái lại, nếu phép toán `not` được định nghĩa như là `fx`, thì biểu thức trên sẽ không còn đúng nữa, vì rằng tham đối của `not` đầu tiên là `not p`, sẽ có cùng độ ưu tiên với nó. Trong trường hợp này, biểu thức phải được viết kết hợp với các cặp dấu ngoặc :

```
not ( not p )
```

Tính dễ đọc của một chương trình tuỳ thuộc vào cách sử dụng các phép toán. Trong các chương trình Prolog, những mệnh đề sử dụng phép toán mới do người dùng định nghĩa thường được gọi là các *chỉ dẫn* hay *định hướng* (directive). Các chỉ dẫn phải xuất hiện trước khi một phép toán mới được sử dụng đến trong một mệnh đề, có dạng như sau :

`:- op(`*Độ ưu tiên*`,` *Cách kết hợp*`,` *Tên phép toán*`).`

Chẳng hạn người ta định nghĩa phép toán `is` nhờ chỉ dẫn :

```
:- op( 600, xfx, is ).
```

Chỉ dẫn này báo cho Prolog biết rằng `is` sẽ được sử dụng như là một phép toán có độ ưu tiên là `600`, còn ký hiệu đặc tả `xfx` chỉ định đây là một phép toán trung tố. Ý nghĩa của `xfx` như sau : `f` là dấu phép toán được đặt ở giữa, còn `x` là tham đối được đặt hai bên dấu phép toán.

Việc định nghĩa một phép toán không kéo theo một hành động (action) hoặc một thao tác (opration) nào. Về nguyên lý, *không một thao tác nào trên dữ liệu được kết hợp với một phép toán* (trừ một vài trường hợp hiếm gặp đặc biệt, như các phép toán số học). Tương tự như mọi hàm tử, các phép toán chỉ được dùng để cấu trúc các hàm tử, mà không kéo theo một thao tác nào trên các dữ liệu, dẫu rằng tên gọi «phép toán» có thể gợi lên vai trò hoạt động.

Prolog cung cấp sẵn một số phép toán chuẩn. Những phép toán tiền định nghĩa này thay đổi tùy theo phiên bản Prolog. Hình 3.5 dưới đây trình bày một số phép toán chuẩn tiền định nghĩa của Prolog. Chú ý rằng cùng một mệnh đề có thể

định nghĩa nhiều phép toán, miễn là chúng cùng kiểu và cùng độ ưu tiên. Các tên phép toán được viết trong một danh sách.

Các phép toán tiền định nghĩa trong Prolog như sau :

Độ ưu tiên	*Cách kết hợp*	*Các phép toán*
1200	*xfx*	-->, :-
1200	*fx*	:-, ?-
1100	*xfy*	;, |
1000	*xfy*	,
900	*fy*	\+
900	*fx*	~
700	*xfx*	<, =, =.., =@=, =:=, =<, ==, =\=, >, >=, @<, @=<, @>, @>=, \=, \==, is
600	xfy	:
500	*yfx*	+, -, /\, \/, xor
500	*fx*	+, -, ?, \
400	*yfx*	*, /, //, <<, >>, mod, rem
200	*xfx*	**
200	*xfy*	^

Hình 3.5. Các phép toán tiền định nghĩa trong Prolog.

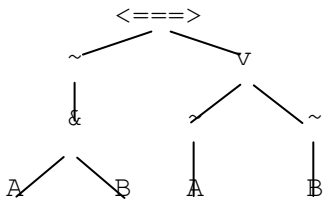
Để minh hoạ, ta xét ví dụ viết một chương trình xử lý các biểu thức lôgich (boolean). Giả sử ta cần biểu diễn một trong các định lý tương đương của Morgan, được viết dưới dạng toán học như sau :

~ (A & G) <===> ~ A ∨ ~B

Trong Prolog, mệnh đề trên phải được viết như sau :

```
equivalent( not ( and( A, B ) ), or(not ( A ), not ( B ) ) ).
```

Tuy nhiên, cách lập trình tốt nhất là thử tìm cách bảo lưu tối đa sự giống nhau giữa các ký hiệu trong bài toán đã cho với các ký hiệu được sử dụng trong chương trình..



Hình I.4. Biểu diễn cây của hạng `~ ( A & B ) <===> ~ A v ~ B`

Trong ví dụ trên, ta dễ dàng định nghĩa lại các phép toán lôgich như sau :

```
:- op( 800, xfx, <===> ).
:- op( 700, xfy, v ).
:- op( 600, xfy, & ).
:- op( 500, fy, ~ ).
```

Từ đây, định lý Morgan được viết lại thành hạng sau (xem hình trên) :

```
~ ( A & B ) <===> ~ A v ~ B
```

II. Các phép so sánh của Prolog

II.1. Các phép so sánh số học

Prolog có các phép so sánh và hàm số học như sau :

Ký hiệu	*Giải thích phép toán*
`Expr1 > Expr2`	Thành công nếu `Expr1` có giá trị số lớn hơn `Expr2`
`Expr1 < Expr2`	Thành công nếu `Expr1` có giá trị số nhỏ hơn `Expr2`
`Expr1 =< Expr2`	Thành công nếu Expr1 có giá trị số nhỏ hơn hoặc bằng `Expr2`
`Expr1 >= Expr2`	Thành công nếu `Expr1` có giá trị số lớn hơn hoặc bằng `Expr2`
`Expr1 =\= Expr2`	Thành công nếu `Expr1` có giá trị số khác `Expr2`
`Expr1 =:= Expr2`	Thành công nếu `Expr1` có giá trị số bằng `Expr2`
`between(Low, High, Value)`	`Low` và `High` là các số nguyên, `Low=< Value=< High`. `Value` là biến sẽ được nhận giá trị giữa `Low` và `High`
`succ(Int1, Int2)`	Thành công nếu `Int2= Int1+ 1` và `Int1>= 0`
`plus(Int1, Int2, Int3)`	Thành công nếu `Int3= Int1+Int2`

Chú ý rằng các phép toán **=** và **=:=** là hoàn toàn khác nhau, chẳng hạn trong các đích `X = Y` và `X =:= Y` :

- Đích `X = Y` kéo theo việc đồng nhất các đối tượng `X` và `Y`, nếu chúng đồng nhất với nhau thì có thể ràng buộc một số biến nào đó trong `X` và `Y`.
- Đích `X =:= Y` chỉ gây ra một phép tính số học để so sánh mà không xảy phép ràng buộc nào trên các biến.

Ví dụ II.1 :

```
?- X = Y.
X = _G997
Y = _G997
Yes

?- 1 + 2 =:= 2 + 1.
Yes.

?- 1 + 2 = 2 + 1.
No.

?- 1 + 2 = 1 + 2.
Yes.
?- 1 + X = 1 + 2.
X = 2

?- 1 + A = B + 2.

A = 2

B = 1

?- 1 + 2 =:= 2 + 1.
Yes.

?- 1 + X =:= 1 + 2.
ERROR: Arguments are not sufficiently instantiated (sai do
            a không phải là số)
?- 1 + 2 == 1 + 2.
Yes.

?- 1 + 2 == 2 + 1.
No.

?- 1 + X == 1 + 2.
No.

?- 1 + a == 1 + a.
Yes.

1 is sin(pi/2).
Yes
```

```
?- 1.0 is sin(pi/2).
No
?- 1.0 is float(sin(pi/2)).
Yes
?- 1.0 =:= sin(pi/2).
Yes
```

II.2. Các phép so sánh hạng

Các phép so sánh hạng của Prolog như sau :

Ký hiệu	*Giải thích phép toán*
`Term1 == Term2`	Thành công nếu `Term1` tương đương với `Term2`. Một biến chỉ đồng nhất với một biến cùng chia sẻ trong hạng (sharing variable)
`Term1 \== Term2`	Tương đương với `\Term1 == Term2`.
`Term1 = Term2`	Thành công nếu `Term1` khớp được với `Term2`
`Term1 \= Term2`	Tương đương với `\Term1 = Term2`
`Term1 =@= Term2`	Thành công nếu `Term1` có cùng cấu trúc (structurally equal) với `Term2`. Tính có cùng cấu trúc yếu hơn tính tương đương (equivalence), nhưng lại mạnh hơn phép hợp nhất
`Term1 \=@= Term2`	Tương đương với `` `\Term1 =@= Term2' ``
`Term1 @< Term2`	Thành công nếu `Term1` và `Term2` theo thứ tự chuẩn của các hạng
`Term1 @=< Term2`	Thành công nếu hoặc hai hạng bằng nhau hoặc `Term1` đứng trước `Term2` theo thứ tự chuẩn của các hạng
`Term1 @> Term2`	Thành công nếu `Term1` đứng sau `Term2` theo thứ tự chuẩn của các hạng
`Term1 @>= Term2`	Thành công nếu hoặc hai hạng bằng nhau both hoặc `Term1` đứng sau `Term2` theo thứ tự chuẩn của các hạng

`compare(?Order, Hạng1, Hạng2)` Kiểm tra thứ tự <, > hoặc = giữa hai hạng

Ví dụ II.2 :

```
?- free_variables(a(X, b(Y, X), Z), L).
L = [G367, G366, G371]
X = G367
```

```
Y = G366
Z = G371

?- a =@= A.
No

?- a =@= B.
No

?- x(A, A) =@= x(B, C).
No

?- x(A, A) =@= x(B, B).
A = _G267
B = _G270
Yes

5 ?- x(A, B) =@= x(C, D).
A = _G267
B = _G268
C = _G270
D = _G271
Yes

?- 3 @< 4.
Yes

?- 3 @< a.
Yes

?- a @< abc6.
Yes

?- abc6 @< t(c, d).
Yes

?- t(c, d) @< t(c, d, X).
X = _G284
Yes
```

II.3. Vị từ xác định kiểu

Do Prolog là một ngôn ngữ định kiểu yếu nên NLT thường xuyên phải xác định kiểu của các tham đối. Sau đây là một số vị từ xác định kiểu (type predicates) của Prolog..

Vị từ	*Kiểm tra*
`var(V)`	`V` là một biến ?

`nonvar(X)`	`X` không phải là một biến ?
`atom(A)`	`A` là một nguyên tử ?
`integer(I)`	`I` là một số nguyên ?
`float(R)`	`R` là một số thực (dấu chấm động) ?
`number(N)`	`N` là một số (nguyên hoặc thực) ?
`atomic(A)`	`A` là một nguyên tử hoặc một số ?
`compound(X)`	`X` là một hạng có cấu trúc ?
`ground(X)`	`X` là một hạng đã hoàn toàn ràng buộc ?

Ví dụ II.3 :

```
?- var(X).
X = _G201
Yes
?- integer(34).
Yes
?- ground(f(a, b)).
Yes
?- ground(f(a, Y)).
No
```

II.4. Một số vị từ xử lý hạng

Vị từ	*Kiểm tra*
`functor(T, F, N)`	`T` là một hạng với `F` là hạng tử và có `N` đối (arity)
`T =..L`	Chuyển đổi hạng `T` thành danh sách `L`
`clause(Head, Term)`	`Head :- Term` là một luật trong chương trình ?
`arg(N, Term, X)`	Thế biến `X` cho tham đối thứ `N` của hạng `Term`
`name(A, L)`	Chuyển nguyên tử `A` thành danh sách `L` gồm các mã ASCII (danh sách sẽ được trình bày trong chương sau).

Ví dụ II.4 :

```
?- functor(t(a, b, c), F, N).
F = t
N = 3
Yes
?- functor(father(jean, isa), F, N).
F = father, N = 2.
```

```
Yes
?- functor(T, father, 2).
T = father(_G346, _G347).      % _G346 và _G347 là hai biến của Prolog
?- t(a, b, c) =..L.
L = [t, a, b, c]
Yes
?- T =..[t, a, b, c, d, e].
T = t(a, b, c, d, e)
Yes
?- arg(1, father(jean, isa), X).
X = jean
?- name(toto, L).
L = [116, 111, 116, 111].
Yes
?- name(A, [116, 111, 116, 111]).
A = toto.
Yes
```

Ví dụ II.5 : Cho cơ sở dữ liệu :

```
personal(tom).
personal(ann).
father(X, Y) :- son(Y, X), male(X).
?- clause(father(X, Y), C).
C = (son(Y, X), male(X)).
?- clause(personal(X), C).
X = tom, C = true;
X = ann, C = true
Yes
```

III. Định nghĩa hàm

Prolog không có kiểu hàm, hàm phải được định nghĩa như một quan hệ trên các đối tượng. Các tham đối của hàm và giá trị trả về của hàm phải là các đối tượng của quan hệ đó. Điều này có nghĩa là không thể xây dựng được các hàm tổ hợp từ các hàm khác.

Ví dụ III.1 : Định nghĩa hàm số học cộng hai số bất kỳ

```
plus(X, Y, Z) :-        % trường hợp tính Z = X + Y
      nonvar(X), nonvar(Y),
      Z is X + Y.
plus(X, Y, Z) :-        % trường hợp  tính X = Z - Y
      nonvar(Y), nonvar(Z),
      X is Z - Y.
plus(X, Y, Z) :-        % trường hợp  tính Y - Z - X
      nonvar(X), nonvar(Z),
      Y is Z - X.

?- add1(2, 3, X).
X = 5
Yes

add1(7, X, 3).
X = -4
Yes

add1(X, 2, 6).
X = 4
Yes
```

III.1. Định nghĩa hàm sử dụng đệ quy

Trong chương 1, ta đã trình bày cách định nghĩa các luật (mệnh đề) đệ quy. Sau đây, ta tiếp tục ứng dụng phép đệ quy để xây dựng các hàm. Tương tự các ngôn ngữ lập trình mệnh lệnh, một thủ tục đệ quy của Prolog phải chứa các mệnh đề thoả mãn 3 điều kiện :

- *Một khởi động quá trình lặp.*
- *Một sơ đồ lặp lại chính nó.*
- *Một điều kiện dừng.*

Ví dụ thủ tục đệ quy tạo dãy 10 số tự nhiên chẵn đầu tiên như sau : đầu tiên lấy giá trị `0` để khởi động quá trình. Sau đó lấy `0` là giá trị hiện hành để tạo số tiếp theo nhờ sơ đồ lặp : `even_succ_nat = even_succ_nat + 2`. Quá trình

cứ tiếp tục như vậy cho đến khi đã có đủ 10 số `0 2 4 6 8 10 12 14 16 18` thì dừng lại.

Trong Prolog, một mệnh đề đệ quy (để tạo sơ đồ lặp) là mệnh đề có chứa trong thân (vế phải) ít nhất một lần lời gọi lại chính mệnh đề đó (vế trái) :

```
a(X) :- b(X, Y), a(Y).
```

Mệnh đề `a` gọi lại chính nó ngay trong vế phải. Dạng sơ đồ lặp như vậy được gọi là *đệ quy trực tiếp*. Để không xảy ra lời gọi vô hạn, cần có một mệnh đề làm điều kiện dừng đặt trước mệnh đề. Mỗi lần vào lặp mới, điều kiện dừng sẽ được kiểm tra để quyết định xem có thể tiếp tục gọi `a` hay không ?

Ta xây dựng thủ tục `even_succ_nat(Num, Count)` tạo lần lượt các số tự nhiên chẵn `Num`, biến `Count` để đếm số bước lặp. Điều kiện dừng là `Count=10`, ta có :

```
even_succ_nat(Num, 10).
```

Mệnh đề lặp được xây dựng như sau :

```
even_succ_nat(Num, Count) :-
     write(Num), write(' '),
     Count1 is Count + 1,
     Num1 is Num + 2,
     even_succ_nat(Num1, Count1).
```

Như vậy, lời gọi tạo 10 số tự nhiên chẵn đầu tiên sẽ là :

```
?- even_succ_nat(0, 0).
0 2 4 6 8 10 12 14 16 18
Yes
```

Một cách khác để xây dựng sơ đồ lặp được gọi là *đệ quy không trực tiếp* có dạng như sau :

```
a(X) :- b(X).
b(X) :- c(Y...), a(Z).
```

Trong sơ đồ lặp này, mệnh đề đệ quy `a` không gọi gọi trực tiếp đến `a`, mà gọi đến một mệnh đề `b` khác, mà trong `b` này lại có lời gọi đến `a`. Để không xảy ra lời gọi luẩn quẩn vô hạn, trong `b` cần thực hiện các tính toán làm giảm dần quá trình lặp trước khi gọi lại mệnh đề `a` (ví dụ mệnh đề `c`). Ví dụ sơ đồ dưới đây sẽ gây ra vòng luẩn quẩn vô hạn :

```
a(X) :- b(X, Y).
b(X, Y) :- a(Z).
```

Bài toán tạo 10 số tự nhiên chẵn đầu tiên được viết lại theo sơ đồ đệ quy không trực tiếp như sau :

```
a(0).
```

```
a(X) :- b(X).
b(X) :- X1 is X - 2, write(X), write(' '), a(X1).
```

Chương trình này không gọi « đệ quy » như `even_succ_nat`. Kết quả sau lời gọi `a(20)` là dãy số giảm dần `20 18 16 14 12 10 8 6 4 2`.

Ví dụ III.2 : Xây dựng số tự nhiên (Peano) và phép cộng trên các số tự nhiên

```
/* Định nghĩa số tự nhiên */
nat(0).                    % 0 là một số tự nhiên
nat(s(N)) :-               % s(X) cũng là một số tự nhiên
      nat(N).              % nếu N là một số tự nhiên
```

Chẳng hạn số 5 được viết : `s(s(s(s(s(zero)))))`

```
/* Định nghĩa phép cộng */
addi(0, X, X).             % luật 1 : 0 + X = X
/* addi(X, 0, X).          có thể sử dụng them luật 2 : X + 0 = X
addi(s(X), Y, s(Z)) :- % luật 3 : nếu X + Y = Z thì (X+1) + Y =
      (Z+1)
      addi(X, Y, Z).
```

Hoặc định nghĩa theo `nat(X)` như sau :

```
addi(0, X, X) :- nat(X).
```

```
?- addi(X, Y, s(s(s(0)))).
X = 0
Y = s(s(s(0)))
Yes
?- addi(X, s(s(0)), s(s(s(s(0))))).
X = s(s(0))
Yes
?- THREE = s(s(0)), FIVE = s(s(s(s(0)))),
addi(THREE, FIVE, EIGHT).
THREE = s(s(0))
FIVE = s(s(s(s(0))))
EIGHT = s(s(s(s(s(s(0))))))
Yes
```

Ví dụ III.3 : Tìm ước số chung lớn nhất (GCD: Greatest Common Divisor)

Cho trước hai số nguyên `X` và `Y`, ta cần tính ước số `D` và USCLN dựa trên ba quy tắc như sau :

1. Nếu `X = Y`, thì `D` bằng `X`.
2. Nếu `X < Y`, thì `D` bằng USCLN của `X` và của `Y - X`.
3. Nếu `X > Y`, thì thực hiện tương tự bước 2, bằng cách hoán vị vai trò `X` và `Y`.

Có thể dễ dàng tìm được các ví dụ minh hoạ sự hoạt động của ba quy tắc trước đây. Với `X =20` và `Y =25`, thì ta nhận được `D =5` sau một dãy các phép trừ.

Chương trình Prolog được xây dựng như sau :

```
gcd( X, X, X ).
gcd( X, Y, D ) :-
     X < Y,
     Y1 is Y - X,
     gcd( X, Y1, D ).
gcd( X, Y, D ) :-
     X > Y,
     gcd( Y, X, D ).
```

Đích cuối cùng trong mệnh đề thứ ba trên đây có thể được thay thế bởi :

```
  X1 is X - Y,
  gcd( X1, Y, D ).
```

Kết quả chạy Prolog như sau :

```
?- gcd( 20, 55, D ).
D = 5
```

Ví dụ III.4 : Tính giai thừa

```
fac(0, 1).
fac(N, F) :-
     N > 0,
     M is N - 1,
     fac(M, Fm),
     F is N * Fm.
```

Mệnh đề thứ hai có nghĩa rằng nếu lần lượt :

`N > 0`, `M = N - 1`, `Fm is (N-1)!`, và `F = N * Fm`,

thì `F` là `N!`. Phép toán `is` giống phép gán trong các ngôn ngữ lập trình mệnh lệnh nhưng trong Prolog, `is` không gán giá trị mới cho biến. Về mặt lôgich, thứ tự các mệnh đề trong vế phải của một luật không có vai trò gì, nhưng lại có ý nghĩa thực hiện chương trình. `M` không phải là biến trong lời gọi thủ tục đệ quy vì sẽ gây ra một vòng lặp vô hạn.

Các định nghĩa hàm trong Prolog thường rắc rối do hàm là quan hệ mà không phải là biểu thức. Các quan hệ được định nghĩa sử dụng nhiều luật và thứ tự các luật xác định kết quả trả về của hàm...

Ví dụ III.5 : Tính số Fibonacci

```
/* Fibonacci function */
```

```
fib(0, 0). % fib0 = 0
fib(1, 1).        % fib1 = 1
fib(N, F) :-      % fibn+2 = fibn+1 + fibn
    N > 1,
    N1 is N - 1, fib(N1, F1),
    N2 is N - 2, fib(N2, F2),
    F is F1 + F2.
?- fib(20, F).
F = 10946
Yes
?- fib(21, F).
ERROR: Out of local stack
```

Ta nhận thấy thuật toán tính số Fibonacci trên đây sử dụng hai lần gọi đệ quy đã nhanh chóng làm đầy bộ nhớ và chỉ với N=21, SWI-prolog phải dừng lại để thông báo lỗi.

Ví dụ III.6 : Tính hàm Ackerman

```
/* Ackerman's function */
ack(0, N, A) :-   % Ack(0, n) = n + 1
    A is N + 1.
ack(M1, 0, A) :-  % Ack(m, n) = Ack(m-1, 1)
    M > 0,
    M is M - 1,
    ack(M, 1, A).
ack(M1, N1, A) :-      % Ack(m, n) = Ack(m-1, Ack(m, n-1))
    M1 > 0, N1 > 0,
    M is M - 1, N is N - 1,
    ack(M1, N, A1),
    ack(M, A1, A).
```

Ví dụ III.7 : Hàm tính tổng

```
plus(X, Y, Z) :-
    nonvar(X), nonvar(Y),
    Z is X + Y.
plus(X, Y, Z) :-
    nonvar(Y), nonvar(Z),
    X is Z - Y.
plus(X, Y, Z) :-
    nonvar(X), nonvar(Z),
    Y is Z - X.
```

Ví dụ III.8 : Thuật toán hợp nhất

Sau đây là một thuật toán hợp nhất đơn giản cho phép xử lý trường hợp một biến nào đó được thay thế (hợp nhất) bởi một hạng mà hạng này lại có chứa đúng tên biến đó. Chẳng hạn phép hợp nhất `X = f(X)` là không hợp lệ.

```
% unify(T1, T2).
unify(X, Y) :-          % trường hợp 2 biến
    var(X), var(Y), X = Y.

unify(X, Y) :-          % trường hợp biến = không phải biến
    var(X), nonvar(Y), X = Y.

unify(X, Y) :-          % trường hợp không phải biến = biến
    nonvar(X), var(Y), Y = X.

unify(X, Y) :-          % nguyên tử hay số = nguyên tử hay số
    nonvar(X), nonvar(Y),
    atomic(X), atomic(Y),
    X = Y.

unify(X, Y) :-          % trường hợp cấu trúc = cấu trúc
    nonvar(X), nonvar(Y),
    compound(X), compound(Y),
    termUnify(X, Y).

termUnify(X, Y) :-      % hợp nhất hạng với hạng chứa cấu trúc
    functor(X, F, N),
    functor(Y, F, N),
    argUnify(N, X, Y).

argUnify(N, X, Y) :-    % hợp nhất N tham đối của X và Y
    N>0,
    argUnify1(N, X, Y),
    Ns is N - 1,
    argUnify(Ns, X, Y).

argUnify(0, X, Y).

argUnify1(N, X, Y) :-   % hợp nhất các tham đối có bậc N
    arg(N, X, ArgX),
    arg(N, Y, ArgY),
    unify(ArgX, ArgY).
```

Ví dụ III.9 : Lý thuyết số

Ta tiếp tục xây dựng hàm mới trên các số tự nhiên đã được định nghĩa trong ví dụ 1. Ta xây dựng phép so sánh hai số tự nhiên dựa trên phép cộng như sau :

```
egal(+(X, 0), X).       % phép cộng có tính giao hoán
egal(+(0, X), X).

egal(+(X, s(Y)), s(Z)) :-   % .X .Y..Z.egal(X+Y, Z) →
    egal(X+s(Y), s(Z))
```

```
    egal(+(X, Y), Z).
```

Sau đây là một số kết quả :

```
?- egal(s(s(0))+s(s(s(0))), s(s(s(s(s(0))))));
Yes
?- egal(+(s(s(0)), s(s(0))), X).
X = s(s(s(s(0))))
?- egal(+(X, s(s(0))), s(s(s(s(s(0))))));
X = s(s(s(0)))
Yes
?- egal(+(X, s(s(0))), s(s(s(s(s(0))))));
X = s(s(s(0)))
Yes
?- egal(X, s(s(s(s(0))))).
X = s(s(s(s(0))))+0 ;
X = 0+s(s(s(s(0)))) ;
X = s(s(s(0)))+s(0) ;
X = 0+s(s(s(s(0)))) ;
X = s(s(0))+s(s(0)) ;
X = 0+s(s(s(s(0)))) ;
X = s(0)+s(s(s(0))) ;
X = 0+s(s(s(s(0)))) ;
X = 0+s(s(s(s(0)))) ;
X = 0+s(s(s(s(0)))) ;
No
```

Với đích `egal(X, Y)` sau đây, câu trả lời là vô hạn :

```
?- egal(X, Y).
X = _G235+0
Y = _G235 ;
X = 0+_G235
Y = _G235 ;
X = _G299+s(0)
Y = s(_G299) ;
X = 0+s(_G302)
Y = s(_G302) ;
```

```
X = _G299+s(s(0))
Y = s(s(_G299)) ;

X = 0+s(s(_G309))
Y = s(s(_G309)) ;

X = _G299+s(s(s(0)))
Y = s(s(s(_G299))) ;

X = 0+s(s(s(_G316)))
Y = s(s(s(_G316))) ;

X = _G299+s(s(s(s(0))))
Y = s(s(s(s(_G299)))) ;

X = 0+s(s(s(s(_G323))))
Y = s(s(s(s(_G323)))) ;

X = _G299+s(s(s(s(s(0)))))
Y = s(s(s(s(s(_G299))))) ;

...

X = 0+s(s(s(s(s(s(_G337))))))
Y = s(s(s(s(s(s(_G337)))))) ;

X = _G299+s(s(s(s(s(s(s(0)))))))
Y = s(s(s(s(s(s(s(_G299))))))))

v.v...
```

III.2. Tối ưu phép đệ quy

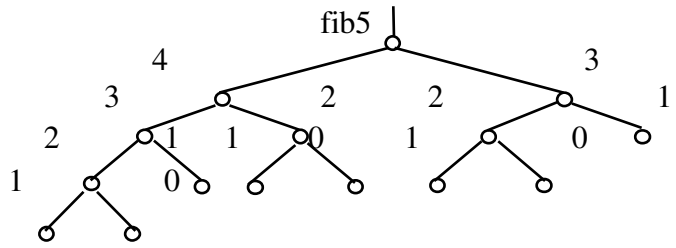
Lời giải các bài toán sử dụng đệ quy trong các ngôn ngữ lập trình nói chung thường ngắn gọn, dễ hiểu và dễ quản lý được chương trình. Tuy nhiên, trong một số trường hợp, sử dụng đệ quy lại xảy ra vấn đề về độ phức tạp tính toán, không những tốn kém bộ nhớ mà còn tốn kém thời gian.

Trong các ngôn ngữ mệnh lệnh, phép tính *n!* sử dụng đệ quy cần sử dụng bộ nhớ có cỡ *0(n)* và thời gian tính toán cũng có cỡ *0(n)*, thay vì gọi đệ quy, người ta thường sử dụng phép lặp *fac=fac*i, i=1..n*.

Ta xét lại ví dụ 4 tính số Fibonacci trên đây với lời gọi đệ quy :

```
fib(N, F) :-
    N > 1, N1 is N - 1, fib(N1, F1), N2 is N - 2,
    fib(N2, F2), F is F1 + F2.
```

Để ý rằng mỗi lần gọi hàm `fib(n)` với `n>1` sẽ dẫn tới hai lần gọi khác, nghĩa là số lần gọi sẽ tăng theo luỹ thừa 2. Với `n` lớn, chương trình gọi đệ quy như vậy dễ gây tràn bộ nhớ. Ví dụ sau đây là tất cả các lời gọi có thể cho trường hợp `n=5`.



Hình III.1. Biểu diễn cây các lời gọi đệ quy tìm số Fibonacci

Một số ngôn ngữ mệnh lệnh tính số Fibonacci sử dụng cấu trúc lặp để tránh tính đi tính lại cùng một giá trị. Chương trình Pascal dưới đây dùng hai biến phụ `x=fib(i)` và `y=fib(i+1)` :

```
{ tính fib(n) với n > 0 }
i:= 1; x:= 1; y:= 0;
while i < n do
     begin x:= x + y; y:= x - y end;
```

Ta viết lại chương trình Prolog như sau :

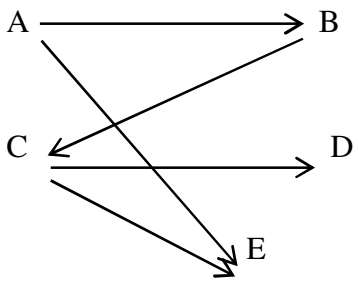
```
fibo(0, 0).
fibo(N, F) :-
    N >= 1,
    fib1(N, 1, 0, F).
fib1(1, F, _, F).
fib1(N, F2, F1, FN) :-
```

```
    N > 1,
    N1 is N - 1,
    F3 is F1 + F2,
    fib1(N1, F3, F2, FN).
?- fibo(21, F).
F = 10946
Yes
?- fibo(200, F).
F = 2.80571e+041
Yes
```

III.3. Một số ví dụ khác về đệ quy

III.3.1. Tìm đường đi trong một đồ thị có định hướng

Cho một đồ thị có định hướng như sau :



Hình III.2. Tìm đường đi trong một đồ thị có định hướng.

Ta xét bài toán tìm đường đi giữa hai đỉnh của đồ thị. Mỗi cung nối hai đỉnh của đồ thị biểu diễn một quan hệ giữa hai đỉnh này. Từ đồ thị trên, ta có thể viết các mệnh đề Prolog biểu diễn các sự kiện :

```
arc(a, b).
arc(b, c).
arc(c, e).
arc(c, d).
arc(a, e).
```

Giả sử cần kiểm tra có tồn tại một đường đi giữa hai nút `a` và `d` (không tồn tại đường đi giữa hai nút này như đã mô tả), ta viết mệnh đề :

```
path(a, d).
```

Để định nghĩa này, ta nhận xét như sau :

- Tồn tại một đường đi giữa hai nút có cung nối chúng.

- Tồn tại một đường đi giữa hai nút `X` và `Y` nếu tồn tại một nút thứ ba `Z` sao cho tồn tại một đường đi giữa `X` và `Z` và một đường đi giữa `Z` và `Y`.

Ta viết chương trình như sau :

```
path(X, Y) :- arc(X, Y).
path(X, Y) :-
      arc(X, Z),
      path(Z, Y).
```

Ta thấy định nghĩa thủ tục `path(X, Y)` tương tự thủ tục tìm tổ tiên gián tiếp giữa hai người trong cùng dòng họ `ancestor(X, Y)` đã xét trước đây.

```
?- path(X, Y).
X = a
Y = b ;

X = b
Y = c ;
...
```

III.3.2. Tính độ dài đường đi trong một đồ thị

Ta xét bài toán tính độ dài đường đi giữa hai nút, từ nút đầu đến nút cuối trong một đồ thị là số cung giữa chúng. Chẳng hạn độ dài đường đi giữa hai nút `a` và `d` là 3 trong ví dụ trên. Ta lập luận như sau :

- Nếu giữa hai nút có cung nối chúng thì độ dài đường đi là 1.
- Gọi `L` là độ dài đường đi giữa hai nút `X` và `Y`, `L1` là độ dài đường đi giữa một nút thứ ba `Z` và `Y` nếu tồn tại và giả sử có cung nối `X` và `Z`, khi đó `L = L1 + 1`.

Chương trình được viết như sau :

```
trajectory(X, Y, 1) :- arc(X, Y).
trajectory(X, Y, L) :-
      arc(X, Z),
      trajectory(Z, Y, L1),
      L is L1 + 1.

trajectory(a, d, L).
L = 3
Yes
```

III.3.3. Tính gần đúng các chuỗi

Trong Toán học thường gặp bài toán tính gần đúng giá trị của một hàm số với độ chính xác nhỏ tuỳ ý (e) theo phương pháp khai triển thành chuỗi Max Loren. Ví dụ tính hàm mũ e^x với độ chính xác 10^{-6} nhờ khai triển chuỗi Max Loren :

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + ...$$

Gọi `expower(X, S)` là hàm tính giá trị hàm mũ theo `X`, biến `S` là kết quả gần đúng với độ chính xác e=10^{-6}. Từ công thức khai triển Max Loren trên đây, ta nhận thấy giá trị của hàm mũ e^x là tổng vô hạn có dạng :

sum(0) = 1, t_0 = 1 tương ứng với x = 0 và $e^x = 1$
sum(i+1) = sum(i) + t_{i+1}, với $t_{i+1} = t_i * x /(i+1)$, i = 0, 1, 2 ...

Để thực hiện phép lặp, ta cần xây dựng hàm đệ quy tính tổng `sum(X, S, I, T)` trong đó sử dụng các biến trung gian `I` là bước lặp thứ *i* và `T` là số hạng t_i. Theo cách xây dựng này, hàm tính tổng `sum(X, S, I, T)` là tổng của các số hạng thứ `I` trở đi của chuỗi. Quá trình tính các tổng dừng lại khi t_i< e, nghĩa là đã đạt được độ chính xác e. Tại thời điểm này, giá trị của tổng cũng chính là số hạng t_i. Điều kiện khởi động quá trình lặp là chuyển vị từ `expower(X, S)` thành vị từ tính tổng `sum(X, S, I, T)` với giá trị đầu `I=0` và `T=1`.

Ta có chương trình đệ quy như sau :

```
expower(X, S) :-
      sum(X, S, 0, 1).
sum(_, T, _, T) :-
      abs(T) < 0.000001.
sum(X, S, I, T) :-
      abs(T) > 0.000001,
      I1 is I + 1, T1 is T*X/I1,
      sum(X, S1, I1, T1),
      S is S1 + T.
?- expower(1, S).
S = 2.71828
Yes
?- expower(10, S)
S = 22026.5
Yes
```

Tóm tắt chương 3

- Các phép toán số học được thực hiện nhờ các thủ tục thường trú trong Prolog.

- Vai trò của các phép toán tương tự vai trò của các hàm tử, chỉ để nhóm các thành phần của các cấu trúc mà thôi.
- Mỗi NLT có thể tự định nghĩa những phép toán riêng của mình. Mỗi phép toán được định nghĩa bởi tên, độ ưu tiên và kiểu gọi tham đối.
- Các phép toán cho phép NLT vận dụng cú pháp linh hoạt cho các nhu cầu riêng của họ. Sử dụng các phép toán làm cho chương trình trở nên dễ đọc (readability).
- Để tính một biểu thức số học, mọi tham đối có mặt trong biểu thức đó phải được ràng buộc bởi các giá trị số.
- Chỉ dẫn `op` dùng để định nghĩa một phép toán mới, gồm các yếu tố : tên, kiểu và độ ưu tiên của phép toán mới.
- Sử dụng các phép toán trung tố, tiền tố, hoặc hậu tố làm tăng cường tính dễ đọc của một chương trình Prolog.
- Độ ưu tiên là một số nguyên nằm trong một khoảng giá trị cho trước, thông thường nằm giữa 1 và 1200. Hàm tử chính của một biểu thức là phép toán có độ ưu tiên cao nhất. *Các phép toán có độ ưu tiên thấp nhất được ưu tiên nhất*.
- Kiểu của một phép toán phụ thuộc vào hai yếu tố :

  1. vị trí của phép toán so với các tham đối,

  2. độ ưu tiên của các tham đối được so sánh với độ ưu tiên của phép toán.

  Đối với các ký hiệu đặc tả `xfy`, tham đối `x` có độ ưu tiên *bé hơn hẳn* độ ưu tiên của phép toán, còn tham đối `y` có độ ưu tiên *bé hơn hoặc bằng* độ ưu tiên của phép toán.

Bài tập chương 3

1. Cho biết kết quả của các câu hỏi sau đây :

```
?- X=Y.
?- X is Y
?- X=Y, Y=Z, Z=1.
?- X=1, Z=Y, X=Y.
?- X is 1+1, Y is X.
?- Y is X, X is 1+1.
?- 1+2 == 1+2.
?- X == Y.
?- X == X.
```

```
?- 1 =:= 2-1
?- X =:= Y.
```

2. Cho biết kết quả của các câu hỏi sau đây :

```
?- op(X) is op(1).
?- op(X) = op(1).
?- op(op(Z), Y) = op(X, op(1)).
?- op(X, Y) = op(op(Y), op(X)).
```

3. Từ các định nghĩa số tự nhiên (`nat`) và phép cộng (`addi`) cho trong ví dụ 1 ở mục định nghĩa hàm, hãy viết tiếp các hàm trừ (`subt`), nhân (`multi`), chia (`divi`), luỹ thừa (`power`), giai thừa (`fact`), so sánh nhỏ hơn (`less`) và tìm ước số chung lớn nhất (`pdg`) sử dụng các hàm đã có (chẳng hạn `less`, `subt`...).

4. Viết hàm Prolog để kiểm tra một số nguyên tuỳ ý N :
   a. N là số chẵn (even number) sử dụng đệ quy trực tiếp
      Hướng dẫn : N chẵn thì N±2 cũng là số chẵn
   b. N là số lẻ (odd number) sử dụng đệ quy trực tiếp
      Hướng dẫn : N lẻ thì N±2 cũng là số lẻ
   c. N chẵn sử dụng hàm kiểm tra số lẻ câu **d** (N chẵn thì N±1 là số lẻ)
   d. N là số lẻ sử dụng hàm kiểm tra số chẵn câu **c** (N lẻ thì N±1 chẵn).

5. Viết hàm Prolog để làm duyệt (tracking/traverse) trên cây nhị phân theo các thứ tự trước (reorder), sau (post-order) và giữa (in-order).

   Giả sử cây nhị phân tương ứng với biểu thức số học `(5+6)*(3-(2/2))` là các mệnh đề Prolog như sau :

```
tree('*', tree('+', leaf(5), leaf(6)), tree('-',
leaf(3), tree('/', leaf(2), leaf(2)))
```

   Kết quả duyệt cây như sau : theo thứ tự trước :

```
[*, +, 5, 6, -, 3, /, 2, 2]
```

   thứ tự giữa :

```
[5, +, 6, *, 3, -, 2, /, 2]
```

   thứ tự sau :

```
[5, 6, +, 3, 2, 2, /, -, *]
```

6. Viết lại hàm tạo 10 số tự nhiên chẵn đầu tiên (đã cho trong phần đệ quy) sao cho kết quả trả về là dãy số tăng dần.

7. Lập bảng nhân `table(R, N)` có số bị nhân (multiplicator) từ 1 trở đi với số nhân `N` (multiplier) và dừng lại khi gặp số bị nhân `R` (kết quả `R * N`).

8. Viết các hàm tính gần đúng giá trị các hàm sau với độ chính xác e = 10^{-5} :

   $\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + ...$ cho đến khi $\left|\frac{1}{2n-1}\right| < \varepsilon$

   $1 + \frac{x^2}{2} + \frac{2}{3}\times\frac{x^4}{4} + \frac{2}{3}\times\frac{4}{5}\times\frac{x^6}{6} + ...$ cho đến khi phần tử thứ n < e

   $S = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \ldots + (-1)^n \frac{x^n}{n!} + \ldots$ cho đến khi $\left|\frac{x^n}{n!}\right| < \varepsilon$

   $S = 1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \frac{x^6}{6!} + ... + \frac{x^{2n}}{(2n)!} + ...$ cho đến khi $\left|\frac{x^{2n}}{(2n)!}\right| < 10^{-5}$

   $y = \sqrt{x + \sqrt{x + \ldots + \sqrt{x}}}$ có n > 1 dấu căn

9. Trình Prolog dưới đây là một trình diễn dịch (interpreter) cho một ngôn ngữ lập trình đơn giản chỉ gồm các số nguyên int(N), các biến id(X), các hàm fn(X,E), và gọi hàm app(E1,E2) :

```
%% subst(E1, E2, X, V)
%% thực hiện phép thế biến X bởi biến V trong E1 để trả về E2.
subst(int(N), int(N), _, _).
subst(id(X), V, X, V).
subst(id(Y), id(Y), X, _) :-
        X \= Y.
subst(fn(X, E), fn(X, E), X, _).
subst(fn(Y, Ea), fn(Y, Eb), X, V) :-
        X \= Y,
        subst(Ea, Eb, X, V).
subst(app(E1a, E2a), app(E1b, E2b), X, V) :-
        subst(E1a, E1b, X, V),
        subst(E2a, E2b, X, V).
%% reduce(E, V)
%% thực hiện phép tính giá trị của E để trả về V.
reduce(int(N), int(N)).
reduce(fn(X, B), fn(X, B))
reduce(app(E1, E2), V) :-
        reduce(E1, fn(X, B)),
        reduce(E2, V2),
```

```
        subst(B, E, X, V2),
        reduce(E, V).
```

Câu hỏi :

a. Cho biết cách trao đổi tham biến hợp lệ trong ngôn ngữ mô tả trên đây ? Cách trao đổi tham biến nào thì không thể thực hiện được ?

b. Tìm cách thay đổi trình Prolog trên đây để có thể thực hiện được các phương pháp trao đổi tham biến khác nhau.

c. Cho biết tầm vực (scope) của các biến là tĩnh hay động ?

10. Cho ví dụ một đồ thị không định hướng dưới đây :

```
arc(a,b).                    arc(d,f).
arc(b,c).                    arc(f,a).
arc(c,d).                    arc(a,b).
arc(c,e).                    arc(h,i).
arc(c,g).                    arc(i,j).
arc(g,f).
```

Hãy viết hàm tìm đường đi giữa hai đỉnh của đồ thị.

CHƯƠNG 4

Cấu trúc danh sách

Chương này trình bày khái niệm về *danh sách*, một trong những cấu trúc đơn giản nhất và thông dụng nhất, cùng với những chương trình tiêu biểu minh hoạ cách vận dụng danh sách trong Prolog. Cấu trúc danh sách tạo nên một môi trường lập trình thuận tiện của ngôn ngữ Prolog.

I. Biểu diễn cấu trúc danh sách

Danh sách là kiểu cấu trúc dữ liệu được sử dụng rộng rãi trong các ngôn ngữ lập trình phi số. Một danh sách là một dãy bất kỳ các đối tượng. Khác với kiểu dữ liệu tập hợp, các đối tượng của danh sách có thể trùng nhau (xuất hiện nhiều lần) và mỗi vị trí xuất hiện của đối tượng đều có ý nghĩa.

Danh sách là cách diễn đạt ngắn gọn của kiểu dữ liệu hạng phức hợp trong Prolog. Hàm tử của danh sách là dấu chấm "**.**". Do việc biểu diễn danh sách bởi hàm tử này có thể tạo ra những biểu thức mập mờ, nhất là khi xử lý các danh sách gồm nhiều phần tử lồng nhau, cho nên Prolog quy ước đặt dãy các phần tử của danh sách giữa các cặp móc vuông.

Chẳng hạn `.(a,.(b,[ ]))`. Là danh sách `[ a, b ]`.

Danh sách các phần tử `anne`, `tennis`, `tom`, `skier` (tên người) được viết :

```
[ anne, tennis, tom, skier ]
```

chính là hàm tử :

```
. ( anne, .( tennis, .( tom, .( skier, [ ] ) ) ) )
```

Cách viết dạng cặp móc vuông chỉ là xuất hiện bên ngoài của một danh sách. Như đã thấy ở mục trước, mọi đối tượng cấu trúc của Prolog đều có biểu diễn cây. Danh sách cũng không nằm ngoại lệ, cũng có cấu trúc cây.

Làm cách nào để biểu diễn danh sách bởi một đối tượng Prolog chuẩn ? Có hai khả năng xảy ra là danh sách có thể rỗng hoặc không. Nếu danh sách rỗng, nó được viết dưới dạng một nguyên tử :

```
[ ]
```

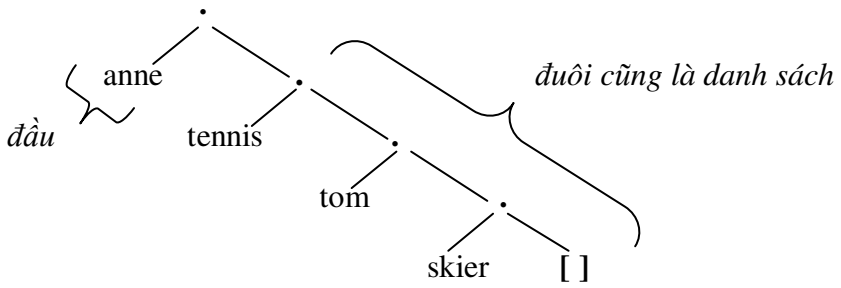
Nếu danh sách khác rỗng, có thể xem nó được cấu trúc từ hai thành phần (pair syntax) :

1. Thành phần thứ nhất, được gọi là *đầu* (head) của danh sách.
2. Thành phần thứ hai, phần còn lại của danh sách (trừ ra phần đầu), được gọi là *đuôi* (tail) của danh sách, cũng là một danh sách.

Trong ví dụ trên thì đầu là `anne`, còn đuôi là danh sách :

```
[ tennis, tom, skier ]
```

Nói chung, đầu của danh sách có thể là một đối tượng bất kỳ của Prolog, có thể là cây hoặc biến, nhưng đuôi phải là một danh sách. Hình I.1. Biểu diễn dạng cây của danh sách mô tả cấu trúc cây của danh sách đã cho :



Hình I.1. Biểu diễn dạng cây của danh sách

Vì đuôi `tail` là một danh sách, nên `tail` có thể rỗng, hoặc lại có thể được tạo thành từ một đầu `head` và một đuôi `tail` khác.

Chú ý rằng danh sách rỗng xuất hiện trong số các hạng, vì rằng phần tử cuối cùng có thể xem là danh sách chỉ gồm một phần tử duy nhất có phần đuôi là một danh sách rỗng:

```
[ skier ]
```

Ví dụ trên đây minh hoạ nguyên lý cấu trúc dữ liệu tổng quát trong Prolog áp dụng cho các danh sách có độ dài tuỳ ý.

```
?- L1 = [ a, b, c ].
?- L2 = [ a, a, a ].
L1 = [ a, b, c ]
L2 = [ a, a, a ]
?- Leisure1 = [ tennis, music, [ ] ].
?- Leisure2 = [ sky, eating ],
?- L = [ anne, Leisure1, tom, Leisure2 ].
Leisure1 = [ tennis, music ]
Leisure2 = [ sky, eating ]
L = [ anne, [ tennis, music ], tom, [ sky, eating ] ]
```

Như vậy, các phần tử của một danh sách có thể là các đối tượng có kiểu bất kỳ, kể cả kiểu danh sách. Thông thường, người ta xử lý đuôi của danh sách như là một danh sách. Chẳng hạn, danh sách :

```
L = [ a, b, c ]
```

có thể viết :

`tail = [ b, c ]` và `L = .(a, tail)`

Để biểu diễn một danh sách được tạo thành từ đầu (`Head`) và đuôi (`Tail`), Prolog sử dụng ký hiệu | (split) để phân cách phần đầu và phần đuôi như sau :

```
L = [ a | Tail ]
```

Ký hiệu | được dùng một cách rất tổng quát bằng cách viết một số phần tử tuỳ ý của danh sách trước | rồi danh sách các phần tử còn lại. Danh sách bây giờ được viết lại như sau :

```
 [ a, b, c ] = [ a | [ b, c ] ] = [ a, b | [ c ] ] = [
a, b, c | [ ] ]
```

Sau đây là một số cách viết danh sách :

Kiểu hai thành phần	*Kiểu liệt kê phần tử*
`[ ]`	`[ ]`
`[ a | [ ] ]`	`[ a ]`
`[ a | b | [ ] ]`	`[ a, b ]`
`[ a | X ]`	`[ a | X ]`
`[ a | b | X ]`	`[ a, b | X ]`
`[ X1 | [ ... [ Xn | [ ] ]... ] ]`	`[ X1, ... , Xn ]`

Ta có thể định nghĩa danh sáchtheo kiểu đệ quy như sau :

```
List → [ ]
List → [ Element | List ]
```

II. Một số vị từ xử lý danh sách của Prolog

SWI-Prolog có sẵn một số vị từ xử lý danh sách như sau :

Vị từ	*Ý nghĩa*
`append(List1, List2, List3)`	Ghép hai danh sách `List1` và `List2` thành `List3`.
`member(Elem, List)`	Kiểm tra `Elem` có là phần tử của danh sách `List` hay không, nghĩa là `Elem` hợp nhất được với một trong các phần tử của `List`.
`nextto(X, Y, List)`	Kiểm tra nếu phần tử Y có đứng ngay sau phần tử X trong danh sách `List` hay không.

`delete(List1, Elem, List2)`	Xoá khỏi danh sách `List1` những phần tử hợp nhất được với `Elem` để trả về kết quả `List2`.
`select(Elem, List, Rest)`	Lấy phần tử `Elem` ra khỏi danh sách `List` để trả về những phần tử còn lại trong Rest, có thể dùng để chèn một phần tử vào danh sách.
`nth0(Index, List, Elem)`	Kiểm tra phần tử thứ Index (tính từ 0) của danh sách `List` có phải là `Elem` hay không.
`nth1(Index, List, Elem)`	Kiểm tra phần tử thứ `Index` (tính từ 1) của danh sách `List` có phải là `Elem` hay không.
`last(List, Elem)`	Kiểm tra phần tử đứng cuối cùng trong danh sách `List` có phải là `Elem` hay không.
`reverse(List1, List2)`	Nghịch đảo thứ tự các phần tử của danh sách `List1` để trả về kết quả `List2`.
`permutation(List1, List2)`	Hoán vị danh sách `List1` thành danh sách `List2`.
`flatten(List1, List2)`	Chuyển danh sách `List1` chứa các phần tử bất kỳ thành danh sách phẳng `List2`. Ví dụ : `flatten([a, [b, [c, d], e]], X).` cho kết quả `X = [a, b, c, d, e]`.
`sumlist(List, Sum)`	Tính tổng các phần tử của danh sách `List` chứa toàn số để trả về kết quả `Sum`.
`numlist(Low, High, List)`	Nếu `Low` và `High` là các số sao cho `Low =< High`, thì trả về danh sách `List = [Low, Low+1, ..., High]`.

Chú ý một số vị từ xử lý danh sách có thể sử dụng cho mọi ràng buộc, kể cả khi các tham đối đều là biến.

Trong Prolog, tập hợp được biểu diễn bởi danh sách, tuy nhiên, thứ tự các phần tử trong một tập hợp là không quan trọng, các đối tượng dù xuất hiện nhiều lần chỉ được xem là một phần tử của tập hợp. Các phép toán về danh sách có thể áp dụng cho các tập hợp. Đó là :

- Kiểm tra một phần tử có mặt trong một danh sách tương tự việc kiểm tra một phần tử có thuộc về một tập hợp không ?
- Ghép hai danh sách để nhận được một danh sách thứ ba tương ứng với phép hợp của hai tập hợp.
- Thêm một phần tử mới, hay loại bỏ một phần tử.

Prolog có sẵn một số vị từ xử lý tập hợp như sau :

Vị từ	*Ý nghĩa*
`is_set(Set)`	Kiểm tra `Set` có phải là một tập hợp hay không
`list_to_set(List, Set)`	Chuyển danh sách `List` thành tập hợp `Set` giữ nguyên thứ tự các phần tử của `List` (nếu `List` có các phần tử trùng nhau thì chỉ lấy phần tử gặp đầu tiên). Ví dụ : `list_to_set([a,b,a], X)` cho kết quả `X = [a,b].`
`intersection(Set1, Set2, Set3)`	Phép giao của hai tập hợp `Set1` và `Set2` là `Set3`.
`subtract(Set, Delete, Result)`	Trả về kết quả phép hiệu của hai tập hợp `Set` và `Delete` là `Result` (là tập `Set` sau khi đã xoá hết các phần tử của `Delete` có mặt trong đó).
`union(Set1, Set2, Set3)`	Trả về kết quả phép hợp của hai tập hợp `Set1` và `Set2` là `Set3`.
`subset(Subset, Set)`	Kiểm tra tập hợp `Subset` có là tập hợp con của `Set` hay không.

III. Các thao tác cơ bản trên danh sách

III.1. Xây dựng lại một số vị từ có sẵn

Sau đây ta sẽ trình bày một số thao tác cơ bản trên danh sách bằng cách xây dựng lại một số vị từ có sẵn của Prolog.

III.1.1. Kiểm tra một phần tử có mặt trong danh sách

Prolog kiểm tra một phần tử có mặt trong một danh sách như sau :

```
member(X, L)
```

trong đó, `X` là một phần tử và `L` là một danh sách. Đích `member(X, L)` được thoả mãn nếu `X` xuất hiện trong `L`. Ví dụ :

```
?- member( b, [ a, b, c ] )
Yes
?- member( b, [ a, [ b, c ] ] )
No
?- member( [ b, c], [ a, [ b, c ] ] )
Yes
```

Từ các kết quả trên, ta có thể giải thích quan hệ `member(X, L)` như sau :

Phần tử `X` thuộc danh sách `L` nếu :

1. `X` là đầu của `L`, hoặc nếu
2. `X` là một phần tử của đuôi của `L`.

Ta có thể viết hai điều kiện trên thành hai mệnh đề, mệnh đề thứ nhất là một sự kiện đơn giản, mệnh đề thứ hai là một luật :

```
member( X, [ X | Tail ] ).
member( X, [ Head | Tail ] ) :- member( X, Tail ).
```

hoặc :

```
member(X, [X|T]).
member(X, [_|T]) :- member(X, T).
```

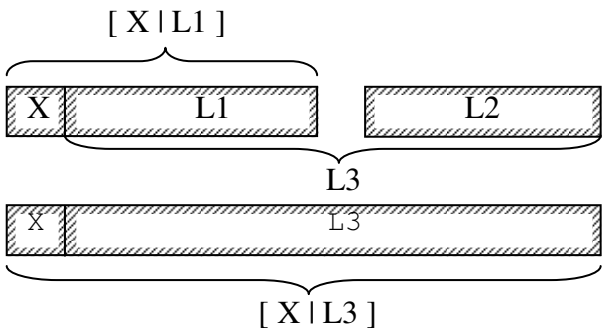
III.1.2. Ghép hai danh sách

Để ghép hai danh sách, Prolog có hàm :

```
append( L1, L2, L3).
```

trong đó, `L1` và `L2` là hai danh sách, `L3` là danh sách kết quả của phép ghép `L1` và `L2`. Ví dụ :

```
?- append( [ a, b ], [ c, d ], [ a, b, c, d ] ).
Yes
?- append( [ a, b ], [ c, d ], [ a, b, a, c ] ).
No
```



Hình III.1. Ghép hai danh sách `[ X | L1 ]` và `L2` thành `[ X | L3 ]`.

Hàm `append` hoạt động phụ thuộc tham đối đầu tiên `L1` theo cách như sau :

1. Nếu tham đối đầu tiên là danh sách rỗng, thì tham đối thứ hai và thứ ba phải là một danh sách duy nhất, gọi là `L`. Ta viết trong Prolog như sau :

   ```
   append( [ ], L, L).
   ```

2. Nếu tham đối đầu tiên của `append` là danh sách khác rỗng, thì nó gồm một đầu và một đuôi như sau

   ```
   [ X | L1 ]
   ```

Kết quả phép ghép danh sách là danh sách `[ X | L3 ]`, với `L3` là phép ghép của `L1` và `L2`. Ta viết trong Prolog như sau :

```
append( [ X | L1 ], L2, [ X | L3 ] ) :- append( L1,
L2, L3 ).
```

Hình 4.2 minh hoạ phép ghép hai danh sách `[ X | L1 ]` và `L2`.

Ta có các ví dụ sau :

```
?- append( [ a, b, c ], [ 1, 2, 3 ], L ).
L = [ a, b, c, 1, 2, 3 ]
?- append( [ a, [ b, c ], d ], [ a, [ ], b ], L ] ).
L = [ a, [ b, c ], d, a, [ ], b ]
```

Thủ tục `append` được sử dụng rất mềm dẻo theo nhiều cách khác nhau.

Chẳng hạn Prolog đưa ra bốn phương án để phân tách một danh sách đã cho thành hai danh sách mới như sau :

```
?- append( L1, L2, [ a, b, c ] ).
L1 = [ ]
L2 = [ a, b, c ];
L1 = [ a ]
L2 = [ b, c ];
L1 = [ a, b ]
L2 = [ c ];
L1 = [ a, b, c ]
L2 = [ ];
Yes
```

Sử dụng `append`, ta cũng có thể tìm kiếm một số phần tử trong một danh sách. Chẳng hạn, từ danh sách các tháng trong năm, ta có thể tìm những tháng đứng trước một tháng đã cho, giả sử tháng năm (May) :

```
?- append( Before, [ May | After ] ,
     [ jan, fev, mar, avr, may, jun, jul, aut, sep, oct,
     nov, dec ] ).
Before = [ jan, fev, mar, avr ]
After = [ jun, jul, aut, sep, oct, nov, dec ]
Yes
```

Tháng đứng ngay trước và tháng đứng ngay sau tháng năm nhận được như sau :

```
?- append( _, [ Month1, may, Month2 | _ ] ,
     [ jan, fev, mar, avr, may, jun, jul, aut, sep, oct,
     nov, dec ] ).
```

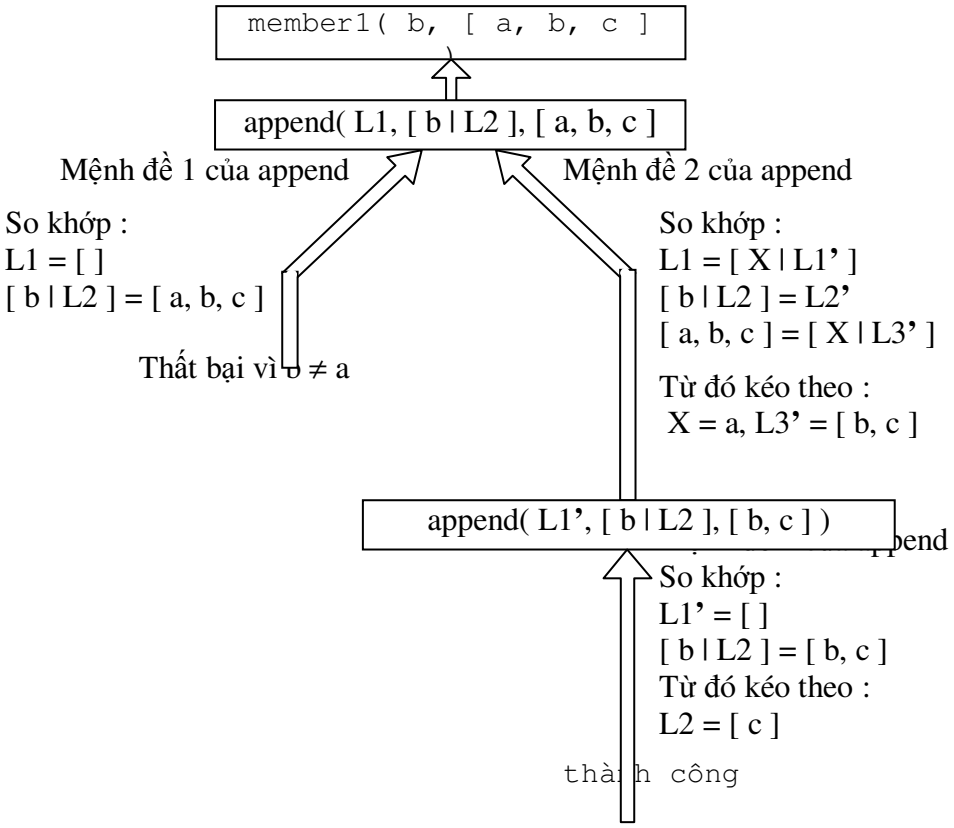
```
Month1 = avr
Month2 = jun
Yes
```

Bây giờ cho trước danh sách :

```
L1 = [ a, b, z, z, c, z, z, z, d, e ]
```

Ta cần xóa các phần tử đứng sau ba chữ `z` liên tiếp, kể cả ba chữ `z` :

```
?- L1 = [ a, b, z, z, c, z, z, z, d, e ],
     append( L2, [ z, z, z | _ ], L1 ).
L1 = [ a, b, z, z, c, z, z, z, d, e ]
L2 = [ a, b, z, z, c ]
```



Hình III.2. Thủ tục `member1` tìm tuần tự một đối tượng trong danh sách đã cho.

Trước đây ta đã định nghĩa quan hệ `member( X, L )` để kiểm tra một phần tử `X` có mặt trong một danh sách `L` không. Bây giờ bằng cách sử dụng `append`, ta có thể định nghĩa lại `member` như sau :

```
member1( X, L ) :- append( L1, [ X | L2], L).
```

Mệnh đề này có nghĩa : nếu `X` có mặt trong danh sách `L` thì `L` có thể được phân tách thành hai danh sách, với `X` là đầu của danh sách thứ hai. Định nghĩa `member1` hoàn toàn tương đương với định nghĩa `member`.

Ở đây ta sử dụng hai tên khác nhau để phân biệt hai cách cài đặt Prolog. Ta cũng có thể định nghĩa lại `member1` bằng cách sử dụng biến nặc danh (anonymous variable) :

```
member1( X, L ) :-
            append( _ , [ X | _ ], L).
```

So sánh hai cách cài đặt khác nhau về quan hệ thành viên, ta nhận thấy nghĩa thủ tục trong định nghĩa `member` được thể hiện rất rõ :

Trong `member`, để kiểm tra phần tử `X` có mặt trong một danh sách `L` không,

1. Trước tiên kiểm tra phần tử đầu của `L` là đồng nhất với `X`, nếu không,
2. Kiểm tra rằng `X` có mặt trong phần đuôi của `L`.

Nhưng trong trường hợp định nghĩa `member1`, ta thấy hoàn toàn nghĩa khai báo mà không có nghĩa thủ tục.

Để hiểu được cách `member1`hoạt động như thế nào, ta hãy xem xét quá trình Prolog thực hiện câu hỏi :

```
?- member1( b, [ a, b, c ] ).
```

Cách tìm của thủ tục `member1` trên đây tương tự `member`, bằng cách duyệt từng phần tử, cho đến khi tìm thấy đối tượng cần tìm, hoặc danh sách đã cạn.

III.1.3. Bổ sung một phần tử vào danh sách

Phương pháp đơn giản nhất để bổ sung một phần tử vào danh sách là đặt nó ở vị trí đầu tiên, để nó trở thành đầu. Nếu `X` là một đối tượng mới, còn `L` là danh sách cần bổ sung thêm, thì danh sách kết quả sẽ là :

```
[ X | L ]
```

Người ta không cần viết thủ tục để bổ sung một phần tử vào danh sách. Bởi vì việc bổ sung có thể được biểu diễn dưới dạng một sự kiện nếu cần :

```
insert( X, L, [ X | L ] ).
```

III.1.4. Loại bỏ một phần tử khỏi danh sách

Để loại bỏ một phần tử `X` khỏi danh sách `L`, người ta xây dựng quan hệ :

```
remove( X, L, L1 )
```

trong đó, `L1` đồng nhất với `L`, sau khi `X` bị loại bỏ khỏi `L`. Thủ tục `remove` có cấu trúc tương tự `member`. Ta có thể lập luận như sau

1. Nếu phần tử `X` là đầu của danh sách, thì kết quả là đuôi của danh sách.

2. Nếu không, tìm cách loại bỏ X khỏi phần đuôi của danh sách.

```
remove( X, [ X | Tail ], Tail ).
remove( X, [ Y | Tail ], [ Y | Tail1 ] ) :-
    remove( X, Tail, Tail1 ).
```

Tương tự thủ tục member, thủ tục remove mang tính không xác định. Nếu có nhiều phần tử là X có mặt trong danh sách, thì remove có thể xoá bất kỳ phần tử nào, do quá trình quay lui. Tuy nhiên, mỗi lần thực hiện, remove chỉ xoá một phần tử là X mà không đụng đến những phần tử khác. Ví dụ :

```
?- remove( a, [ a, b, a, a ], L ).

L = [ b, a, a ];
L = [ a, b, a ];
L = [ a, b, a ]
No
```

Thủ tục remove thất bại nếu danh sách không chứa phần tử cần xoá. Người ta có thể sử dụng remove trong một khía cạnh khác, mục đích để bổ sung một phần tử mới vào bất cứ đâu trong danh sách.

Ví dụ, nếu ta muốn đặt phần tử a vào tại mọi vị trí bất kỳ trong danh sách [1, 2, 3], chỉ cần đặt câu hỏi : Cho biết danh sách L nếu sau khi xoá a, ta nhận được danh sách [1, 2, 3] ?

```
?- remove( a, L, [ 1, 2, 3 ] ).

L = [ a, 1, 2, 3 ];
L = [ 1, a, 2, 3 ];
L = [ 1, 2, a, 3 ];
L = [ 1, 2, 3, a ]
No
```

Một cách tổng quát, phép toán chèn insert một phần tử X vào một danh sách List được định nghĩa bởi thủ tục remove bằng cách sử dụng một danh sách lớn hơn LargerList làm tham đối thứ hai :

```
insert( X, List, LargerList ) :-
    remove( X, LargerList, List ).
```

Ta đã định nghĩa quan hệ thuộc về trong thủ tục member1 bằng cách sử dụng thủ tục append. Tuy nhiên, ta cũng có thể định nghĩa lại quan hệ thuộc về trong thủ tục mới member2 bởi thủ tục remove bằng cách xem một phần tử X thuộc về một danh sách List nếu X bị xoá khỏi List :

```
member2( X, List ) :-
      remove( X, List, _ ).
```

III.1.5. Nghịch đảo danh sách

Sử dụng `append`, ta có thể viết thủ tục nghịch đảo một danh sách như sau :

```
reverse ( [ ], [ ] ).
reverse ( [ X | Tail ], R ) :-
     reverse (Tail, R1 ),
     append(R1, [X], R).
?- reverse( [ a, b, c , d, e, f ] , L).
L = [f, e, d, c, b, a]
Yes
```

Sau đây là một thủ tục khác để nghịch đảo một danh sách nhưng có sử dụng hàm bổ trợ trong thân thủ tục :

```
revert(List, RevList) :-
     rev(List, [ ], RevList).
rev([ ], R, R).
rev([H|T], S, R) :-
     rev(T, [H|S], R).
?- revert( [ a, b, c , d, e, f ] , R).
R = [f, e, d, c, b, a]
Yes
```

Sử dụng `reverse`, ta có thể kiểm tra một danh sách có là đối xứng (palindrome) hay không :

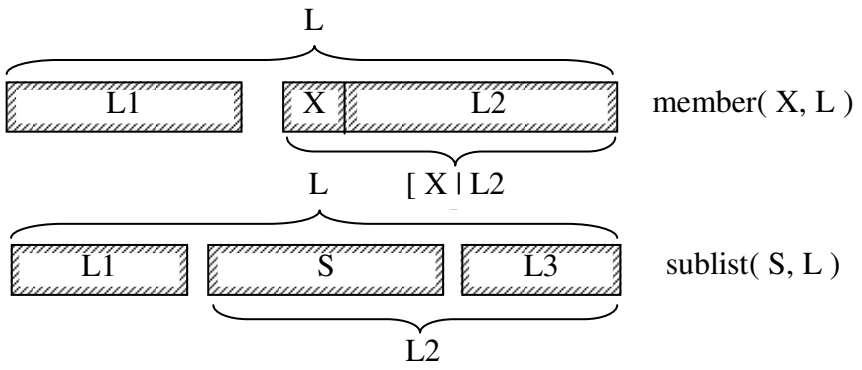
```
palindrome(L) :-
     reverse( L, L ).
?- palindrome([ a, b, c , d, c, b, a ]).
Yes
```

III.1.6. Danh sách con

Ta xây dựng thủ tục `sublist` nhận hai tham đối là hai danh sách `L` và `S` sao cho `S` là danh sách con của `L` như sau :

```
?- sublist( [ c, d, e ], [ a, b, c , d, e, f ] )
Yes
?- sublist( [ c, e ], [ a, b, c , d, e, f ] )
No
```

Nguyên lý để xây dựng thủ tục `sublist` tương tự thủ tục `member1`, mặc dù ở đây quan hệ danh sách con tổng quát hơn.



Hình III.3. Các quan hệ `member` *và* `sublist`.

Quan hệ danh sách con được mô tả như sau :

`S` là một danh sách con của `L` nếu :

1. Danh sách `L` có thể được phân tách thành hai danh sách `L1` và `L2`, và nếu
2. Danh sách `L2` có thể được phân tách thành hai danh sách `S` và `L3`.

Như đã thấy, việc phân tách các danh sách có thể được mô tả bởi quan hệ ghép `append`.

Do đó ta viết lại trong Prolog như sau :

```
sublist( S, L ) :-
    append( L1, L2, L ), append( S, L3, L2 ).
```

Ta thấy thủ tục `sublist` rất mềm dẻo và do vậy có thể sử dụng theo nhiều cách khác nhau. Chẳng hạn ta có thể liệt kê mọi danh sách con của một danh sách đã cho như sau :

```
?- sublist( S, [ a, b, c ] ).
S = [ ];
S = [ a ];
S = [ a, b ];
S = [ a, b, c ];
S = [ b ];
...
```

III.2. Hoán vị

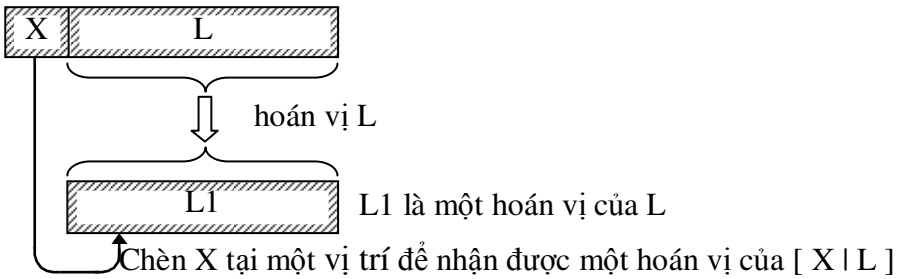
Đôi khi, ta cần tạo ra các hoán vị của một danh sách. Ta xây dựng quan hệ `permutation` có hai tham biến là hai danh sách, mà một danh sách là hoán vị của danh sách kia. Ta sẽ tận dụng phép quay lui như sau :

```
?- permutation( [ a, b, c ], P ).
P = [ a, b, c ];
P = [ a, c, b ];
```

```
P = [ b, a, c ];
...
```

Nguyên lý hoạt động của thủ tục `swap` dựa trên hai trường hợp phân biệt, tuỳ theo danh sách thứ nhất :

1. Nếu danh sách thứ nhất rỗng, thì danh sách thứ hai cũng phải rỗng.
2. Nếu danh sách thứ nhất khác rỗng, thì nó sẽ có dạng `[ X | L ]` và được tiến hành hoán vị như sau : trước tiên hoán vị `L` để nhận được `L1`, sau đó chèn `X` vào tất cả các vị trí trong `L1`.



Hình III.4. Một cách xây dựng hoán vị `permutation` của danh sách `[ X | L ]`.

Ta nhận được hai mệnh đề tương ứng với thủ tục như sau :

```
permutation( [ ], [ ] ).
permutation( [ X | L ], P ) :-
    permutation( L, L1 ), insert( X, L1, P ).
```

Một phương pháp khác là loại bỏ phần tử `X` khỏi danh sách đầu tiên, hoán vị phần còn lại của danh sách này để nhận được danh sách `P`, sau đó thêm `X` vào phần đầu của `P`. Ta có chương trình khác `permutation2` như sau :

```
permutation2( [ ], [ ] ).
permutation2( L, [ X | P ] ) :-
    remove( X, L, L1 ), permutation2( L1, P ).
```

Từ đây, ta có thể khai thác thủ tục hoán vị, chẳng hạn (chú ý khi chạy Arity Prolog cần gõ vào một dấu chấm phẩy `;` sau `->`) :

```
?- permutation( [ red, blue, green ], P ).
P = [ red, blue, green ];
P = [ red, green, blue ];
P = [ blue, red, green ];
P = [ blue, green, red ];
P = [ green, red, blue ];
P = [ green, blue, red ];
Yes
```

Hoặc nếu sử dụng `permutation` theo cách khác như sau :

```
?- permutation( L, [ a, b, c ] ).
```

Prolog sẽ ràng buộc liên tiếp cho `L` để đưa ra 6 hoán vị khác nhau có thể. Tuy nhiên, nếu NSD yêu cầu một giải pháp khác, Prolog sẽ không bao giờ trả lời "`No`", mà rơi vào một vòng lặp vô hạn do phải tìm kiếm một hoán vị mới mà thực ra không tồn tại. Trong trường hợp này, thủ tục `permutation2` chỉ tìm thấy một hoán vị thứ nhất, sau đó ngay lập tức rơi vào một vòng lặp vô hạn. Vì vậy, cần chú ý khi sử dụng các quan hệ hoán vị này.

III.3. Một số ví dụ về danh sách

III.3.1. Sắp xếp các phần tử của danh sách

Xây dựng thủ tục sắp xếp các phần tử có của một danh sách bằng phương pháp chèn như sau :

```
ins(X, [ ], [ X ]).
ins(X, [H|T], [ X,H|T ]) :-
    X @=< H.
ins(X, [ H|T ], [ H|L ]) :-
    X @> H, ins( X, T, L ).
?- ins(8, [ 1, 2, 3, 4, 5 ], L).
L = [1, 2, 3, 4, 5, 8]
Yes
?- ins(1, L, [ 1, 2, 3, 4, 5 ]).
L = [2, 3, 4, 5]
Yes
ins_sort([ ], [ ]).
ins_sort([H|T], L) :-
    ins_sort(T, L1),
    ins(H, L1, L).
?- ins_sort([3, 2, 6, 4, 7, 1], L).
L = [1, 2, 3, 4, 6, 7]
Yes
```

III.3.2. Tính độ dài của một danh sách

Xây dựng thủ tục tính độ dài hay đếm số lượng các phần tử có mặt trong một danh sách đã cho như sau :

```
length( L, N ).
```

Xảy ra hai trường hợp :

1. Nếu danh sách rỗng, thì độ dài `N = 0`.
2. Nếu danh sách khác rỗng, thì nó được tạo thành từ danh sách có dạng :
   `[ head | queue ]`
   và có độ dài bằng `1` cộng với độ dài của `queue`.

Ta có chương trình Prolog như sau :

```
length( [ ], 0 ).
length( [ _ | Queue ], N ) :-
      length(Queue, N1 ),
      N is 1 + N1.
```

Kết quả chạy Prolog như sau :

```
?- length( [ a, b, c, d, e ], N ).
N = 5
Yes
?- length( [ a, [ b, c ], d, e ], N ).
N = 4
Yes
```

Ta thấy rằng trong mệnh đề thứ hai, hai đích của phần thân là không thể hoán đổi cho nhau, vì rằng `N1` phải được ràng buộc trước khi thực hiện đích :

```
N is 1 + N1
```

Chẳng hạn, nếu gọi `trace`, quá trình thực hiện `length( [ 1, 2, 3 ], N )` như sau :

```
(0)   gọi          length([1, 2, 3], N)   ->
(1)   gọi          length([2, 3], N')     ->
(2)   gọi          length([3], N'')  ->
(3)   gọi          length([ ], N''') ->   N''' = 0
(4)   gọi          N'' is 1 + 0 ->   N'' = 1
(5)   gọi          N' is 1 + 1  ->   N' = 2
(6)   gọi          N is 1 + 2   ->   N = 3
```

Với `is`, ta đã đưa vào một quan hệ nhạy cảm với thứ tự thực hiện các đích, và do vậy không thể bỏ qua yếu tố thủ tục trong chương trình.

Điều gì sẽ xảy ra nếu ta không sử dụng `is` trong chương trình. Chẳng hạn :

```
length1( [ ], 0 ).
length1( [ _ | Queue ], N ) :-
      length1( Queue, N1 ),
      N = 1 + N1.
```

Lúc này, nếu gọi :

```
?- length1( [ a, [ b, c ], d, e ], N ).
```

Prolog trả lời :

```
N = 1 + (1 + (1 + (1 + 0)))
Yes
```

Phép cộng do không được khởi động một cách tường minh nên sẽ không bao giờ được thực hiện. Tuy nhiên, ta có thể hoán đổi hai đích của mệnh đề thứ hai trong `length1` :

```
length1( [ ], 0 ).
length1( [ _ | Queue ], N ) :-
      N = 1 + N1,
      length1( Queue, N1 ).
```

Kết quả chạy chương trình sau khi hoán đổi vẫn y hệt như cũ. Bây giờ, ta lại có thể rút gọn mệnh đề về chỉ còn một đích :

```
length1( [ ], 0 ).
length2( [ _ | Queue ], 1 + N ) :-
     length2( Queue, N ).
```

Kết quả chạy chương trình lần này vẫn y hệt như cũ. Prolog không đưa ra trả lời như mong muốn, mà là :

```
?- length1([ a, b, c, d], N).
N = 1+ (1+ (1+ (1+0)))
Yes
```

III.3.3. Tạo sinh các số tự nhiên

Chương trình sau đây tạo sinh và liệt kê các số tự nhiên :

```
% Natural Numbers
nat(0).
nat(N) :- nat(M), N is M + 1.
```

Khi thực hiện các đích con trong câu hỏi :

```
?- nat(N), write(N), nl, fail.
```

các số tự nhiên được tạo sinh liên tiếp nhờ kỹ thuật quay lui. Sau khi số tự nhiên đầu tiên `nat(N)` được in ra nhờ `write(N)`, hằng `fail` bắt buộc thực hiện quay lui. Khi đó, luật thứ hai được vận dụng để tạo sinh số tự nhiên tiếp theo và cứ thế tiếp tục cho đến khi NSD quyết định dừng chương trình (`^C`).

Tóm tắt chương 4

- Danh sách là một cấu trúc hoặc rỗng, hoặc gồm hai phần : phần đầu là một phần tử và phần còn lại là một danh sách.
- Prolog quản lý các danh sách theo cấu trúc cây nhị phân. Prolog cho phép sử dụng nhiều cách khác nhau để biểu diễn danh sách.

  `[ Object1, Object2, ... ]`

  hoặc `[ Head | Tail ]`

  hoặc `[ Object1, Object2, ... | Others ]`

  Với `Tail` và `Others` là các danh sách.
- Các thao tác cổ điển trên danh sách có thể lập trình được là : kiểm tra một phần tử có thuộc về một danh sách cho trước không, phép ghép hai danh sách, bổ sung hoặc loại bỏ một phần tử ở đầu hoặc cuối danh sách, trích ra một danh sách con...

Bài tập chương 4

1. Viết một thủ tục sử dụng `append` để xóa ba phần tử cuối cùng của danh sách `L`, tạo ra danh sách `L1`. Hướng dẫn : `L` là phép ghép của `L1` với một danh sách của ba phần tử (đã bị xóa khỏi `L`).
2. Viết một dãy các đích để xóa ba phần tử đầu tiên và ba phần tử cuối cùng của một danh sách `L`, để trả về danh sách `L2`.
3. Định nghĩa quan hệ :

   `last_element( Object, List )`

   sao cho `Object` phải là phần tử cuối cùng của danh sách `List`. Hãy viết thành hai mệnh đề, trong đó có một mệnh đề sử dụng `append`, mệnh đề kia không sử dụng `append`.
4. Định nghĩa hai vị từ :

   `even_length( List )` và `odd_length( List )`

   được thõa mãn khi số các phân tử của danh sách `List` là chẵn hay lẻ tương ứng. Ví dụ danh sách :

   `[ a, b, c, d ]` có độ dài chẵn,

   `[ a, b, c ]` có độ dài lẽ.
5. Cho biết kết quả Prolog trả lời các câu hỏi sau :

```
?- [1,2,3] = [1|X].
?- [1,2,3] = [1,2|X].
```

```
?- [1 | [2,3]] = [1,2,X].
?- [1 | [2,3,4]] = [1,2,X].
?- [1 | [2,3,4]] = [1,2|X].
?- b(o,n,j,o,u,r) =.. L.
?- bon(Y) =.. [X,jour].
?- X(Y) =.. [bon,jour].
```

6. Viết chương trình Prolog kiểm tra một danh sách có phải là một tập hợp con của một danh sách khác không ? Chương trình hoạt động như sau :

```
?- subset2([4,3],[2,3,5,4]).
Yes
```

7. Viết chương trình Prolog để lấy ra các phần tử từ một danh sách. Chương trình cũng có thể chèn các phần tử vào một danh sách hoạt động như sau :

```
?- takeout(3,[1,2,3],[1,2]).
Yes
?- takeout(X,[1,2,3],L).
X = 1
L = [2, 3] ;
X = 2
L = [1, 3] ;
X = 3
L = [1, 2] ;
No
?- takeout(4,L,[1,2,3]).
4
L = [4, 1, 2, 3] ;
L = [1, 4, 2, 3] ;
L = [1, 2, 4, 3] ;
L = [1, 2, 3, 4] ;
No
```

8. Viết vị từ Prolog `getEltFromList(L,N,E)` cho phép lấy ra phần tử thứ N trong một danh sách. Thất bại nếu danh sách không có đủ N phần tử. Chương trình hoạt động như sau :

```
?- getEltFromList([a,b,c],0,X).
No
?- getEltFromList([a,b,c],2,X).
X = b
?- getEltFromList([a,b,c],4,X).
No
```

9. Viết chương trình Prolog tìm phần tử lớn nhất và phần tử nhỏ nhất trong một danh sách các số. Chương trình hoạt động như sau :

```
?- maxmin([3,1,5,2,7,3],Max,Min).
Max = 7
Min = 1
Yes
?- maxmin([2],Max,Min).
Max = 2
Min = 2
Yes
```

10. Viết chương trình Prolog chuyển một danh sách phức hợp, là danh sách mà mỗi phần tử có thể là một danh sách con chứa các danh sách con phức hợp khác, thành một danh sách phẳng là danh sách chỉ chứa các phần tử trong tất cả các danh sách con có thể, giữ nguyên thứ tự lúc đầu. Chương trình hoạt động như sau :

```
flatten([[1,2,3],[4,5,6]], Flatlist).
Flatlist = [1,2,3,4,5,6]
Yes
flatten([[1,[hallo,[[aloha]]],2,[],3],[4,[],5,6]],
Flatlist)
Flatlist = [1, hallo, aloha, 2, 3, 4, 5, 6]
Yes
```

11. Viết các chương trình Prolog thực hiện các vị từ xử lý tập hợp cho ở phần lý thuyết (mục II).

12. Sử dụng vị từ `forall` để viết chương trình Prolog kiểm tra hai danh sách có rời nhau (disjoint) không ? Chương trình hoạt động như sau :

```
?- disjoint([a,b,c],[d,g,f,h]).
Yes
?- disjoint([a,b,c],[f,a]).
No
```

13. Vị từ `forall(Cond, Action)` thực hiện kiểm tra sự so khớp tương ứng giữa `Cond`, thường kết hợp với vị từ `member`, và `Action`. Ví dụ dưới đây kiểm tra việc thực hiện các phép toán số học trong danh sách L là đúng đắn.

```
?- forall(member(Result = Formula, [2 = 1 + 1, 4 = 2 *
2]), Result =:= Formula).
Result = _G615
Formula = _G616
Yes
```

14. Sử dụng vị từ `forall` để viết chương trình Prolog kiểm tra một danh sách có là một tập hợp con của một danh sách khác hay không ? Chương trình hoạt động như sau :

```
?- subset3([a,b,c],[c,d,a,b,f]).
Yes
?- subset3([a,b,q,c],[d,a,c,b,f])
No
```

15. Sử dụng vị từ `append` ghép hai danh sách để viết các chương trình Prolog thực hiện các việc sau :

`prefixe(L1, L2)` danh sách `L1` đứng trước (prefixe list) danh sách `L2`.
`suffixe(L1, L2)` danh sách `L1` đứng sau (suffixe list) danh sách `L2`.
`isin(L1, L2)` các phần tử của danh sách `L1` có mặt trong danh sách `L2`.

16. Sử dụng phương pháp Quicksort viết chương trình Prolog sắp xếp nhanh một danh sách các số đã cho theo thứ tự tăng dần.

17. Đọc hiểu chương trình sau đây rồi dựng lại thuật toán :

```
/* Missionarys & Cannibals */
/* Tránh vòng lặp */
lNotExist(_,[]).
lNotExist(X,[T|Q])  :-
           X\==T, lNotExist(X,Q).
/* Kiểm tra tính hợp lý của trạng thái */
valid(MG,CG,MD,CD)  :-
           MG>=0, CG>=0, MD>=0, CD>=0, MG=0, MD>=CD.
valid(MG,CG,MD,CD)  :-
           MG>=0, CG>=0, MD>=0, CD>=0, MG>=CG, MD=0.
valid(MG,CG,MD,CD)  :-
           MG>=0, CG>=0, MD>=0, CD>=0, MG>=CG, MD>=CD.
/* Xây dựng cung và kiểm tra */
sail(1,0). sail(0,1). sail(1,1). sail(2,0). sail(0,2).
arc([left,MGi,CGi,MDi,CDi],[droite,MGf,CGf,MDf,CDf])  :-
          sail(Mis,Can),
          MGf is MGi-Mis, MDf is MDi+Mis,
          CGf is CGi-Can, CDf is CDi+Can,
          valid(MGf,CGf,MDf,CDf).
arc([right,MGi,CGi,MDi,CDi],[left,MGf,CGf,MDf,CDf])  :-
           sail(Mis,Can),
          MGf is MGi+Mis, MDf is MDi-Mis,
          CGf is CGi+Can, CDf is CDi-Can,
          valid(MGf,CGf,MDf,CDf).
/* Phép đệ quy */
```

```
cross(A,A,[A],Non).
cross(X,Y,Ch,Non)  :-
          arc(X,A), lNotExist(A,Non),
          cross(A,Y,ChAY,[A|Non]), Ch=[X|ChAY].
/* Đi qua */
traverse(X,Y,Ch)  :-
           cross(X,Y,Ch,[X]).
```

CHƯƠNG 5

Kỹ thuật lập trình Prolog

I. Nhát cắt

I.1. Khái niệm nhát cắt

Như đã thấy, một trình Prolog được thực hiện nhờ các mệnh đề và các đích. Sau đây ta sẽ xét một kỹ thuật khác của Prolog cho phép ngăn chặn sự quay lui là *nhát cắt* (cut).

Prolog tự động quay lui khi cần tìm một tìm kiếm một mệnh đề khác để thoả mãn đích. Điều này rất có ích đối với người lập trình khi cần sử dụng nhiều phương án giải quyết vấn đề . Tuy nhiên, nếu không kiểm soát tốt quá trình này, việc quay lui sẽ trở nên kém hiệu quả. Vì vậy, Prolog sử dụng kỹ thuật nhát cắt kiểm soát quay lui, hay cấm quay lui, để khắc phục khiếm khuyết này.

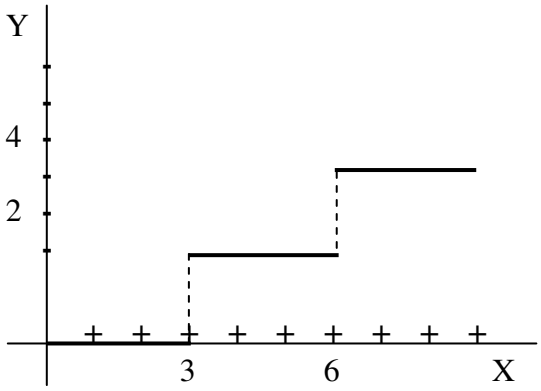
Trong ví dụ sau đây, một chương trình Prolog sử dụng kỹ thuật quay lui kém hiệu quả. Ta cần xác định các vị trí mà từ đó chương trình bắt đầu quá trình quay lui. Ta xét hàm bậc thang

Ta có ba quy tắc xác định quan hệ giữa hai trục X và Y như sau :

1. Nếu X < 3 thì Y = 0
2. Nếu X ≤ 3 và X < 6 thì Y = 2
3. Nếu X ≤ 6 thì Y = 4

Ta viết thành quan hệ nhị phân `f( X, Y )` trong Prolog như sau :

```
f( X, 0) :- X < 3.    % luật 1
f( X, 2) :- 3 =< X, X < 6.  % luật 2
f( X, 4) :- 6 =< X.   % luật 3
```



Hình I.1.Hàm bậc thang có hai bậc.

Khi chạy chương trình, giả sử rằng biến X của hàm `f( X, Y )` đã được nhận một giá trị số để có thể thực hiện phép so sánh trong thân hàm. Từ đây, xảy ra hai khả năng sử dụng kỹ thuật nhát cắt như sau :

I.2. Kỹ thuật sử dụng nhát cắt

I.2.1. Tạo đích giả bằng nhát cắt

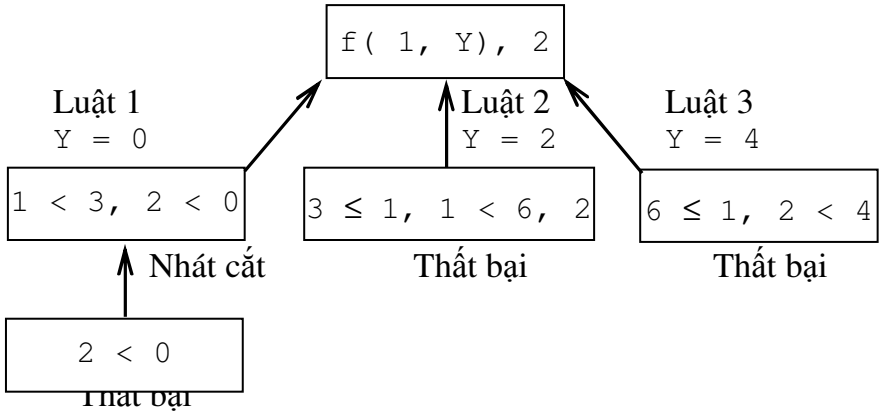
Giả sử ta đặt ra câu hỏi :

```
?- f( 1, Y ), 2 < Y.
```

Lúc này, Y nhận giá trị 0, đích thứ hai trở thành :

```
2 < 0
```

và gây ra kết quả `No` (thất bại) cho cả danh sách các đích còn lại, vì Prolog còn tiếp tục tiến hành thêm hai quá trình quay lui vô ích khác :



Hình I.2. Tại vị trí «Nhát cắt», các luật 2 và 3 đã biết trước thất bại.

Cả ba luật định nghĩa quan hệ `f` có tính chất loại trừ lẫn nhau, chỉ có duy nhất một trong chúng là có thể thành công. Người lập trình biết điều này nhưng

Prolog lại không biết, cho nên cứ tiếp tục áp dụng tất cả các luật mặc dù đi đến thất bại. Trong ví dụ trên, luật 1 được áp dụng tại vị trí «Nhát cắt» và gây ra thất bại. Để tránh sự quay lui không cần thiết bắt đầu từ vị trí này, chúng ta cần báo cho Prolog biết một cách tường minh, bằng cách sử dụng một nhát cắt, ký hiệu bởi một dấu chấm than «!» thực chất là một *đích giả* (pseudo goal) được chèn vào giữa các đích thật khác. Chương trình hàm bậc thang được viết lại như sau :

```
f( X, 0) :- X < 3, !.  % luật 1
f( X, 2) :-
     3 =< X, X < 6, !. % luật 2
f( X, 4) :-
     6 =< X.       % luật 3
```

Nhát cắt `!` sẽ cấm mọi quá trình quay lui từ vị trí xuất hiện của nó trong chương trình. Nếu bây giờ ta yêu cầu thực hiện đích :

```
?- f( 1, Y ), 2 < Y.
```

Prolog chỉ thực hiện nhánh trái nhất ứng với luật 1 trong hình trên, trả về kết quả thất bại vì xảy ra `2 < 0` mà không tiếp tục quay lui thực hiện các nhánh tương ứng với luật 2 và 3, do đã gặp nhát cắt `!`. Chương trình mới sử dụng nhát cắt chạy hiệu quả hơn chương trình cũ. Khi xảy ra thất bại, Prolog sẽ nhanh chóng dừng, mà không mất thời gian để thực hiện những việc vô ích khác. Sử dụng nhát cắt trong một chương trình làm thay đổi nghĩa thủ tục nhưng không làm thay đổi nghĩa khai báo. Tuy nhiên sau đây ta sẽ thấy rằng nhát cắt có thể làm mất đi nghĩa khai báo.

I.2.2. Dùng nhát cắt loại bỏ hoàn toàn quay lui

Giả sử bây giờ ta gọi thực hiện đích :

```
?- f( 7, Y ).
Y=4
Yes
```

Quá trình thực hiện được mô tả như sau : trước khi nhận được kết quả, về nguyên tắc, Prolog phải sử dụng cả ba luật để có quá trình xoá đích.

Thử luật 1 `7 < 3` thất bại, quay lui, thử luật 2 (nhát cắt chưa được sử dụng).

Thử luật 2 `3 ≤ 7` thoả mãn, nhưng `7 < 6` thất bại, quay lui, thử luật 3 (nhát cắt chưa được sử dụng).

Thử luật 3 `6 <= 7` thoả mãn.

Đến đây, ta lại thấy xuất hiện chương trình thực hiện kém hiệu quả. Khi xảy ra đích `X < 3` (nghĩa là `7 < 3`) thất bại, đích tiếp theo `3 ≤ X` (`3 ≤ 7`) thoả mãn, Prolog tiếp tục kiểm tra đích trong luật 3. Nhưng ta biết rằng nếu một đích

thứ nhất thất bại, thì đích thứ hai bắt buộc phải được thoả mãn vì nó là phủ định của đích thứ nhất. Việc kiểm tra lần nữa sẽ trở nên dư thừa vì đích tương ứng với nó có thể bị xoá. Như vậy việc kiểm tra đích `6 <= X` của luật 3 là không cần thiết. Với nhận xét này, ta có thể viết lại chương trình hàm bậc thang tiết kiệm hơn như sau :

Nếu `X < 3` *thì* `Y = 0`,
Nếu không, nếu `X < 6` *thì* `Y = 2`,
Nếu không `Y = 4`.

Bằng cách loại khỏi chương trình những điều kiện mà biết chắc chắn sẽ đúng, ta nhận được chương trình mới như sau :

```
f( X, 0) :- X < 3, !.
f( X, 2) :- X < 6, !.
f( X, 4).
```

Chương trình này cho kết quả tương tự hai chương trình trước đây nhưng thực hiện nhanh hơn do đã loại bỏ hoàn toàn những quay lui không cần thiết.

```
?- f(1, Y ).
Y = 0
Yes
?- f(5, Y ).
Y = 2
Yes
?- f(7, Y ).
Y = 4
Yes
```

Nhưng vấn đề gì sẽ xảy ra nếu bây giờ ta lại loại bỏ hết các nhát cắt ra khỏi chương trình ? Chẳng hạn :

```
f( X, 0) :- X < 3.
f( X, 2) :- X < 6.
f( X, 4).
```

Với lời gọi :

```
?- f( 1, Y ).
Y = 0;
Y = 2;
Y = 4;
No
```

Prolog đưa ra nhiều câu trả lời nhưng không đúng. Như vậy, việc sử dụng nhát cắt đã làm thay đổi đồng thời nghĩa thủ tục và nghĩa khai báo. Kỹ thuật nhát cắt có thể được mô tả như sau :

Ta gọi «đích cha» là đích tương ứng với phần đầu của mệnh đề chứa nhát cắt. Ngay khi gặp nhát cắt, Prolog xem rằng một đích đã được thoả mãn một cách tự động, và giới hạn sự lựa chọn các mệnh đề trong phạm vi giữa lời gọi đích cha và thời điểm thực hiện nhát cắt. Tất cả các mệnh đề tương ứng với các đích con chưa được kiểm tra so khớp giữa đích cha và nhát cắt đều được bỏ qua.

Để minh hoạ, ta xét mệnh đề có dạng :

```
H :- G1, G2, ... Gm, ! , ... , Bn.
```

Giả sử rằng mệnh đề này được khởi động bởi một đích `G` hợp nhất được với `H`, khi đó, `G` là đích cha. Cho đến khi gặp nhát cắt, Prolog đã tìm được các lời giải cho các đích con `G1`, `G2`, ... `Gm`.

Ngay sau khi thực hiện nhát cắt, các đích con `G1`, `G2`, ... `Gm` bị «vô hiệu hoá», kể cả các mệnh đề tương ứng với các đích con này cũng bị bỏ qua. Hơn nữa, do `G` hợp nhất với `H` nên Prolog không tiếp tục tìm kiếm để so khớp `H` với đầu (head) của các mệnh đề khác.

Chẳng hạn, áp dụng nguyên lý trên cho ví dụ sau :

```
C :- P, Q, R, ! S, T, U.
C :- V.
A :- B, C, D.
?- A.
```

Giả sử `A, B, C, D, P, ...` đều là các hạng. Tác động của nhát cắt khi thực hiện đích `C` như sau : quá trình quay lui xảy ra bên trong danh sách các đích `P, Q, R`, nhưng ngay khi thực hiện nhát cắt, mọi con đường dẫn đến các mệnh đề trong danh sách `P, Q, R` đều bị bỏ qua. Mệnh đề `C` thứ hai :

```
C :- V.
```

cũng bị bỏ qua. Tuy nhiên, việc quay lui vẫn có thể xảy ra bên trong danh sách các đích `S, T, U`. Đích cha của mệnh đề chứa nhát cắt là `C` ở trong mệnh đề :

```
A :- B, C, D.
```

Như vậy, nhát cắt chỉ tác động đối với mệnh đề `C`, mà không tác động đối với `A`. Việc quay lui tự động trong danh sách các đích `B, C, D` vẫn được thực hiện, độc lập với nhát cắt hiện diện trong `C`.

I.2.3. Ví dụ sử dụng kỹ thuật nhát cắt

1. Tìm số max

Xây dựng chương trình tìm số lớn `nhất trong hai số có dạng :`

```
max( X, Y, MaX )
```

trong đó, `Max = X` nếu `X` lớn hơn hoặc bằng `Y`, và `Max = Y` nếu `X` nhỏ hơn hoặc bằng `Y`. Ta xây dựng hai quan hệ như sau :

```
max( X, Y, X ) :- X >= Y.
max( X, Y, Y ) :- X < Y.
```

Hai quan hệ trên loại trừ lẫn nhau. Nếu quan hệ thứ nhất thoả mãn, thì quan hệ thứ 2 chỉ có thể thất bại và ngược lại. Áp dụng dạng điều kiện quen thuộc «nếu-thì-nếu không thì» để làm gọn chương trình lại như sau :

Nếu X ≥ Y thì Max = X,
Nếu không thì Max = Y.

Sử dụng kỹ thuật nhát cắt, chương trình được viết lại như sau :

```
max( X, Y, X ) :- X >= Y, !.
max( X, Y, Y ).
```

2. Kiểm tra một phần tử có thuộc danh sách đã cho không

Ta đã xây dựng quan hệ :

```
membre( X, L).
```

để kiểm tra phần tử `X` có nằm trong danh sách `L` hay không. Chương trình như sau :

```
membre( X, [X | L]).
membre( X, [X | L]) :- membre( X, L).
```

Tuy nhiên, chương trình này hoạt động một cách «không đơn định». Nếu `X` xuất hiện nhiều lần trong danh sách, thì bất kỳ phần tử nào bằng `X` cũng được tìm thấy. Bây giờ ta chuyển `membre` thành một quan hệ đơn định chỉ tác động đối với phần tử `X` đầu tiên. Việc thay đổi rất đơn giản như sau : chỉ việc cấm quay lui ngay khi `X` được tìm thấy, nghĩa là khi mệnh đề đầu tiên được thoả mãn :

```
membre( X, [ X | L ]) :- !.
membre( X, [ X | L ]) :- membre( X, L).
```

Khi đó, trong ví dụ sau, Prolog chỉ đưa ra một lời giải :

```
?- membre( X, [a, a, b, c]).
X = a ;
No
```

3. Thêm một phần tử vào danh sách mà không bị trùng lắp

Thông thường, khi muốn thêm một phần tử mới, chẳng hạn `X`, vào danh sách `L`, người ta muốn trước đó, `L` không chứa phần tử này. Giả sử quan hệ cần xây dựng :

```
ajoute( X, L, L1)
```

có X là phần tử mới cần thêm vào danh sách L, L1 là kết quả có chứa đúng một X. Ta lập luận như sau :

Nếu X thuộc danh sách L, thì L1 = L,
Nếu không, L1 là L đã được thêm X vào.

Cách đơn giản nhất là chèn phần tử X vào ngay đầu danh sách sao cho nó là phần tử đầu (head) của L1. Ta có chương trình như sau :

```
ajoute( X, L, L) :- membre( X, L), !.

ajoute( X, L, [ X | L ] ).
```

Sau đây là các vận dụng chương trình :

```
?- ajoute( a, [ b, c ], L).
L = [ a, b, c ]

?- ajoute( X, [ b, c ], L).
L = [ b, c ]
X = b

?- ajoute( a, [ b, c, X ], L).
X = _G333
L = [a, b, c, _G333]

?- ajoute( a, [ a, b, c ], L).
L = [a, b, c];
```

Trong ví dụ này, nhờ sử dụng kỹ thuật nhát cắt, người lập trình dễ dàng thêm một phần tử mới vào danh sách mà không làm trùng lặp phần tử đó. Nếu không sử dụng kỹ thuật nhát cắt, việc thêm một phần tử mới vào một danh sách có thể làm trùng lặp phần tử.

Như vậy, kỹ thuật nhát cắt không những làm tối ưu hiệu quả lập trình, mà còn rất cần thiết để đặc tả đúng đắn mối quan hệ giữa các đối tượng.

4. Sử dụng nhát cắt để phân loại dữ liệu

Giả sử ta cần quản lý một CSDL chứa kết quả các trận đấu của các hội viên một câu lạc bộ quần vợt. Các trận đấu không được sắp xếp một cách có hệ thống, mà mỗi hội viên có thể đấu với bất cứ ai. Kết quả các trận đấu được biểu diễn bởi các sự kiện như sau :

```
bat( tom, jim).
bat( ann, tom).
bat( pat, jim).
```

Ta cần định nghĩa quan hệ :

```
classe(Player, CategorY ).
```

để phân thứ hạng cho mỗi người chơi quần vợt trong ba hạng như sau :

`champion`	người luôn thắng trong tất cả các trận đấu
`combative`	người có cả bàn thắng và có cả bàn thua
`dilettante`	người luôn thua trong tất cả các trận đấu

Từ kết quả những trận đấu đã có được cho trong các sự kiện, ta thấy Ann và Pat được xếp hạng quán quân (champion), Tom được xếp hạng trung bình (combative), còn Jim thì được xếp hạng yếu kém (dilettante). Ta có thể dễ dàng xây dựng các luật xếp hạng như sau :

X được xếp hạng trung bình nếu
tồn tại Y sao cho X thắng Y, và
tồn tại Z sao cho Z thắng X.

X được xếp hạng quán quân nếu
X thắng Y, và
X không bị thua bất kỳ đối thủ nào.

Luật xếp hạng quán quân có chứa phép phủ định (not) mà cho đến lúc này, ta chưa tìm hiểu cách biểu diễn như thế nào trong Prolog. `Luật` xếp hạng yếu kém cũng xây dựng tương tự luật xếp hạng quán quân. Ta có thể sử dụng sơ đồ *if-then-else* để xử lý `đồng thời` hai tình huống như sau :

Nếu X thắng và X bị thua khi đấu với bất kỳ ai
thì X được xếp hạng trung bình
nếu không, nếu X thắng bất kỳ ai
thì X được xếp hạng quán quân
nếu không, nếu X luôn bị thua
thì X được xếp hạng yếu kém.

Từ sơ đồ trên ta có thể chuyển sang Prolog sử dụng kỹ thuật nhát cắt để xử lý khả năng loại trừ nhau giữa ba thứ hạng.

```
classe( X, combative) :-
     bat( X, _ ),
     bat( _, X ), !.

classe( X, champion) :-
     bat( X, _ ), !.

classe( X, dilettante) :-
     bat( _, X ).
```

Chú ý rằng không nhất thiết phải sử dụng nhát cắt trong mệnh đề `champion` vì bản chất của ba thứ hạng.

I.3. Phép phủ định

I.3.1. Phủ định bởi thất bại

Trong Prolog, ta có thể nói được câu : «Marie thích tất cả loài động vật trừ loài rắn» hay không ?

Đối với vế thứ nhất, ta có thể dễ dàng dịch ra thành : *Dù X là gì, Marie thích X nếu X là loài động vật :*

```
enjoy( marie, X ) :-
     animal( X ).
```

Tuy nhiên cần loại trừ loài rắn. Lúc này ta cần dịch ra như sau :

Nếu X là loài rắn, thì «Marie thích X» là sai,
Nếu không, nếu X là loài động vật thì Marie thích X.

Những gì không đúng thì có thể sử dụng đích đặc biệt `fail` (thất bại) để luôn luôn sai, và cũng làm cho đích cha thất bại. Chương trình được viết lại như sau :

```
enjoy( marie, X ) :-
     serpent( X ), !, fail.

enjoy( marie, X ) :-
     animal( X ).
```

Luật thứ nhất xử lý tình huống Marie không thích loài rắn : nếu X là loài rắn, thì nhát cắt sẽ ngăn sự quay lui (và do đó, luật thứ hai không được thực hiện), và đích `fail` sẽ gây ra thất bại. Ta có thể sử dụng dấu `;` để viết cô đọng hai luật thành một luật như sau :

```
enjoy( marie, X ) :-
     serpent( X ), !, fail;
     animal( X ).
```

Một cách tương tự, ta định nghĩa quan hệ khác nhau :

```
diffĩrent( X, Y )
```

thoả mãn nếu `X` và `Y` là khác nhau. Do sự khác nhau có thể được diễn giải theo nhiều cách nên ta cần chỉ rõ như sau :

- X và Y không phải là các trực hằng (literal) đồng nhất,
- X và Y không thể khớp với nhau,
- Các giá trị của các biểu thức số học X và Y không thể bằng nhau.

Ta nói rằng X và Y khác nhau do chúng không thể khớp được với nhau :

Nếu X và Y là đồng nhất, thì `diffĩrent( X, Y )` *thất bại,*
Nếu không, `diffĩrent( X, Y )` *thành công.*

Ta sử dụng nhát cắt và đích `fail` để viết quan hệ này thành hai luật :

```
diffĩrent( X, X ) :- !, fail.
```

```
diffĩrent( X, Y ).
```

Hoặc viết lại thành một luật như sau :

```
diffĩrent( X, Y ) :-
     X = Y, !, fail;
     true.
```

Chú ý rằng đích `true` (đúng) luôn luôn thành công.

Từ đây, ta có thể định nghĩa vị từ `not(Goal)` cho phép kiểm tra đích không thoả mãn như sau :

Nếu `Goal` *thoả mãn, thì* `not(Goal)` *thất bại,*
Nếu không, `not(Goal)` *thành công.*

Chương trình Prolog :

```
not( P ) :-
     P, !, fail;
     true.
```

Hầu hết các phiên bản Prolog hiện nay đều có vị từ `not`

```
not(2 = 3).
Yes

?- not(2 = 2).
No
```

Sử dụng vị từ `not`, ta có thể định nghĩa lại các quan hệ `enjoy`, `diffĩrent` và `classe` như sau :

```
enjoy( marie, X ) :-
     animal( X ),
     not (serpent( X )).

diffĩrent( X, Y ) :-
     not( X = Y ).

classe( X, combatif) :-
     bat( X, _ ),
     bat( _ , X ).

classe( X, champion) :-
     bat( X _ ),
     not bat( _ , X ).

classe( X, dilettante) :-
     bat( _ , X ),
     not bat( X, _ ).
```

I.3.2. Sử dụng kỹ thuật nhát cắt và phủ định

Ưu điểm của kỹ thuật nhát cắt có thể tóm tắt như sau :

1. Nâng cao tính hiệu quả của một chương trình nhờ nguyên tắc thông báo một cách tường minh cho Prolog tránh không đi theo những con đường dẫn đến thất bại.
2. Kỹ thuật nhát cắt cho phép xây dựng những luật có tính chất loại trừ nhau có dạng :

   Nếu điều kiện P xảy ra thì kết luận là Q,
   Nếu không, thì kết luận là R.

Tuy nhiên sử dụng nhát cắt có thể làm mất sự tương ứng giữa nghĩa khai báo và nghĩa thủ tục của một chương trình. Nếu trong chương trình không xuất hiện nhát cắt, thì việc thay đổi thứ tự các mệnh đề và các đích chỉ làm ảnh hưởng đến hiệu quả chạy chương trình mà không làm thay đổi nghĩa khai báo. Còn khi có mặt nhát cắt trong một chương trình, thì lại xảy ra vấn đề, lúc này có thể có thể nhiều kết quả khác nhau. Ví dụ :

```
p :- a, b.
p :- c.
```

Xét về mặt nghĩa khai báo, chương trình trên có nghĩa : p đúng nếu và chỉ nếu cả a và b đều đúng, hoặc c đúng. Từ đó ta xây dựng biểu thức lôgich như sau :

```
p ⇔ (a ∧ b) ∨ c
```

Nghĩa khai báo không còn đúng nữa nếu ta thay đổi mệnh đề thứ nhất bằng cách thêm vào một nhát cắt :

```
p :- a, !, b.
p :- c.
```

Biểu thức lôgich tương ứng như sau :

```
p ⇔ (a ∧ b) ∨ (~a ∧ c)
```

Nếu ta đảo thứ tự hai mệnh đề :

```
p :- c.
p :- a, !, b.
```

thì ta lại có cùng nghĩa như ban đầu :

```
p ⇔ c ∨ (a ∧ b)
```

Người ta phải thận trọng khi sử dụng kỹ thuật nhát cắt do nhát cắt làm thay đổi nghĩa thủ tục và làm tăng nguy cơ xảy ra sai sót trong chương trình. Như đã xét trong các ví dụ trước đây, việc loại bỏ nhát cắt có thể làm thay đổi nghĩa khai báo của một chương trình. Tuy nhiên trong một số trường hợp, nhát cắt không

ảnh hưởng đến nghĩa khai báo. Người ta gọi những nhát cắt không làm thay đổi ngữ nghĩa của chương trình là *nhát cắt xanh* (green cuts). Đứng trên quan điểm lập trình dễ đọc và dễ hiểu (readability), các nhát cắt xanh là an toàn và người ta thường hay sử dụng chúng. Thậm chí, người ta có thể bỏ qua sự có mặt của chúng khi đọc chương trình. Người ta nói nhát cắt xanh làm *rõ ràng* (explicit) *tính tiền định* (determinism) vốn không rõ ràng (implicit). Thông thường nhát cắt xanh được đặt ngay sau phép kiểm tra tiền định.

Ví dụ sử dụng nhát cắt xanh tìm số min :

```
minimum(X, Y, X) :-
     X =< Y, !.
minimum(X, Y, Y) :-
     X > Y, !.
```

Ví dụ sử dụng nhát cắt xanh kiểm tra kiểu của cây nhị phân các số nguyên :

```
int_bin_tree(ab(X,G,D)) :-
     integer(X),
     int_bin_tree(G),
     int_bin_tree(D).
int_bin_tree(X) :-
     integer(X).
```

Trong các trường hợp khác, các nhát cắt ảnh hưởng đến nghĩa khai báo được gọi là *nhát cắt đỏ* (red cuts). Sự có mặt của các nhát cắt đỏ thường làm cho chương trình trở nên khó đọc, khó hiểu. Để sử dụng được chúng, NSD phải hết sức chú ý. Ví dụ sử dụng nhát cắt đỏ tìm số min thay đổi ngữ nghĩa :

```
minimum_cut( X, Y, X ) :-
     X =< Y, !.
minimum_cut( X, Y, Y ).
```

Trong một số trường hợp, một câu hỏi có thể không liên quan đến ngữ nghĩa của chương trình. Ví dụ vị từ kiểm tra một phần tử có thuộc danh sách không :

```
member_cut(X, [ X | _ ] ) :- !.
member_cut(X, [ _ | L ] ) :- member_cut(X, L ).
```

Với câu hỏi `member_cut(X, [ 1, 2, 3 ] )` sẽ không cho kết quả `X = 2`.

```
?- member_cut(X, [ 1, 2, 3 ] ).
X = 1 ;
No
```

Thông thường, đích `fail` được dùng cặp đôi với nhát cắt (cut-fail). Người ta thường định nghĩa phép phủ định một đích (`not`) bằng cách gây ra sự thất bại của đích này, thực chất là cách sử dụng nhát cắt có hạn chế. Để chương trình dễ hiểu

hơn, thay vì sử dụng cặp đôi cut-fail, người ta sử dụng `not`. Tuy nhiên, phép phủ định `not` cũng không phải không gây ra những phiền phức cho người dùng. Nhiều khi sử dụng not không hoàn toàn chính xác với phép phủ định trong Toán học. Chẳng hạn nếu trong chương trình có định nghĩa quan hệ `man`, mà ta đưa ra một câu hỏi đại loại như :

```
?- not( man( marie)).
```

Khi đó, Prolog sẽ trả lời `No` nếu đã có định nghĩa `man( marie)`, trả lời `Yes` nếu chưa có định nghĩa như vậy. Tuy nhiên, khi trả lời `No`, không phải Prolog nói rằng «Marie không phải là một người», mà nói rằng «Không tìm thấy trong chương trình thông tin để chứng minh Marie là một người». Khi thực hiện phép `not`, Prolog không chứng minh trực tiếp mà tìm cách chứng minh điều ngược lại. Nếu chứng minh được, Prolog suy ra rằng đích `not` thành công. Cách lập luận như vậy được gọi là *giả thuyết về thế giới khép kín* (hypothesis of the enclosed world). Theo giả thuyết này, thế giới khép kín có nghĩa là những gi tồn tại (đúng) đều nằm trong chương trình hoặc được suy ra từ chương trình. Những gì không nằm trong chương trình, hoặc không thể suy ra từ chương trình, thì sẽ là không đúng (sai), hay điều phủ định là đúng. Vì vậy, cần chú ý khi sử dụng phủ định do thông thường, người ta đã không giả thiết rằng thế giới là khép kín. Trong chương trình, do thiếu khai báo mệnh đề :

```
man( marie).
```

nên Prolog không chứng minh được rằng Marie là một người.

Sau đây là một ví dụ khác sử dụng phép phủ định `not` :

```
r( a).

q( b).

p( X ) :- not( r( X )).
```

Nếu đặt câu hỏi :

```
?- q( X ), p( X ).
```

thì Prolog sẽ trả lời :

```
X=b
Yes
```

Nhưng nếu đặt câu hỏi :

```
?- p( X ), q( X ).
```

thì Prolog sẽ trả lời :

```
No
```

Để hiểu được vì sao cùng một chương trình nhưng với hai cách đặt câu hỏi khác nhau lại có hai cách trả lời khác nhau, ta cần tìm hiểu cách Prolog lập luận.

Trong trường hợp thứ nhất, biến `X` được ràng buộc giá trị là `b` khi thực hiện đích `q( X )`. Tiếp tục thực hiện đích con `p( X )`, nhờ ràng buộc `X=b`, đích `not( r( X ))` thoả mãn vì đích `r( b )` không thoả mãn, Prolog trả lời `Yes`.

Trái lại trong trường hợp thứ hai, do Prolog thực hiện đích con `p( X )` trước nên sự thất bại của `not( r( X ))`, tức `r( X )` thành công với ràng buộc `X=a`, dẫn đến câu trả lời `No`.

II. Sử dụng các cấu trúc

Kiểu dữ liệu cấu trúc, danh sách, kỹ thuật so khớp, quay lui và nhát cắt là những điểm mạnh trong lập trình Prolog. Chương này sẽ tiếp tục trình bày một số ví dụ tiêu biểu về :

- Truy cập thông tin cấu trúc từ một cơ sở dữ liệu.
- Mô phỏng một ôtômat hữu hạn không đơn định và máy Turing.
- Lập kế hoạch đi du lịch
- Bài toán tám quân hậu

Đồng thời, ta cũng trình bày cách Prolog trừu tượng hoá dữ liệu.

II.1. Truy cập thông tin cấu trúc từ một cơ sở dữ liệu

Sau đây là một ví dụ cho phép biểu diễn và thao tác các dữ liệu cấu trúc. Từ đó, ta cũng hiểu cách sử dụng Prolog như một ngôn ngữ truy vấn cơ sở dữ liệu.

Trong Prolog, một cơ sở được biểu diễn dưới dạng một tập hợp các sự kiện. Chẳng hạn, một cơ sở dữ liệu về các gia đình sẽ mô tả mỗi gia đình (`family`) như một mệnh đề. Mỗi gia đình sẽ gồm ba phần tử lần lượt : chồng, vợ (`individual`) và các con (`children`). Do các phần tử này thay đổi tuy theo từng gia đình, nên các con sẽ được biểu diễn bởi một danh sách để có thể nhận được một số lượng tuỳ ý số con. Mỗi người trong gia đình được biểu diễn bởi bốn thành phần : tên, họ, ngày tháng năm sinh và việc làm. Thành phần việc làm có thể có giá trị "thất nghiệp" (`inactive`), hoặc chỉ rõ tên cơ quan công tác và thu nhập theo năm.

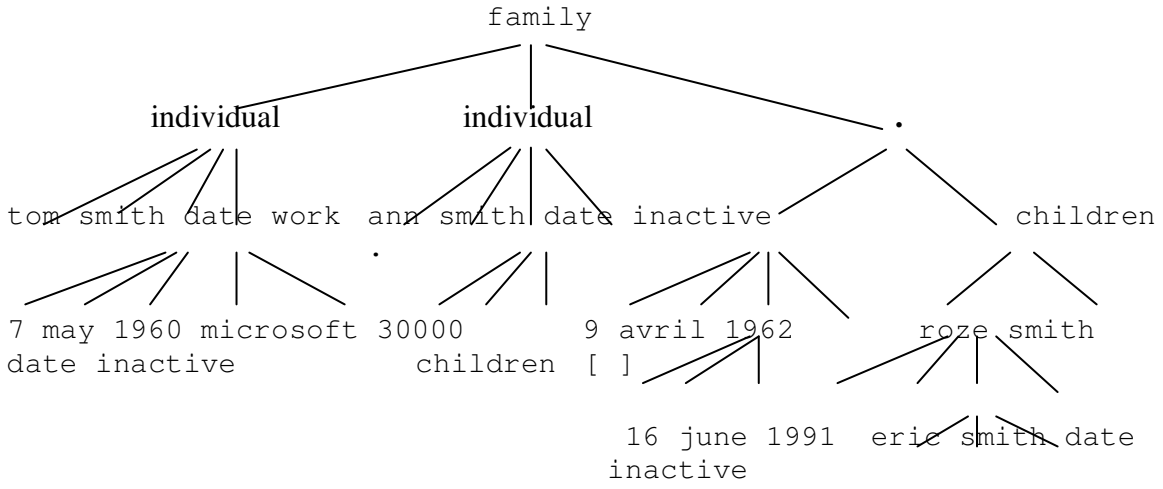
Giả sử cơ sở dữ liệu chứa mệnh đề đầu tiên như sau :

```
family(
   individual( tom, smith, date(7, may, 1960),
   work(microsoft, 30000) ),
   individual( ann, smith, date(9, avril, 1962),
   inactive),
   [individual( roze, smith, date(16, june, 1991),
   inactive),
    individual( eric, smith, date(23, march, 1993),
   inactive) ] ).
```

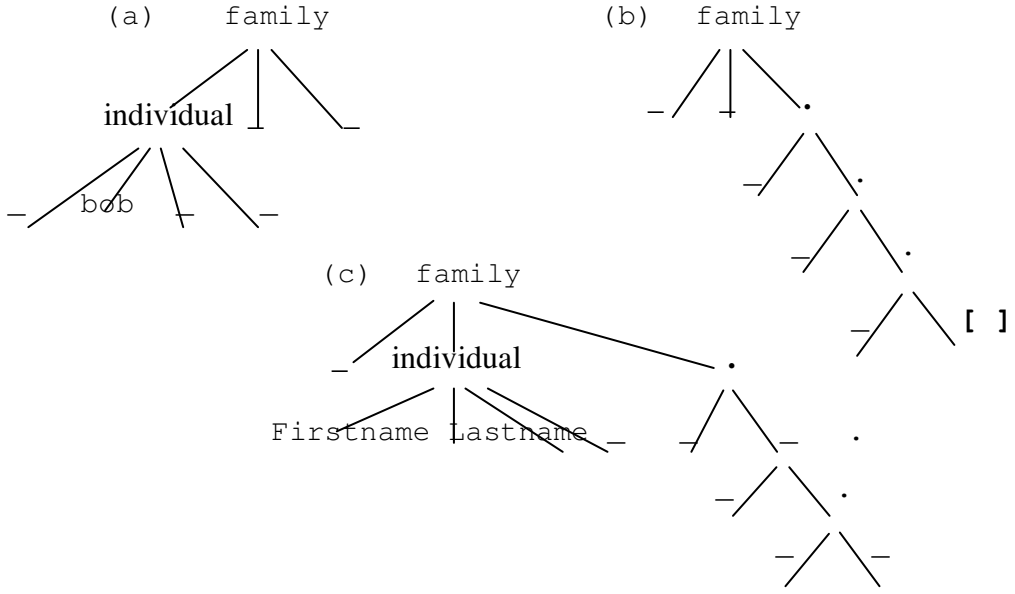
Dữ liệu về những gia đình khác tiếp tục được bổ sung dưới dạng các mệnh đề tương tự. Hình 5.1 dưới đây minh hoạ cách tổ chức cơ sở dữ liệu.

Prolog là một ngôn ngữ rất thích hợp cho việc khôi phục thông tin : người sử dụng có thể gọi các đối tượng mà không nhất thiết chỉ rõ tất cả các thành phần. Người sử dụng chỉ cần chỉ ra *cấu trúc* của các đối tượng mà họ quan tâm một cách tượng trưng, không cần phải chỉ ra hết. Hình 1.2 minh hoạ những cấu trúc như vậy. Ví dụ, để biểu diễn những gia đình dòng họ Smith, trong Prolog viết :

```
family( individual( _ , smith, _ , _ ), _ , _ )
```



Hình II.1. *Cấu trúc cây biểu diễn thông tin về một gia đình*



Hình II.2. tính chất cấu trúc của các đối tượng Prolog cho phép biểu diễn : (a) một gia đình Smith nào đó ; (b) những gia đình có đúng ba con ; (c) những gia đình

có ít nhất ba con. Riêng trường hợp (c) còn cho phép biểu diễn tên của người vợ nhờ sự ràng buộc các biến `Firstname` *và* `Lastname`.

Những dấu gạch dưới dòng như đã biết là các biến nặc danh, người sử dụng không cần quan tâm đến giá trị của chúng. Một cách tương tự, những gia đình có ba con được biểu diễn bởi :

```
family( _ , _ , [ _ , _ , _ ] )
```

Ta cũng có thể đặt câu hỏi tìm những người vợ trong những gia đình có ít nhất ba con :

```
?- family( _ , individual( Firstname, Lastname, _ , _
  ), [ _ , _ , _ | _ ] ).
```

Những ví dụ trên đây chỉ ra rằng ta có thể biểu diễn các đối tượng bởi cấu trúc của chúng mà không cần quan tâm đến nội dung, bằng cách bỏ qua những tham đối vô định.

Sau đây là một số mệnh đề được đưa thêm vào cơ sở dữ liệu các gia đình để có thể đặt các câu hỏi vấn tin khác nhau (có thể bổ sung thêm các gia đình mới bởi mệnh đề `family`) :

```
husban( X ) :-          % X là một người chồng
   family( X , _ , _ ).
wife( X ) :-            % X là một người vợ
   family( _, X , _ ).
chidren( X ) :-         % X là một người con, chú ý các tên biến chữ hoa
   family( _, _ , Chidren ),
   ismember( X, Chidren ).
ismember( X, [ X | L ] ).   % có thể sử dụng mệnh đề member của
   Prolog
ismember( X, [ Y | L ] ) :-
   ismember( X, L ).
exist( Individual ) :-      % mọi thành viên của gia đình
   husban( Individual ) ;
   wife( Individual ) ;
   chidren( Individual ).
dateofbirth( individual( _ , _, Date , _ ), Date ).
salary( individual( _ , _, _ , work( _ , S ) ), S ). %
   thu nhập của người lao động
salary( individual( _ , _, _ , inactive ), 0 ). % người
   không có nguồn thu nhập
```

Bây giờ ta có thể đặt các câu hỏi như sau :

1. Tìm tên họ của những người có mặt trong cơ sở dữ liệu :

```
?- exist( individual( Firstname, Lastname, _ , _ ) ).
```

2. Tìm những người con sinh năm 1991 :

```
?- chidren( X ), dateofbirth( X, date( _ , _ , 1991 ) ).
```

3. Tìm những người vợ có việc làm :

```
?- wife( individual( Firstname, Lastname, _ , work( _ , _ ) ) ).
```

4. Tìm những người không có việc làm sinh trước năm 1975 :

```
?-                              exist( individual(
  Firstname, Lastname, date( _ , _ , Year ), inactive
  ) ),
  Year < 1975.
```

5. Tìm những người sinh trước năm 1975 có thu nhập dưới 10000 :

```
?-                              exist( Individual ),
  dateofbirth( Individual, date( _ , _ , Year ) ),
  Year < 1975,
  salary( Individual, Salary ),
  Salary < 10000.
```

6. Tìm những gia đình có ít nhất ba con :

```
?- family( individual( _, Name, _ , _ ), _, [ _, _, _ | _ ] ).
```

Để tính tổng thu nhập của một gia đình, ta có thể định nghĩa một quan hệ nhị phân cho phép tính tổng các thu nhập của một danh sách những người đang có việc làm dạng :

```
total( List_of_ individual, Sum_of_ salary )
```

Ta viết trong Prolog như sau :

```
total( [ ], 0 )          % danh sách rỗng

total( [ Individual | List ], Sum ) :-
   salary( Individual, S ),       % S là thu nhập của người đầu tiên
   total( List, Remain ),    % Remain là thu nhập của tất cả những người còn lại
   Sum is S + Remain.
```

Như vậy, tổng thu nhập của một gia đình được tính bởi câu hỏi :

```
?- family( Husban, Wife, Chidren ),
   total( [ Husban, Wife | Chidren ], Income ).
```

Các phiên bản Prolog đều có thể tính độ dài (`length`) của một danh sách (xem mục III chương 1 trước đây, ta cũng đã tìm cách xây dựng quan hệ này).

Bây giờ ta có thể áp dụng để tìm những gia đình có nguồn thu nhập nhỏ hơn 5000 tính theo đầu người :

```
?- family( Husban, Wife, Chidren ),
   total( [ Husban, Wife | Chidren ], Income )
   length( [ Husban, Wife | Chidren ], N ),
   Income / N < 5000.    % N là số người trong một gia đình
```

II.2. Trừu tượng hoá dữ liệu

Trừu tượng hoá dữ liệu (data abstraction) được xem là cách tổ chức tự nhiên (một cách có thứ cấp) những thành phần khác nhau trong cùng những đơn vị thông tin, sao cho về mặt ý niệm, người sử dụng có thể hiểu được cấu trúc bên trong. Chương trình phải dễ dàng truy cập được vào từng thành phần dữ liệu. Một cách lý tưởng thì người sử dụng không nhìn thấy được những chi tiết cài đặt các cấu trúc này, người sử dụng chỉ quan tâm đến những đối tượng và quan hệ giữa chúng. Với mục đích đó, Prolog phải có cách biểu diễn thông tin phù hợp.

Để tìm hiểu cách Prolog giải quyết, ta quay lại ví dụ cơ sở dữ liệu gia đình trong mục trước đây. Mỗi gia đình là một nhóm các thông tin khác nhau về bản chất, mỗi người hay mỗi gia đình được xử lý như một đối tượng độc lập.

Giả thiết rằng mỗi gia đình được biểu diễn như Hình II.1. Bây giờ ta tiếp tục định nghĩa các quan hệ để có thể tiếp cận đến các thành phần của gia đình mà không cần biết chi tiết. Những quan hệ này được gọi là các *bộ chọn* (selector), vì chúng chọn những thành phần nào đó. Mỗi bộ chọn sẽ có tên là tên thành phần mà nó chọn ra, và có hai tham đối : đối tượng chứa thành phần được chọn và bản thân thành phần đó :

```
selector_relation( Object, Selected_component )
```

Sau đây là một số ví dụ về các bộ chọn :

```
husban( family( Husban, _ , _ ), Husban ).
wife( family( _ , Wife, _ ), Wife ).
chidren( family( _ , _ , ChidrenList ), ChidrenList ).
```

Ta cũng có thể định nghĩa những bộ chọn chọn ra những người con đặc biệt như con trưởng, con út và con thứ N trong gia đình :

```
eldest( Family, Eldest ) :-      % người con trưởng
   chidren(Family, [ Eldest | _ ] ).

cadet( Family, Eldest ) :-       % người con út
   chidren( Family, [ Eldest | _ ] ).
```

Chọn ra một người con bất kỳ nào đó :

```
% người con thứ N trong gia đình
```

```
nth_child( N, Family, Chidren ) :-
   chidren( Family, ChidrenList ),
   % phần tử thứ N của một danh sách
   nth_member( N, ChidrenList, Chidren ).
```

Từ biểu diễn cấu trúc minh hoạ trong `Hình II.1`, sau đây là một số bộ chọn nhận tham đối là một thành viên trong gia đình (`individual`) :

```
lastname( individual( _ , Lastname, _ , _ ), Lastname ).
                                          % tên gia đình (họ)

firstname( individual( Firstname, _ , Wife, _ ),
Firstname ).                              % tên riêng

born( individual( _ , _ , Date, _ ), Date ).    % ngày sinh
```

Làm cách nào để có thể áp dụng các bộ chọn ? Mỗi khi các bộ chọn đã được định nghĩa, ta không cần quan tâm đến cách biểu diễn những thông tin có cấu trúc. Để tạo ra và để thao tác trên những thông tin cấu trúc, chỉ cần biết tên các bộ chọn và sử dụng chúng trong chương trình. Với phương pháp này, các biểu diễn phức tạp cấu trúc dữ liệu sẽ dễ dàng hơn so với phương pháp mô tả đã xét.

Ví dụ, người sử dụng không cần biết những người con trong gia đình được lưu giữ trong một danh sách. Giả sử rằng ta muốn hai người con Johan Smith và Eric Smith cùng thuộc một gia đình, và Eric là em thứ hai của Johan. Ta có thể sử dụng bộ chọn để định nghĩa hai cá thể, được gọi là `Individual1` và `Individual2`, và định nghĩa gia đình như sau :

```
% Johan Smith
lastname( Individual1, smith ), firstname( Individual1,
johan ).

% Eric Smith
lastname( Individual2, smith ), firstname( Individual1,
    eric ),
    husban( Family, Individual1 ).

nth_child( 2, Family, Individual2 ).
```

Việc sử dụng các bộ chọn làm thay đổi dễ dàng một chương trình Prolog. Giả sử ta muốn thay đổi dữ liệu của một chương trình, ta chỉ cần định nghĩa lại các bộ chọn, phần còn lại của chương trình vẫn hoạt động như cũ.

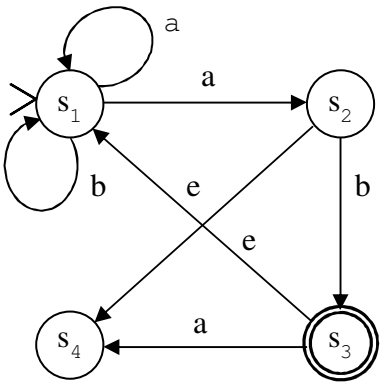
II.3. Mô phỏng ôtômat hữu hạn

Ví dụ sau đây minh hoạ cách Prolog biểu diễn các mô hình toán học trừu tượng.

II.3.1. Mô phỏng ôtômat hữu hạn không đơn định

Một *ođtômat hữu hạn không đơn định* (Non-deterministic Finite Automaton, viết tắt NFA) là một máy trừu tượng có thể đọc một *câu vào* (input string) là một xâu (hay chuỗi) ký tự nào đó và có thể quyết định có *thừa nhận* (accept) hay *không thừa nhận* (rejecting). Ôtômat có một số hữu hạn *trạng thái* (state) và luôn ở một trạng thái nào đó để có thể *chuyển tiếp* (transition) qua một trạng thái khác sau khi đọc (thừa nhận) một ký hiệu (symbol) hay ký tự thuộc một *bảng ký tự* (alphabet hay set of characters) hữu hạn nào đó. Một xâu đã cho được gọi là *được thừa nhận* bởi ôtômat, nếu sau khi đọc hết câu vào, ôtômat rơi vào một trong các trạng thái thừa nhận.

Người ta thường biểu diễn ođtômat hữu hạn bởi một đồ thị định hướng mô tả các chuyển tiếp trạng thái có thể. Mỗi cung định hướng của đồ thị được gắn nhãn là ký tự sẽ đọc. Mỗi nút của đồ thị là một trạng thái, trong đó, *trạng thái đầu* (initial state) được đánh dấu bởi **>**, và các *trạng thái thừa nhận* (accepted state) được đánh dấu bởi đường kép.



Hình II.3. Một ođtômat hữu hạn không đơn định bốn trạng thái.

Hình 5.3 minh hoạ một ôtômat hữu hạn không đơn định có bốn trạng thái `s1`, `s2`, `s3` và `s4`, trong đó, `s1` là trạng thái đầu và ôtômat chỉ có một trạng thái thừa nhận duy nhất là `s3`. Chú ý ôtômat có hai chuyển tiếp nối vòng (chu kỳ) tại trạng thái `s1` (nghĩa là ôtômat không thay đổi trạng thái sau khi đọc xong hoặc ký tự `a`, hoặc ký tự `b`).

Mỗi chuyển tiếp của ôtômat được xác định bởi một quan hệ giữa trạng thái hiện hành, ký tự sẽ đọc và trạng thái sẽ đạt tới. Chú ý rằng mỗi chuyển tiếp có thể không đơn định. Trong Hình II.3, từ trạng thái `s1`, sau khi đọc ký tự `a`, ôtômat có

thể rơi vào hoặc trạng thái s_1, hoặc trạng thái s_2. Ta cũng thấy một số cung có nhãn e (câu rỗng), tương ứng với "chuyển tiếp epsilon", ký hiệu e-chuyển tiếp. Những cung này mô tả sự chuyển tiếp "không nhìn thấy được" của ôtômat : ôtômat chuyển qua một trạng thái mới khác mà không hề đọc một ký tự nào. Nghĩa là phần câu vào vẫn không thay đổi, nhưng ôtômat đã thay đổi trạng thái.

Người ta nói ôtômat thừa nhận câu vào nếu tồn tại một dãy các chuyển tiếp trong đồ thị sao cho :

1. Lúc đầu, ôtômat ở trạng thái đầu (ví dụ s_1).
2. Ôtômat kết thúc việc đoán nhận câu vào và ở trạng thái thừa nhận (s_3).
3. Các nhãn trên các cung của con đường chuyển tiếp từ trạng thái đầu đến trạng thái thừa nhận tương ứng với câu vào là xâu đã đọc.

Trong quá trình đoán nhận câu vào, ôtômat quyết định lựa chọn một trong số các chuyển tiếp có thể để tiếp tục. Đặc biệt, ôtômat có thể thực hiện hay không thực hiện một e-chuyển tiếp, nếu trạng thái hiện hành cho phép. Ôtômat không thừa nhận câu vào nếu nó không rơi vào trạng thái thừa nhận dù đã đọc hết câu vào, hoặc không còn khả năng tiếp tục chuyển tiếp mà câu vào chưa kết thúc, hoặc có thể bị quẩn vô hạn.

Như đã biết, các ôtômat hữu hạn không đơn định trừu tượng có một tính chất thú vị : tại mỗi thời điểm, ôtômat có khả năng lựa chọn, trong số các chuyển tiếp có thể, một chuyển tiếp "tốt nhất" để thừa nhận câu vào.

Chẳng hạn, ôtômat cho ở Hình II.3 sẽ thừa nhận các xâu `ab` và `aabaab`, nhưng không thừa nhận các xâu `abb` và `abba`. Một cách tổng quát, ôtômat thừa nhận mọi xâu kết thúc bởi `ab`, nhưng không thừa nhận các xâu khác.

Trong Prolog, một ôtômat được định nghĩa bởi ba quan hệ :

1. Một quan hệ một ngôi `satisfaction` cho phép xác định các trạng thái thừa nhận của ôtômat.
2. Một quan hệ ba ngôi `trans` cho phép xác định các trạng thái chuyển tiếp, chẳng hạn :

   ```
   trans( S1, X, S2 ).
   ```

   có nghĩa là ôtômat chuyển tiếp từ trạng thái `S1` qua trạng thái `S2` sau khi đọc ký tự `X`.
3. Một quan hệ hai ngôi `epsilon` chỉ ra phép chuyển tiếp rỗng từ trạng thái `S1` qua trạng thái `S2` :

   ```
   epsilon( S1, S2 ).
   ```

Ôtômat đã cho ở Hình II.3 được mô tả bởi các mệnh đề Prolog như sau :

```
satisfaction( s3 ).
```

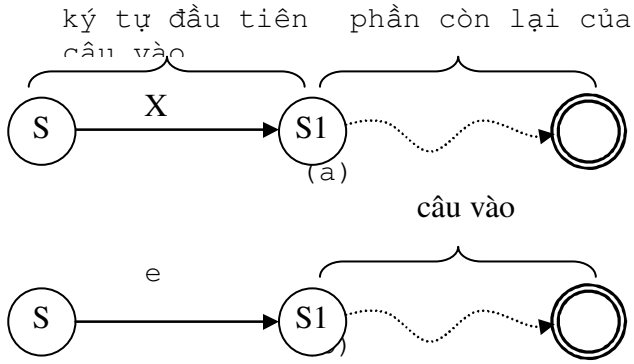
```
trans( s1, a, s1 ).
trans( s1, a, s2 ).
trans( s1, b, s1 ).
trans( s2, b, s3 ).
trans( s3, b, s4 ).

epsilon( s2, s4 ).
epsilon( s3, s1 ).
```

Để biểu diễn các xâu ký tự trong Prolog, ta sẽ sử dụng kiểu danh sách. Chẳng hạn xâu `aab` được biểu diễn bởi `[ a, b, a ]`. Xuất phát từ một câu vào, ôtômat vừa mô tả trên đây sẽ mô phỏng quá trình đoán nhận, bằng cách đọc lần lượt các phần tử của danh sách, để thừa nhận hay không thừa nhận.

Theo định nghĩa, ôtômat hữu hạn không đơn định sẽ thừa nhận câu vào nếu, xuất phát từ trạng thái đầu, sau khi đọc hết câu (xử lý hết mọi phần tử của danh sách), ôtômat rơi vào trạng thái thừa nhận. Quan hệ hai ngôi `accept` sau đây cho phép mô phỏng quá trình đoán nhận một câu vào từ một trạng thái đã cho :

```
accept( State, InputString )
```

Quan hệ `accept` là đúng nếu `State` là trạng thái đầu và `InputString` là một câu vào.



Hình II.4. Ođtômat thừa nhận câu vào :
(a) đọc ký tự đầu tiên X ; (b) thực hiện một e-chuyển tiếp.

Ba mệnh đề cho phép định nghĩa quan hệ này, tương ứng với ba trường hợp như sau :

1. Xâu rỗng `[ ]` được thừa nhận tại trạng thái `S` nếu ôtômat đang ở tại trạng thái `S` và `S` là một trạng thái thừa nhận.
2. Một xâu khác rỗng được thừa nhận tại trạng thái `S` nếu đầu đọc đang ở tại vị trí đọc ký tự đầu tiên của xâu để sau khi đọc, ôtômat chuyển qua trạng thái `S1` và xuất phát từ trạng thái `S1` này, ôtômat thừa nhận toàn bộ phần còn lại của câu vào (xem minh hoạ ở Hình II.4 (a)).

3. Một xâu khác rỗng được thừa nhận tại trạng thái `S` nếu ôtômat có thể thực hiện một e-chuyển tiếp từ trạng thái `S` qua trạng thái `S1` và xuất phát từ trạng thái `S1` này, ôtômat thừa nhận toàn bộ phần còn lại của câu vào (xem minh hoạ ở Hình II.4 (b)).

Ta có thể viết trong Prolog như sau :

```
accept( S, [ ] ) :-          % thừa nhận xâu rỗng
    satisfaction( S ).

accept( S, [ X | Remainder ] ) :-      % thừa nhận sau khi đọc ký tự đầu tiên
    trans( S, X, S1 ),
    accept( S1, Remainder).

accept( S, InputString ) :- % thừa nhận bởi e-chuyển tiếp
    epsilon( S, S1 ),
    accept( S1, Remainder).
```

Bây giờ, ta có thể yêu cầu ôtômat nhận biết xâu `aaab` bởi câu hỏi sau :

```
?- accept( s1, [ a, a, a, b ] ).
Yes
```

Tuy nhiên, ôtômat không thừa nhận xâu `abbb` :

```
?- accept( s1, [ a, b, b, b ] ).
ERROR: Out of local stack
```

Ta cũng thấy rằng các chương trình Prolog thường giải quyết các bài toán tổng quát hơn những gì mà NLT tạo ra chúng. Ví dụ, để yêu cầu Prolog cho biết trạng thái đầu nào thì xâu `ab` được thừa nhận :

```
?- accept( S, [ a, b ] ).
S = s1
Yes
```

Thú vị hơn nữa, ta có thể yêu cầu Prolog cho biết những xâu ba ký tự nào thì được thừa nhận bởi ôtômat :

```
?- accept( s1, [ X1, X2, X3 ] ).

X1 = a
X2 = a
X3 = b
Yes
```

Nếu ta muốn kết quả trả về là một xâu, ta chỉ cần đặt câu hỏi :

```
?- InputString = [ _ , _ , _ ], accept( s1, InputString
).
InputString = [a, a, b]
```

```
Yes
```

Đi xa hơn, tại sao ta không thể yêu cầu Prolog cho biết những trạng thái đầu nào của ôtômat cho phép nhận biết những xâu có bảy ký tự, v.v... ?

Cần phải có những thay đổi trên các quan hệ `satisfaction`, `trans` và `epsilon` nếu ta muốn ôtômat thực hiện những xử lý tổng quát hơn. Ôtômat đã cho ở Hình II.4 hông chứa các nối vòng e-chuyển tiếp. Bây giờ nếu ta thêm một chuyển tiếp :

```
epsilon( s1, s3 ).
```

thì ta đã tạo ra một nối vòng trên xâu rỗng e làm rối loạn chức năng đoán nhận của ôtômat. Lúc này với câu hỏi :

```
?- accept( s1, [ a ] ).
```

sẽ gây ra một vòng lặp quẩn vô hạn tại trạng thái s_1, trong khi ôtômat cố gắng tìm một con đường đến trạng thái thừa nhận s_3.

II.3.2. Mô phỏng ôtômat hữu hạn đơn định

Một ôđtômat hữu hạn là *đơn định* (Deterministic Finite Automaton, viết tắt DFA) nếu chuyển tiếp của ôtômat được xác định đơn định : ôtômat chỉ có thể chuyển qua một và chỉ một trạng thái tiếp theo sau khi đọc một ký tự và không có các « chuyển tiếp epsilon». Thay vì sử dụng thuật ngữ quan hệ ba ngôi, người ta thường sử dụng thuật ngữ *hàm chuyển tiếp* `delta` d`(s, a) = s'` để mô tả các hoạt động đoán nhận câu của ôtômat đơn định.

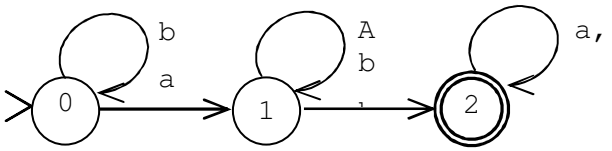
DFA được viết trong Prolog như sau :

```
parse(L) :-
    start(S),
    trans(S,L).

trans(X,[A|B]) :-
    delta(X,A,Y),           %  X ---A---> Y
    write(X),
    write('  '),
    write([A|B]),
    nl,
    trans(Y,B).

trans(X,[]) :-
    final(X),
    write(X),
    write('  '),
    write([]), nl.
```

DFA sau đây thừa nhận ngôn ngữ `(a,b)*ab(a,b)*` :

```
start(0).
final(2).
delta(0,a,1).
delta(0,b,0).
delta(1,a,1).
delta(1,b,2).
delta(2,a,2).
delta(2,b,2).
```

Sơ đồ biểu diễn ôtômat như sau :



Hình II.5. Ôtômat hữu hạn đơn định có ba trạng thái.

Sau đây là một số hoạt động đoán nhận của ôtômat :

```
?- parse([b,b,a,a,b,a,b]).
0  [b, b, a, a, b, a, b]
0  [b, a, a, b, a, b]
0  [a, a, b, a, b]
1  [a, b, a, b]
1  [b, a, b]
2  [a, b]
2  [b]
2  []
Yes
?- parse([b,b,a]).
0  [b, b, a]
0  [b, a]
0  [a]
No
```

II.4. Ví dụ : lập kế hoạch đi du lịch bằng máy bay

Trong mục này, ta sẽ xây dựng một chương trình Prolog cho phép lập kế hoạch để đi du lịch bằng máy bay. Dẫu rằng đơn giản, những ví dụ này trả lời được những câu hỏi mang tính thực tiễn sau đây :

- Những ngày nào trong tuần có chuyến bay trực tiếp từ Paris đi Ljubljana ?

- Làm cách nào để đi từ Ljubljana đến Grenoble ngày thứ Năm ?
- Tôi phải đi du lịch Milan, Ljubljana và Zurich xuất phát từ Paris ngày thứ Ba và phải quay về trong ngày thứ Sáu. Làm sao để có thể sắp xếp các chuyến đi của tôi sao cho mỗi ngày không đi máy bay quá một lần ?

Chương trình Prolog được dựa trên một cơ sở dữ liệu chứa những thông tin về các chuyến bay. Mỗi chuyến bay là một quan hệ ba ngôi cho biết lịch trình bay `timetable` như sau :

```
timetable( Place1, Place2, Fly_List).
```

Danh sách các chuyến bay có dạng như sau :

```
Departure_hour / Arrival_hour / Fly_Number / Day_List
```

Danh sách các ngày có chuyến bay hoặc là một danh sách các ngày thứ trong tuần, hoặc là một nguyên tử `all` (cho tất cả các ngày). Chẳng hạn, sau đây là một quan hệ `timetable` :

```
timetable( paris , grenoble ,
     [9:40 / 10:50 / ba4732 / all ,
      11:40 / 12:50 / ba4752 / all ,
      18:40 / 19:50 / ba4822 / [ mo , tu , we , th , fr
     ] ] ).
```

Lịch trình bay được biểu diễn bởi các cấu trúc hai thành phần là giờ và phút phân cách nhau bởi phép toán : (dấu hai chấm).

Bài toán chính đặt ra là tìm những lộ trình chính xác giữa hai thành phố (nơi đi và nơi đến) và một ngày nào đó đã cho. Để lập trình, ta sử dụng một quan hệ có bốn tham đối như sau :

```
path( Place1 , Place2 , Day, Path )
```

trong đó, là một dãy các chuyến bay thoả mãn các tiêu chuẩn sau :

(1) Nơi đi là `Place1`.

(2) Nơi đến là `Place2`.

(3) Tất cả những chuyến bay cùng ngày `Day` trong tuần.

(4) Tất cả những chuyến bay của lộ trình `Path` thuộc về quan hệ `timetable`.

(5) Có đủ thới gian để di chuyển giữa các chuyến bay.

Lộ trình được biểu diễn bởi một danh sách các đối tượng như sau :

```
Departure – Arrival : Fly_number : Departure_hour
```

Ta cũng sử dụng các vị từ bổ trợ sau đây :

(1) `fly( Place1, Place2 , Day , Fly_number , Departure_hour, Arrival_hour )`

Có nghĩa là tồn tại một chuyến bay số hiệu `Fly_number` giữa `Place1` và `Place2` trong ngày `Day`, tương ứng với ngày đi và ngày đến đã cho.

(2) `dephour( Path , Hour )`

Giờ xuất phát của lộ trình `Path` là `Hour`.

(3) `connecting( Hour1, Hour2 )`

Có ít nhất 40 phút giữa `Hour1`và `Hour2`, cho phép thực hiện việc di chuyển (nối tiếp giữa hai chuyến bay.

Vấn đề tìm một lộ trình giữa hai thành phố tương tự bài toán đoán nhận xâu của một ôtômat hữu hạn không đơn định đã xét trong mục trước. Những điểm chung là :

- Các trạng thái của ôtômat tương ứng với các thành phố.
- Một chuyển tiếp giữa hai trạng thái tương ứng với các chuyến bay giữa hai thành phố.
- Quan hệ `trans` của ôtômat tương ứng với quan hệ `timetable`.
- Để mô phỏng quá trình đoán nhận câu, ôtômat tìm được một lộ trình giữa trạng thái đầu và một trạng thái thừa nhận. Còn để mô phỏng việc lập kế hoạch đi du lịch, chương trình tìm được một lịch trình bay giữa thành phố xuất phát và thành phố đến.

Chính vì vậy, ta có thể định nghĩa một quan hệ về lộ trình `path` tương tự với quan hệ `accept`, chỉ có khác là quan hệ `path` không chứa chuyển tiếp rỗng.

Xảy ra hai trường hợp như sau :

(1) Nếu có một chuyến bay trực tiếp giữa `P1` và `P2` thì lộ trình được rút gọn thành :

```
path( P1 , P2 , Day, [ P1 – P2 : FlyNum : DepH ] ) :-
      fly( P1 , P2 , Day , FlyNum , Dep , Arr ).
```

(2) Nếu không có một chuyến bay trực tiếp giữa `P1` và `P2` thì lộ trình sẽ phải bao gồm một chuyến bay giữa `P1` và một thành phố trung gian `P3`, rồi một chuyến bay giữa `P3` và `P2`. Lúc này cần có đủ thời gian để di chuyển giữa hai chuyến bay, từ nơi đến của chuyến bay thứ nhất đến nơi xuất phát của chuyến bay thứ hai :

```
path( P1 , P2 , Day , [ P1 – P3 : FlyNum : Dep1 |
      Path ] ) :-
      path( P3 , P2 , Day , Path ),
```

```
        fly( P1 , P3 , Day , FlyNum1 , Dep1 , Arr1 ),
        dephour( Path , Dep2 ) ,
        connecting( Arr1 , Dep2 ).
```

Các quan hệ `fly`, `connecting` và `dephour` được xây dựng tương đối dễ dàng. Dưới đây là chương trình đầy đủ bao gồm cơ sở dữ liệu về lịch trình bay.

Ví dụ này đơn giản, không xảy ra trường hợp có lộ trình vô ích, nghĩa là một lộ trình không dẫn đến đâu. Ta cũng thấy rằng cơ sở dữ liệu về lịch trình bay còn nhỏ. Để có thể quản lý một cơ sở dữ liệu lớn hơn, nhất thiết phải sử dụng một chương trình lập kế hoạch thông minh hơn.

```
% Chương trình lập kế hoạch đi du lịch

:- op( 50 , xfy , : ).

fly( Place1, Place2 , Day , FlyNum , DepH , ArrH ) :-
     timetable( Place1 , Place2 , FlyList ) ,
     ismember( DepH / ArrH / FlyNum / DayList , FlyList ) ,
     flyday( Day , DayList ).

ismember( X , [ X | L ] ).

ismember( X , [ Y | L ] ) :-
     ismember( X , L ).

flyday( Day , DayList ) :-
     ismember( Day , DayList ).

flyday( Day , all ) :-
     ismember( Day , [ mo , tu , we , th , fr , sa , su ] ).

% Chuyến bay trực tiếp

path( P1 , P2 , Day, [ P1 - P2 : FlyNum : DepH ] ) :-
     fly( P1 , P2 , Day , FlyNum , DepH , _ ).

% Chuyến bay không trực tiếp

path( P1 , P2 , Day , [ P1 - P3 : FlyNum : Dep1 | Path ] ) :-
     path( P3 , P2 , Day , Path ),
     fly( P1 , P3 , Day , FlyNum1 , Dep1 , Arr1 ),
     dephour( Path , Dep2 ) ,
     connecting( Arr1 , Dep2 ).

dephour( [ P1 - P2 : FlyNum : Dep | _ ] , Dep ).

connecting( Hour1 : Mins1 , Hour2 : Mins2 ) :-
     60 *( Hour2 - Hour1 ) + Mins2 - Mins1 >= 40.

% Một cơ sở dữ liệu về lịch trình các chuyến bay

timetable( grenoble , paris ,
     [ 9 :40 / 10:50 / ba4733 / all ,
       13 :40 / 14:50 / ba4773 / all ,
       19:40 / 20:50 / ba4833 / [ mo , tu , we , th , fr , su
     ] ] ).

timetable( paris , grenoble ,
     [ 9:40 / 10:50 / ba4732 / all ,
```

```
       11:40 / 12:50 / ba4752 / all ,
       18:40 / 19:50 / ba4822 / [ mo , tu , we , th , fr ] ]
     ).
timetable( paris , ljubljana ,
     [ 13:20 / 16:20 / ju201 / [ fr ] ,
       13:20 / 16:20 / ju213 / [ su ] ] ).
timetable( paris , zurich ,
     [ 9:10 / 11:45 / ba614 / all ,
       14:45 / 17:20 / sr805 / all ] ).
timetable( paris , milan ,
     [ 8:30 / 11:20 / ba510 / all ,
       11:00 / 13:50 / az459 / all ] ).
timetable( ljubljana , zurich ,
     [ 11:30 / 12:40 / ju322 / [ tu , fr ] ] ).
timetable( ljubljana , paris ,
     [ 11:10 / 12:20 / yu200 / [ fr ] ,
       11:25 / 12:20 / yu212 / [ su ] ] ).
timetable( milan , paris ,
     [ 9:10 / 10 :00 / az458 / all ,
       12:20 / 13:10 / ba511 / all ] ).
timetable( milan , zurich ,
     [ 9:25 / 10:15 / sr621 / all ,
       12:45 / 13:35 / sr623 / all ] ).
timetable( zurich , ljubljana ,
     [ 13:30 / 14:40 / yu323 / [ tu , th ] ] ).
timetable( zurich , paris ,
     [ 9:00 / 9:40 / ba613 / [ mo , tu , we, th, fr, sa ],
       16:10 /16:55 / sr806 / [ mo , tu , we, th, fr, su ] ]
     ).
timetable( zurich , milan ,
     [ 7:55 / 8:45 / sr620 / all ] ).
```

Sau đây là một số câu hỏi trên cơ sở dữ liệu về lịch trình hàng không :

- Những ngày nào trong tuần có một chuyến bay trực tiếp giữa Paris và Ljubljana ?

```
?- fly( paris , ljubljana , Day , _ , _ , _ ).
Day = fr;
Day = su;
No
```

- Làm cách nào để có thể đi từ Ljubljana đến Grenoble ngày thứ năm ?

```
?- path( ljubljana , grenoble , th, C ).
C = [ ljubljana-paris:yu200:11:10, paris-
grenoble:ba4822:18:40 ] ;
C = [ ljubljana-paris:yu212:11:25, paris-
```

```
grenoble:ba4822:18:40 ] ;
C = [ ljubljana-zurich:ju322:11:30, zurich-
paris:sr806:16:10,
  paris-grenoble:ba4822:18:40 ]
```

- Làm cách nào để xuất phát từ Paris, có thể du lịch Milan, Ljubljana và Zurich trong ngày thứ Ba, để trở về trong ngày thứ Sáu, sao cho mỗi ngày chỉ thực hiện không quá một chuyến bay ?
  Đây là một câu hỏi tương đối lủng củng. Để trả lời, ta cần sử dụng quan hệ `permutation` đã trình bày trong chương 1, mục 3. Quan hệ này cho phép hoán vị tất cả các thành phố Milan, Ljubljana và Zurich sao cho tồn tại những chuyến bay thích hợp mỗi ngày :

```
?- permutation( [ milan , ljubljana , zurich ] , [ V1,
            V2, V3 ] ),
            fly( paris, V1, tu, FN1, Dep1, Arr1 ),
            fly( V1, V2, tu, FN2, Dep2, Arr2 ),
            fly( V2, V3, tu, FN3, Dep3, Arr3 ),
            fly( V3, paris, fr, FN4, Dep4, Arr4 ).
```

Kết quả

```
-> V1 = ljubljana
   V2 = zurich
   V3 = milan
   FN1 = ju213
   Dep1 = 13:20
   Arr1 = 16:20
   FN2 = ju322
   Dep2 = 11:30
   Arr2 = 12:40
   FN3 = sr620
   Dep3 = 7:55
   Arr3 = 8:45
   FN4 = az458
   Dep4 = 9:10
   Arr4 = 10:0 ;
```

```
-> V1 = milan
   V2 = zurich
   V3 = ljubljana
   FN1 = ba510
   Dep1 = 8:30
   Arr1 = 11:20
   FN2 = sr621
   Dep2 = 9:25
   Arr2 = 10:15
   FN3 = yu323
   Dep3 = 13:30
   Arr3 = 14:40
   FN4 = yu200
   Dep4 = 11:10
   Arr4 = 12:20
```

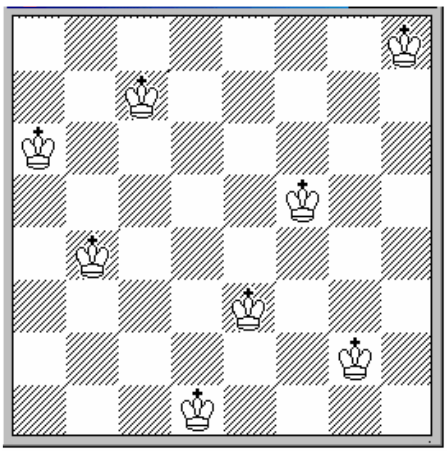
II.5. Bài toán tám quân hậu

Bài toán tám quân hậu do Call Friedrich Gauss đưa ra vào năm 1850 nhưng không có lời giải hoàn toàn theo phương pháp giải tích. Sau đó bài toán này được nhiều người giải trọn vẹn trên MTĐT, theo nhiều cách khác nhau. Bài toán phát biểu như sau :

Hãy tìm cách đặt tám quân hậu lên một bàn cờ vua (có 8 x 8 ô, lúc đầu không chứa quân nào) sao cho không có quân nào ăn được quân nào ? Một quân hậu có thể ăn được bắt cứ quân nào nằm trên cùng cột, hay cùng hàng, hay cùng đường chéo thuận, hay cùng đường chéo nghịch với nó.

Niclaus Wirth trình bày phương pháp *thử-sai* (trial-and-error) như sau :

- Đặt một quân hậu vào cột 1 (trên một hàng tuỳ ý);
- Đặt tiếp một quân hậu thứ hai sao cho 2 quân không ăn nhau;
- Tiếp tục đặt quân thứ 3, v.v...



Hình II.6. Một lời giải của bài toán tám quân hậu

Lời giải có dạng một vòng lặp theo giả ngữ Pascal như sau :

```
Xét-cột-đầu ;
repeat
   Thử_cột ;
   if An_toàn then begin
      Đặt_quân_hậu_vào ;
      Xét_cột_kế_tiếp;
   end else Quay_lại ;
until Đã_xong_với_cột_cuối or Đã_quay_lại_quá_cột_đầu ;
```

Với Prolog, chương trình sẽ có dạng một vị từ :

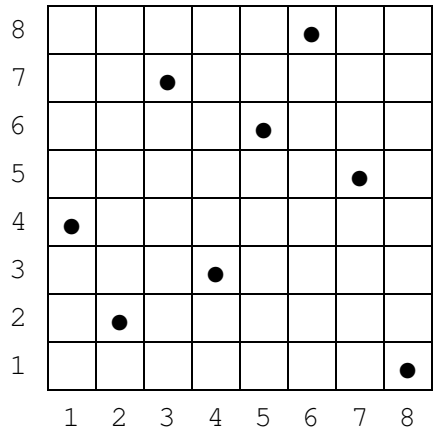
```
solution( Pos )
```

Vị từ này chỉ thoả mãn khi và chỉ khi `Pos` biểu diễn một cách bố trí tám quân hậu sao cho không có quân nào ăn được quân nào. Sau đây ta sẽ trình bày ba cách tiếp cận để lập trình Prolog dựa trên các cách biểu diễn khác nhau.

II.5.1. Sử dụng danh sách toạ độ theo hàng và cột

Ta cần tìm cách biểu diễn các vị trí trên bàn cờ. Giải pháp trực tiếp nhất là sử dụng một danh sách tám phần tử mà mỗi phần tử tương ứng với ô đặt quân hậu. Mỗi phần tử là một cặp số nguyên giữa 1 và 8 chỉ toạ độ của quân hậu :

```
X / Y
```



Hình II.7. Một lời giải của bài toán tám quân hậu, biểu diễn bởi danh sách `[ 1/4, 2/2, 3/7, 4/3, 5/6, 6/8, 7/5, 8/1 ]` .

Ở đây, phép toán / không phải là phép chia, mà chỉ là cách tổ hợp hai toạ độ của một ô bàn cờ. Hình 5.6 trên đây là một lời giải khác của bài toán tám quân hậu được biểu diễn dưới dạng một danh sách như sau :

```
[ 1/4, 2/2, 3/7, 4/3, 5/6, 6/8, 7/5, 8/1 ]
```

Từ cách biểu diễn danh sách, ta cần tìm lời giải có dạng :

```
[ X1/Y1, X2/Y2, X3/Y3, ... , X8/Y8 ]
```

Ta cần tìm các giá trị của các biến `X1`, `Y1`, `X2`, `Y2`, `X3`, `Y3`, `...` , `X8`, `Y8`. Do các quân hậu phải nằm trên các cột khác nhau để không thể ăn lẫn nhau, nên ta có ngay giá trị của các toạ độ `X`, và lời giải lúc này có dạng :

```
[ 1/Y1, 2/Y2, 3/Y3, ... , 8/Y8 ]
```

Cho đến lúc này, bài toán tám quân hậu chỉ đặt ra đối với bàn cờ 8×8. Tuy nhiên, lời giải phải dự kiến được cho trường hợp tổng quát khi lập trình. Ở đây, ta sẽ thấy rằng chính trường hợp tổng quát lại đơn giản hơn bài toán ban đầu. Bàn cờ 8×8 chỉ là một trường hợp riêng.

Để giải quyết cho trường hợp tổng quát, ta chuyển kích thước 8 quân hậu thành một số quân hậu bất kỳ nào đó (mỗi cột một quân hậu), kể cả số cột bằng không. Ta xây dựng quan hệ `solution` từ hai tình huống sau :

1. Danh sách các quân hậu là rỗng : danh sách rỗng cũng là một lời giải vì không xảy ra sự tấn công nào.

   ```
   solution( [ ] ).
   ```

2. Danh sách các quân hậu khác rỗng và có dạng như sau :

   ```
   [ X/Y | Others ]
   ```

Trong trường hợp thứ hai, quân hậu thứ nhất nằm trên ô `X/Y`, còn những quân hậu khác nằm trong danh sách `Others`. Nếu danh sách này là một lời giải, thì những điều kiện sau đây phải được thoả mãn :

1. Những quân hậu trong danh sách `Others` không thể tấn công lẫn nhau, điều này nói lên rằng `Others` cũng là một lời giải.
2. Vị trí `X` và `Y` của những quân hậu phải nằm giữa 1 và 8.
3. Một quân hậu tại vị trí `X/Y` không thể tấn công một quân hậu nào khác trong danh sách `Others`.

Đối với điều kiện thứ nhất, quan hệ `solution` phải được gọi một cách đệ quy.

Điều kiện thứ hai nói lên rằng `Y` phải thuộc về danh sách `[ 1, 2, 3, 4, 5, 6, 7, 8 ]`. Ở đây, ta không cần quan tâm đến vị trí `X`, vì nó phải tương hợp với danh sách kết quả trả về như ta đã xác định ngay từ đầu. Nghĩa là `X` phải thuộc về những giá trị đã được ấn định tương ứng.

Giả sử điều kiện thứ ba được giải quyết nhờ quan hệ `noattack`, chương trình Prolog cho quan hệ `solution` như sau :

```
solution( [ X/Y | Others ] ) :-
     solution( Others ),
     member( Y, [ 1, 2, 3, 4, 5, 6, 7, 8 ] ),
     noattack( X/Y, Others ).
```

Bây giờ ta cần tìm quan hệ `noattack`. Ta thấy :

1. Nếu danh sách `Rlist` rỗng, khi đó `noattack` là đúng, vì không có quân hậu nào tấn công quân hậu tại `X/Y` nào đó.

   ```
   noattack( _, [ ] ).
   ```

2. Nếu danh sách `Rlist` khác rỗng, khi đó có dạng `[ R1 | Rlist1 ]` và hai điều kiện sau đây phải được thoả mãn :

   (a) Quân hậu tại ô `R` không thể tấn công quân hậu tại ô `R1`, và

   (b) Quân hậu tại ô `R` không thể tấn công quân hậu nào trong `Rlist1`.

Để một quân hậu không thể tấn công quân hậu khác, thì chúng không thể nằm trên cùng hàng, cùng cột và cùng đường chéo chính hoặc phụ. Ta biết chắc chắn rằng các quân hậu đã nằm trên các cột phân biệt nhau do mô hình lời giải đã ấn định. Bây giờ ta cần chỉ ra rằng :

- Các toạ độ `Y` của các quân hậu phải phân biệt nhau, và
- Các quân hậu không thể nằm trên cùng đường chéo chính hoặc phụ. nghĩa là khoảng cách giữa các ô trên trục `X` phải khác với các ô trên trục `Y`.

```
noattack( X/Y, [ X1/Y2 | Others ] ) :-
     Y    =\= Y1,
     Y1 - Y    =\= X1 - X,
     Y1 - Y    =\= X - X1,
     noattack( X/Y, Others ).
```

Dưới đây là chương trình Prolog đầy đủ thứ nhất có chứa danh sách lời giải là quan hệ `model`. Mô hình làm cho việc tìm lời giải cho bài toán tám quân hậu trở nên đơn giản hơn.

```
% chương trình thứ nhất giải bài toán tám quân hậu
% The problem of the eight queens - Program 1
% ------------------------------------

solution( [ ] ).
solution( [ X/Y | Others ] ) :-
     solution( Others ),
     ismember( Y, [ 1, 2, 3, 4, 5, 6, 7, 8 ] ),
     noattack( X/Y, Others ).
noattack( _ , [ ] ).
noattack( X/Y, [ X1/Y1 | Others ] ) :-
     Y  =\= Y1,
     Y1 - Y =\= X1 - X,
     Y1 - Y =\= X - X1,
     noattack( X/Y, Others ).
ismember( X , [ X | L ] ).
ismember( X, [ Y | L ] ) :-
     ismember( X, L ).
model( [ 1/Y1, 2/Y2, 3/Y3, 4/Y4, 5/Y5, 6/Y6, 7/Y7, 8/Y8
     ] ).
```

```
?- model( S ), solution( S ).
  S = [1/4, 2/2, 3/7, 4/3, 5/6, 6/8, 7/5, 8/1] ;
  S = [1/5, 2/2, 3/4, 4/7, 5/3, 6/8, 7/6, 8/1] ;
  S = [1/3, 2/5, 3/2, 4/8, 5/6, 6/4, 7/7, 8/1] ;
  S = [1/3, 2/6, 3/4, 4/2, 5/8, 6/5, 7/7, 8/1] ;
  S = [1/5, 2/7, 3/1, 4/3, 5/8, 6/6, 7/4, 8/2] ;
  S = [1/4, 2/6, 3/8, 4/3, 5/1, 6/7, 7/5, 8/2]
  Yes
```

Sử dụng vị từ `not`, ta viết lại chương trình như sau :

```
solution( [ ] ).
solution( [ X/Y | Others ] ) :-
```

```
    solution( Others ),
    member( Y, [ 1, 2, 3, 4, 5, 6, 7, 8 ] ),
    not( attack( X/Y, Others )).

attack( X/Y, Others ) :-
     member( X1/Y1, Others ),
     ( Y1 = Y,
      Y1 is Y + X1 - X;
      Y1 is Y - X1 + X ).

member( A, [ A | L ] ).
member( A, [ B | L ] ) :-
     member( A, L).
```

% Mô hình lời giải

```
model( [ 1/Y1, 2/Y2, 3/Y3, 4/Y4, 5/Y5, 6/Y6, 7/Y7, 8/Y8
] ).
  ?- model( S ), solution( S ).
  S = [1/1, 2/1, 3/1, 4/1, 5/1, 6/1, 7/1, 8/1] ;
  S = [1/1, 2/8, 3/1, 4/1, 5/1, 6/1, 7/1, 8/1] ;
  S = [1/2, 2/8, 3/1, 4/1, 5/1, 6/1, 7/1, 8/1] ;
  S = [1/1, 2/1, 3/7, 4/1, 5/1, 6/1, 7/1, 8/1] ;
  S = [1/3, 2/1, 3/7, 4/1, 5/1, 6/1, 7/1, 8/1] ;
  S = [1/1, 2/2, 3/7, 4/1, 5/1, 6/1, 7/1, 8/1]
  Yes
```

II.5.2. Sử dụng danh sách toạ độ theo cột

Trong chương trình thứ nhất, ta đã đưa ra lời giải biểu diễn bàn cờ có dạng :

```
[ 1/Y1, 2/Y2, 3/Y3, ... , 8/Y8 ]
```

do mỗi cột chỉ đặt đúng một quân hậu. Thực ra, ta không mất thông tin nếu bỏ đi các toạ độ `X`. Ta có thể biểu diễn bàn cờ chỉ với các toạ độ `Y` của các quân hậu :

```
[ Y1, Y2, Y3, ... , Y8 ]
```

Để không xảy ra các quân hậu nằm trên cùng cột, cần phải bố trí mỗi quân hậu một hàng. Từ đây ta đặt ra ràng buộc cho các toạ độ `Y` : mỗi hàng 1, 2, 3, ..., 8 của bàn cờ chỉ được phép đặt duy nhất một quân hậu. Ta nhận thấy rằng mỗi lời giải là một hoán vị của danh sách các số 1 .. 8 sao cho thứ tự của mỗi con số là khác nhau :

```
[ 1, 2, 3, 5, 6, 7, 8 ]
```

Mỗi hoán vị của danh sách là một lời giải S sao cho các quân hậu ở trạng thái an toàn (không ăn được lẫn nhau). Ta có :

```
solution( S ) :-
     permutation( [ 1, 2, 3, 5, 6, 7, 8 ], S ),
     insafety( S ).
```

Trong chương 1 trước đây, ta đã xây dựng quan hệ `permutation`, bây giờ ta cần định nghĩa quan hệ `safety`. Xảy ra hai trường hợp như sau :

1. Nếu danh sách `S` rỗng, khi đó `S` cũng là lời giải, vì không có quân hậu nào tấn công quân hậu nào.

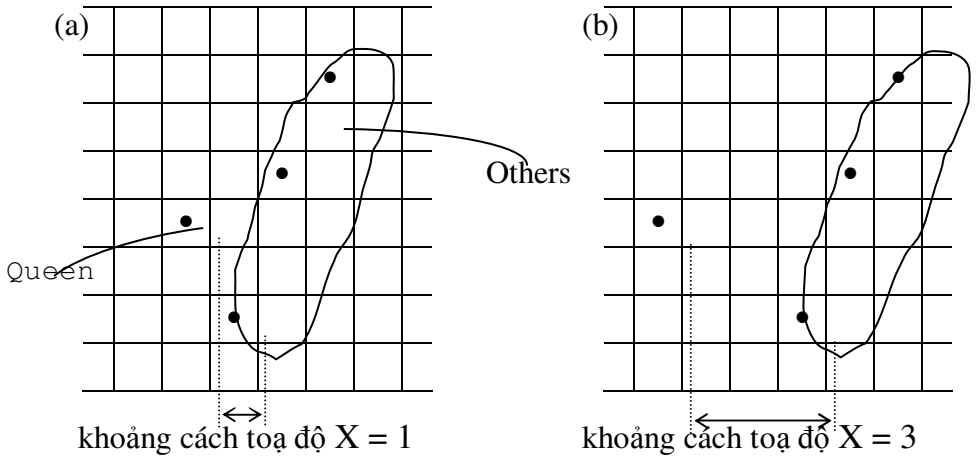
   ```
   insafety( [ ] ).
   ```

2. Nếu danh sách `S` khác rỗng, khi đó `S` có dạng `[ Queen | Others ]`. Ta thấy `S` là lời giải nếu các quân hậu trong `Others` là ở trạng thái an toàn và quân hậu `Queen` không thể tấn công quân hậu nào trong `Others`.

Từ đó ta có :

```
insafety( [ ] ).

insafety( [ Queen | Others ] ) :-
     insafety( Others ),
     noattack( Queen, Others ).
```

Trong định nghĩa `insafety`, quan hệ `noattack` tỏ ra tinh tế hơn so với cũng cùng quan hệ này trong chương trình 1 trên đây. Khó khăn nằm ở chỗ vị trí của một quân hậu chỉ được xác định bởi các toạ độ `Y`, mà vắng mặt toạ độ `X`. Để định nghĩa quan hệ `noattack`, ta tìm cách khái quát vấn đề như như minh hoạ ở hình dưới đây.



Hình II.8. Khoảng cách giữa toạ độ `X` của `Queen` và toạ độ `X` của `Others` là `1`.
(b) Khoảng cách giữa toạ độ `X` của `Queen` và toạ độ `X` của `Others` là `3`.

Ta thấy rằng sử dụng đích :

```
noattack( Queen, Others )
```

là để minh chứng rằng quân hậu `Queen` chỉ có thể tấn công các quân hậu trong danh sách `Others` khi toạ độ `X` của `Queen` cách toạ độ `X` của `Others` ít nhất là `1`.

Để thực hiện điều này, ta thêm một đối thứ ba là `XDist` (khoảng cách theo toạ độ `X` giữa `Queen` và `Others`) vào `noattack` :

```
noattack( Queen, Others, XDist )
```

Vì vậy, ta phải thay đổi lại đích `noattack` trong `insafety` như sau :

```
insafety( [ Queen | Others ] ) :-
     insafety( Others ),
     noattack( Queen, Others, XDist ).
```

Để định nghĩa `noattack`, cần phân biệt hai trường hợp của danh sách `Others` :

1. Nếu `Others` rỗng, khi đó không có quân hậu nào tấn công quân hậu nào.
   ```
   noattack( _ , [ ], _ ).
   ```
2. Nếu danh sách `Others` khác rỗng, khi đó `Queen` không thể tấn công quân hậu là phần tử đầu của danh sách `Others` (khoảng cách giữa toạ độ `X` của `Queen` và toạ độ `X` của phần tử đầu này là 1), cũng như không thể tấn công một quân hậu nào trong phần danh sách còn lại của `Others`, với một khoảng cách là `XDist + 1`.
   Từ đó ta có :
   ```
   noattack( Y, [ Y1 | YList ], XDist ) :-
        Y1 - Y =\= XDist,
        Y - Y1 =\= XDist,
        Dist1 is XDist + 1,
        noattack( Y, YList, Dist1 ).
   ```

Tất cả những lập luận và quan hệ vừa định nghĩa trên đây cho ta chương trình lời giải thứ hai cho bài toán tám quân hậu như sau :

```
% chương trình thứ hai giải bài toán tám quân hậu
% The problem of the eight queens - Program 2
% ———————————————————————————————
solution( Queens ) :-
     permutation( [ 1, 2, 3, 4, 5, 6, 7, 8 ], Queens ),
     insafety( Queens ).
permutation( [ ], [ ] ).
permutation( [ Head | Tail ], PermList ) :-
     permutation( Tail, PermTail ),
     remove( Head, PermList, PermTail ).
remove( X, [ X | L ], L ).
remove( X, [ Y | L ], [ Y | L1 ] ) :-
     remove( X, L, L1 ).
```

```
insafety( [ ] ).
insafety( [ Queen | Others ] ) :-
     insafety( Others ),
     noattack( Queen, Others, 1 ).
noattack( _ , [ ], _ ).
noattack( Y, [ Y1 | YList ], XDist ) :-
     Y1 - Y =\= XDist,
     Y - Y1 =\= XDist,
     Dist1 is XDist + 1,
     noattack( Y, YList, Dist1 ).
```

Sau khi yêu cầu, Prolog đưa ra các lời giải như sau :

```
?- solution( S ).
  S = [5, 2, 6, 1, 7, 4, 8, 3] ;
  S = [6, 3, 5, 7, 1, 4, 2, 8] ;
  S = [6, 4, 7, 1, 3, 5, 2, 8] ;
  S = [3, 6, 2, 7, 5, 1, 8, 4] ;
  S = [6, 3, 1, 7, 5, 8, 2, 4] ;
  S = [6, 2, 7, 1, 3, 5, 8, 4] ;
  S = [6, 4, 7, 1, 8, 2, 5, 3] ;
  …
  Yes
```

II.5.3. Sử dụng toạ độ theo hàng, cột và các đường chéo

Trong chương trình thứ ba, ta đưa ra lập luận như sau :

Cần phải đặt mỗi quân hậu lên một ô, nghĩa là trên một hàng, một cột, một đường chéo nghịch (từ dưới lên) và một đường chéo thuận (từ trên xuống). Để mọi quân hậu không thể ăn được lẫn nhau, chúng phải được đặt mỗi quân trên một hàng, một cột, một đường chéo nghịch và một đường chéo thuận phân biệt. Như vậy, ta dự kiến một hệ thống toạ độ biểu diễn các quân hậu như sau :

- `x` cột
- `y` hàng
- `u` đường chéo nghịch
- `v` đường chéo thuận

Các toạ độ không hoàn toàn độc lập với nhau : với `x` và `y` đã cho, ta có thể tính được `u` và `v` :

```
u = x – y
v = x + y
```

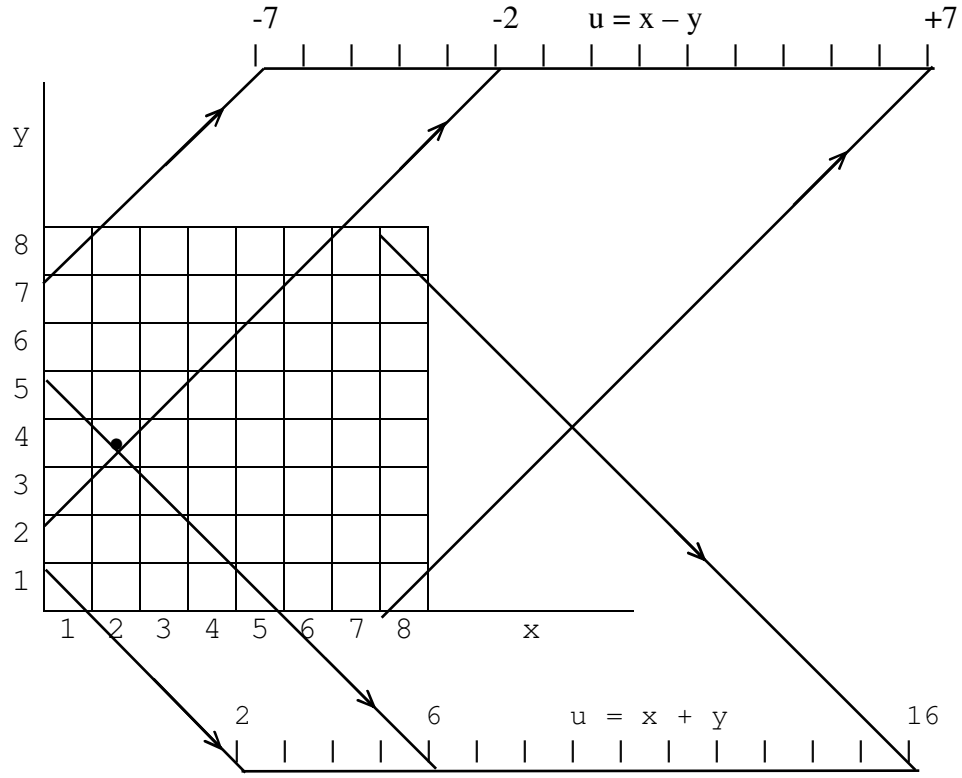
Sau đây là bốn miền giá trị tương ứng với bốn toạ độ `X`, `Y`, `U`, `V` :

```
Dx = [ 1, 2, 3, 4, 5, 6, 7, 8 ]
```

```
Dy = [ 1, 2, 3, 4, 5, 6, 7, 8 ]
Du = [ -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6,
7 ]
Dy = [ 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16 ]
```

Bài toán tám quân hậu bây giờ được phát biểu lại như sau : hãy chọn ra tám bộ bốn (X, Y, U, V), sao cho X ∈ Dx, Y ∈ Dy, U ∈ Du, V ∈ Dv và không bao giờ sử dụng hai lần cùng một phần tử trong mỗi miền giá trị Dx, Dy, Du và Dv.

Như vậy, U và V được sinh ra từ việc lựa chọn X và Y.



Hình II.9. Quan hệ giữa cột, hàng, đường chéo nghịch và đường chéo thuận
Ô có đánh dấu () trong hình có toạ độ x = 2, y = 4, u = 2 − 4 = −2, v = 2 + 4 = 6.

Lời giải đại khái có dạng như sau : cho trước bốn miền giá trị, hãy chọn một vị trí cho quân hậu đầu tiên, rồi xoá các toạ độ của chúng trong miền giá trị, sau đó sử dụng các miền giá trị mới này để đặt các quân hậu khác tiếp theo. Các vị trí trên bàn cờ cũng được biểu diễn bởi một danh sách các toạ độ trên trục Y. Chương trình Prolog giải bài toán tám quân hậu sẽ sử dụng quan hệ :

```
sol ( ListY, Dx, Dy, Du, Dv )
```

cho phép ràng buộc các toạ độ của các quân hậu (trong `ListY`), xuất phát từ nguyên lý là các quân hậu nằm trên các cột liên tiếp nhau lấy từ `Dx`. Các toạ độ `Y`, `U` và `V` được lấy từ `Dy`, `Du` và `Dv` tương ứng.

Lời giải cuối cùng của bài toán tám quân hậu là mệnh đề :

```
?- solution( S )
```

cho phép gọi `sol` với danh sách các tham đối đầy đủ. Chương trình như sau :

```
% chương trình thứ ba giải bài toán tám quân hậu
% The problem of the eight queens - Program 3
% ------------------------------------
solution( ListY) :-
     sol(    ListY ),
         [ 1, 2, 3, 4, 5, 6, 7, 8 ],
         [ 1, 2, 3, 4, 5, 6, 7, 8 ],
         [ -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5,
     6, 7 ],
         [ 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
     15, 16 ] ) .
sol( [ ], [ ],Dy, Du, Dv ).
sol( [ Y | ListY ], [ X | Dx1 ],Dy, Du, Dv ) :-
     del( Y, Dy, Dy1 ),
     U is X - Y,
     del( U, Du, Du1 ),
     V is X + Y,
     del( V, Dv, Dv1 ),
     sol ( ListY, Dx1,Dy1, Du1, Dv1 ).
del( A, [ A | List ], List ) .
del( A, [ B | List ], [ B | List1 ] ) :-
     del( A, List , List1 ).
```

Sau khi yêu cầu, Prolog đưa ra các lời giải như sau :

```
?- solution( S ).
  S = [ 1, 5, 8, 6, 3, 7, 2, 4 ] ;
  S = [ 1, 6, 8, 3, 7, 4, 2, 5 ] ;
  S = [ 1, 7, 4, 6, 8, 2, 5, 3 ] ;
  S = [ 1, 7, 5, 8, 2, 4, 6, 3 ]
  …
  Yes
```

Thủ tục `sol` vừa xây dựng trên đây có tính tổng quát vì có thể dùng để giải quyết bài toán cho `N` quân hậu bất kỳ (trên bàn cờ `N` × `N`). Sự khác nhau là ở chỗ miền giá trị `Dx`, `Dy`, … thay đổi tuỳ theo `N`.

Để tạo sinh các miền giá trị này một cách tự động, ta định nghĩa thủ tục :

```
gen( N1, N2, List ).
```

để tạo ra một danh sách các số nguyên giữa `N1` và `N2` :

```
List = [ N1, N1 + 1, N1 + 2, …, N2 – 1, N2 ]
```

Thân thủ tục như sau :

```
gen( N, N, [ N ] ).
gen( N1, N2, [ N1 | List ] ) :-
     N1 < N2,
     M is N1 + 1,
     gen( M, N2, List ).
```

Bây giờ ta thay đổi quan hệ `solution` như sau :

```
solution( N, S ) :-
     gen( 1, N, Dxy),
     Nu1 is 1 – N,
     Nu2 is N – 1,
     gen( Nu1, Nu2, Du ),
     Nv2 is N + N,
     gen( 2, Nv2, Dv ),
     sol( S, Dxy, Dxy, Du, Dv).
```

Giả sử cần giải bài toán với 12 quân hậu, ta có lời gọi như sau :

```
?- solution( 12, S ).
  S = [ 1, 3, 5, 8, 10, 12, 6, 11, 2, 7, 9, 4]
  …
  Yes
```

II.5.4. Kết luận

Ví dụ bài toán tám quân hậu trên đây minh hoạ cách giải quyết một bài toán trong Prolog theo nhiều lời giải khác nhau, mỗi lời giải sử dụng một phương pháp biểu diễn cấu trúc dữ liệu. Mỗi cách biểu diễn dữ liệu đều có những đặc trưng riêng, như tiết kiệm bộ nhớ, hay biểu diễn tường minh, hay biểu diễn phức hợp các thành phần của đối tượng cần xử lý. Cách biểu diễn dữ liệu nhằm tiết kiệm bộ nhớ có bất lợi ở chỗ là thường xuyên phải tính đi tính lại một số giá trị dữ liệu trước khi có thể sử dụng chúng.

Trong ba lời giải trên đây, lời giải thứ ba minh hoạ rõ nét hơn cả về cách xây dựng các cấu trúc dữ liệu xuất phát từ một tập hợp các phần tử đã cho có nhiều ràng buộc. Hai chương trình đầu xây dựng tất cả các hoán vị có thể rồi lần lượt kiểm tra có phải hoán vị đang xét là một lời giải không để loại bỏ những hoán vị không tốt trước khi xây dựng chúng một cách đầy đủ. Việc xác định các hoán vị

gây ra tốn thời gian do phải thực hiện nhiều lần các phép tính số học. Chương trình thứ ba tránh được điều này nhờ cách biểu diễn bàn cờ hợp lý.

II.5.5. Bộ diễn dịch Prolog

Xây dựng bộ diễn dịch Prolog bằng chính ngôn ngữ Prolog, được gọi là bộ siêu diễn dịch vani (vanilla meta-interpreter).

```
solve(true).
solve((A, B)) :-
      solve(A),
      solve(B).
solve(A) :-
      clause(A, B),
      solve(B).
```

Mệnh đề `clause(A, B)` Prolog cho phép kiểm tra nếu `A` là một `sự kiện` hay vế trái (LHS) của một luật nào đó trong cơ sở dữ liệu (chương trình Prolog), `B` là thân hay vế phải của luật đó (nếu A là một `sự kiện` thì `B = true`). `Ví dụ` :

```
?- clause(ins(X, [H|T], [ X,H|T ]), X @=< H).
X = _G420
H = _G417
T = _G418
Yes
```

Cách gọi bộ siêu diễn dịch vani :

```
?- solve(PrologGoal).
```

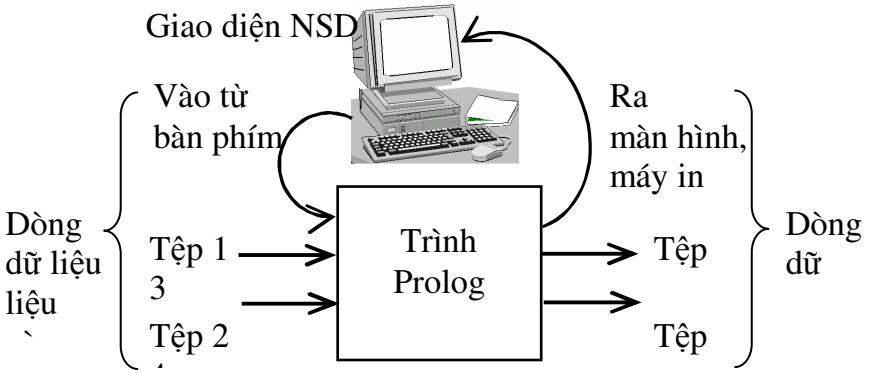
III. Quá trình vào-ra và làm việc với tệp

III.1. Khái niệm

Cho đến lúc này, ta mới làm việc với Prolog qua chế độ tương tác : NSD đặt câu hỏi là dãy các đích `dựa trên` chương trình đã biên dịch (là một CSDL chứa luật và sự kiện), Prolog trả lời cho biết các đích được thoả mãn (`Yes`) hay không thoả mãn (`No`), đồng thời tuỳ theo yêu cầu mà đưa ra kết quả dưới dạng ràng buộc giá trị cho các biến (`X = ...`). Phương pháp này đơn giản, đủ để trao đổi thông tin, tuy nhiên người ta vẫn luôn luôn tìm cách mở rộng khả năng trao đổi này. Người ta cần giải quyết những vấn đề sau :

- Vào dữ liệu cho chương trình dưới các dạng khác câu hỏi, chẳng hạn các câu trong ngôn ngữ tự nhiên (tiếng Anh, tiếng Pháp...).
- Đưa ra thông tin dưới bất kỳ dạng thức nào mong muốn.
- Làm việc được với các *tệp* (file) không chỉ thuần tuý màn hình, bàn phím.

Hầu hết các phiên bản Prolog đều có những vị từ thích hợp giải quyết được những vấn đề nêu trên. Giống như các ngôn ngữ lập trình khác, Prolog xem các thiết bị vào-ra chuẩn (bàn phím, màn hình) là các tệp đặc biệt. Quá trình vào-ra trên các thiết bị này và trên các thiết bị lưu trữ ngoài được xem là quá trình làm việc với các tệp. Hình dưới đây mô tả cách Prolog làm việc với các tệp.



Hình III.1. Liên lạc giữa một trình Prolog và nhiều tệp .

Trình Prolog có thể đọc dữ liệu vào từ nhiều tệp, được gọi là *dòng dữ liệu vào* (input streams), sau khi tính toán, có thể ghi lên nhiều tệp, được gọi là *dòng dữ liệu ra* (output streams). Dữ liệu đến từ giao diện NSD (bàn phím), rồi kết quả gửi ra màn hình, cũng được xử lý như là những dòng dữ liệu vào ra khác. Đây là những *tệp giả* (pseudo-file) được đặt tên là `user` (người sử dụng). Các tệp chứa chương trình, hay dữ liệu Prolog được NSD lựa chọn đặt tên tự do (miễn là khác `user`) trong khuôn khổ của hệ điều hành.

Khi thực hiện một trình Prolog, tại mỗi thời điểm, chỉ có hai tệp hoạt động là tệp đang được đọc, được gọi là *dòng vào hiện hành* (active input streams), và tệp đang được ghi, được gọi là *dòng ra hiện hành* (active output streams).

Lúc mới chạy chương trình, dòng vào hiện hành là bàn phím và dòng ra hiện hành là màn hình (hoặc máy in) tương ứng với chế độ vào ra chuẩn `user`.

III.2. Làm việc với các tệp

III.2.1. Đọc và ghi lên tệp

Một số vị từ xử lý đọc và ghi lên tệp của Prolog như sau :

Tên vị từ	*Ý nghĩa*
`see(File)`	Mở tệp `File` để đọc dữ liệu và xác định `File` là dòng vào hiện hành. Tệp `File` phải có từ trước, nếu không, Prolog báo lỗi tệp `File` không tồn tại.
`see(user)`	Dòng vào hiện hành là bàn phím (chế độ chuẩn).
`seeing(File)`	Hợp nhất tệp `File` với tệp vào hiện hành.

`tell(File)`	Mở tệp `File` để ghi dữ liệu lên và xác định `File` là dòng ra hiện hành. Nếu tệp `File` chưa được tạo ra trước đó, thì tệp `File` sẽ được tạo ra. Nếu tệp `File` đã tồn tại, nội dung tệp `File` sẽ bị xoá để ghi lại từ đầu.
`tell(user)`	Dòng ra hiện hành là màn hình (chế độ chuẩn).
`telling(File)`	Hợp nhất tệp `File` với tệp ra hiện hành.
`told`	Đóng tệp đang ghi lên hiện hành. Dòng vào trở lại chế độ vào chuẩn `user`.
`seen`	Đóng tệp đang đọc hiện hành. Dòng ra trở lại chế độ ra chuẩn `user`.
`read(Term)`	Đọc từ dòng vào hiện hành một giá trị để khớp với hạng `Term`. Nếu `Term` là biến thì được lấy giá trị này và vị từ thoả mãn. Nếu không thể số khớp, vị từ trả về thất bại mà không tiến hành quay lui. Mỗi hạng trong tệp phải kết thúc bởi một dấu chấm và một dấu cách (space) hoặc dấu Enter. Khi thực hiện `read` mà đang ở vị trí cuối tệp, `Term` sẽ nhận giá trị `end_of_file`.
`write(Term)`	Ghi lên tệp hiện hành giá trị của hạng `Term`. Nếu `Term` là biến thì giá trị này được đưa ra theo kiểu của Prolog. Các kiểu giá trị khác nhau đều có thể đưa ra bởi `write`.

Ví dụ III.1 :

NSD định hướng dòng vào là tệp `myexp1.pl` :

```
?- see('myexp1.pl').   % Bắt đầu đọc tệp myexp1.pl.
Yes
```

Hoặc :

```
?- see('C:/My Documents/Gt-Prolog/Example/myexp1.pl').
Yes
```

Đích `see(F)` luôn luôn được thoả mãn, trừ trường hợp xảy ra sai sót đối với các tệp dữ liệu. Chú ý tên thư mục và đường dẫn được viết theo kiểu Unix và được đặt trong cặp dấu nháy đơn. Sau khi làm việc trên tệp `myexp1.pl`, lệnh `seen` cho phép trở về chế độ chuẩn.

```
?- seen.
Yes
```

Ví dụ III.2 :

Dùng `read` để đọc dữ liệu vào bất kỳ từ bàn phím :

```
?- read(N).
|    100.
```

```
N = 100
Yes
?- read('Your name ?').
|    asimo.
No
?- read('Your name ?').
|    'Your name ?'.
Yes
?- read(asimo).
|    Your_name.
Yes
% Đọc và ghi các hạng
?- read(X).
|    father(tom, mary).
X = father(tom, mary)
Yes
T = father(tom, mary), write(T).
father(tom, mary)
T = father(tom, mary)
Yes
```

Ví dụ III.3

Đọc nội dung trong tệp `'myex1.pl'`, sau đó quay lại chế độ vào ra chuẩn.

```
?- see('myex1.pl'), read(T),see(user).
T = del(_G467, [_G467|_G468], _G468)
Yes
```

Trong dãy đích trên, đích `read(T)` đọc được sự kiện `(X, [ X | L ], L ).` là nội dung dòng đầu tiên của tệp có nghĩa, sau khi bỏ qua các dòng chú thích (nếu có).

Ta cũng có thể hướng dòng ra lên tệp bằng cách sử dụng đích :

```
?- tell('myex2.pl').
```

Dãy đích sau đây gửi thông tin là sự kiện `parent(tom, bob).` lên tệp `myex2.pl`, sau đó quay lại chế độ vào ra chuẩn :

```
tell(myex2.txt'), write('parent(tom, bob).'),
tell(user).
```

Các tệp chỉ có thể truy cập tuần tự. Prolog ghi nhớ vị trí hiện hành của dòng vào để đọc dữ liệu. Mỗi lần đọc hết một đối tượng (luật, hay sự kiện), Prolog dời đầu đọc đến vị trí đầu đối tượng tiếp theo. Khi đọc đến hết tệp, Prolog đưa ra thông báo hết tệp :

```
?- see('exp.txt'), read(T),see(user).
T = end_of_file
```

```
Yes
```

Ví dụ III.4 :

Dùng `write` để đưa dữ liệu bất kỳ ra màn hình :

```
?- write(asimo).
asimo
Yes
```

Cách ghi lên tệp cũng theo cơ chế tương tự, dữ liệu được ghi liên tiếp bắt đầu từ vị trí cuối cùng của đối tượng. Prolog không thể quay lui hay ghi đè lên phần đã ghi trước đó.

Prolog chỉ làm việc với các tệp dạng văn bản (text files), nghĩa là chỉ vào ra với các chữ cái chữ số và ký tự điều khiển ASCII.

III.2.2. Một số ví dụ đọc và ghi lên tệp

Một số vị từ đọc và ghi khác của Prolog như sau :

Tên vị từ	*Ý nghĩa*
`write(File, Term)`	Ghi lên tệp `File` giá trị hạng `Term`.
`writeq(Term)`	Ghi lên dòng ra hiện hành giá trị hạng `Term` kèm dấu nháy đơn (quotes).
`writeq(File, Term)`	Ghi lên tệp `File` giá trị hạng `Term` kèm dấu nháy đơn (quotes).
`print(Term)`	In ra dòng ra hiện hành giá trị hạng `Term`.
`print(File, Term)`	In ra tệp `File` giá trị hạng `Term`.
`read(File, Term)`	Đọc từ tệp `File` hiện hành cho `Term`.
`read_clause(Term)`	Tương tự to `read/1`. Đọc một mệnh đề từ dòng vào hiện hành.
`read_clause(File,Term)`	Đọc một mệnh đề từ tệp `File`.
`nl`	Nhảy qua dòng mới (neuwline).
`tab(N)`	In ra `N` dấu khoảng trống (space)
`tab(File, N)`	In ra `N` dấu khoảng trống trên tệp `File`

Ví dụ III.5 :

```
?- nl.  % Qua dòng mới
Yes

?- tab(5), write(*), nl.
     *
Yes
```

đưa ra màn hình 5 dấu cách rồi đến một dấu * và qua dòng.

Ví dụ III.6 :

Viết thủ tục tính luỹ thừa 3 của một số :

```
cube( N, C) :-
     C is N * N* N.
```

Giả sử ta muốn tính nhiều lần `cube`, khi đó ta phải viết nhiều lần đích :

```
?- cube( 2, X ).
X=8
Yes
?- cube( 5, Y ).
V 125
?- cube( 12, Z).
Z = 1728
Yes
```

Để chỉ cần sử dụng một đích mà có thể tính nhiều lần `cube`, ta cần sửa lại chương trình như sau :

```
cube :-
     read( X ),
     compute( X ).

compute( stop ) :- !.

compute( N) :-
     C is N *N* N,
     write( C),
     cube.
```

Nghĩa thủ tục của chương trình `cube` như sau : để tìm luỹ thừa 3, trước tiên đọc `X`, sau đó thực hiện tính toán với `X` và in ra kết quả. Nếu `X` có giá trị là `stop`, ngừng ngay, nếu không, thực hiện tính toán một cách đệ quy. Chú ý khi nhập dữ liệu cho vị từ `read`, cần kết thúc bởi một dấu chấm :

```
?- cube.
|: 3.
27
|: 10.
1000
|: 18.
5832
|: stop.
Yes
```

Ta có thể tiếp tục thay đổi chương trình. Một cách trực giác, nếu viết lại `cube` mà không sử dụng `compute` như sau là sai :

```
cube :-
     read( stop), !.
cube :-
     read( N),
```

```
        C is N *N * N,
        write( C),
        cube.
```

bởi vì, giả sử NSD gõ vào `3`, đích `read( stop)` thất bại, nhát cắt bỏ qua dữ liệu này và do vậy, `cube(3)` không được tính. Lệnh `read( N)` tiếp theo sẽ yêu cầu NSD vào tiếp dữ liệu cho `N`. Nếu `N` là số, việc tính toán thành công, ngược lại, nếu `N` là `stop`, Prolog sẽ thực hiện tính toán trên các dữ liệu phi số `stop` :

```
?- cube1.
|: 3.              % Prolog bỏ qua, không tính
|: 9.
729               % Prolog tính ra kết quả cho N = 9
|: 4.              % Prolog bỏ qua, không tính
|: stop.           % Prolog báo lỗi
ERROR: Arithmetic: `stop/0' is not a function
^  Exception: (9) _L143 is stop*stop*stop ? creep
```

Thông thường các chương trình khi thực hiện cần sự tương tác giữa NSD và hệ thống. NSD cần được biết kiểu và giá trị dữ liệu chương trình yêu cầu nhập vào. Muốn vậy, chương trình cần đưa ra dòng yêu cầu hay lời nhắc (prompt). Hàm `cube` được viết lại như sau :

```
 cube :-
      write('Please enter a number: '),
      read( X ),
      compute( X ).

compute( stop ) :- !.

compute( N) :-
      C is N *N* N,
      write('The cube of '), write(N),
      write(' is '), write( C), nl,
      cube.

cube.
Please enter a number: 3.
The cube of 3 is 27
Please enter a number: stop.
Yes
```

Ví dụ III.7

Ta xây dựng thủ tục `displaylist` sau đây để in ra các phần tử của danh sách :

```
displaylist( [ ]).

displaylist( [X | L ] ) :-
      write( X ), nl,
      displaylist( L).
```

```
?- displaylist( [[a, b, c], [d, e, f], [g, h, i]]).
[a, b, c]
[d, e, f]
[g, h, i]
Yes
```

Ta thấy trong trường hợp các phần tử của một danh sách lại là những danh sách như trên thì tốt hơn cả là in chúng ra trên cùng hàng :

```
displaylist( [ ]).
displaylist( [X | L ] ) :-
     write( X ), tab( 1),
     displaylist( L), nl.
displaylist( [[a, b, c], [d, e, f], [g, h, i]]).
[a, b, c] [d, e, f] [g, h, i]
Yes
```

Thủ tục dưới đây in ra các phần tử kiểu danh sách phẳng trên cùng hàng :

```
displaylist2( [ ] ).
displaylist2( [ L | L1 ] ) :-
     inline( u),
     displaylist2( L1 ), nl.
inline( [ ] ).
inline( [ X I L ] ) :-
     write( X ), tab( 1),
     inline( L).
?- displaylist2( [[a, b, c], [d, e, f], [g, h, i]]).
a b c d e f g h i
Yes
```

Ví dụ dưới đây in ra danh sách các số nguyên dưới dạng một đồ thị gồm các dòng kẻ là các dấu sao (hoa thị) * :

```
barres( [ N | L]) :-
     asterisk(N), nl,
     barres(L).
asterisk( N) :-
     N > 0,
     write( *),
     N1 is N - 1,
     asterisk( N1).
asterisk( N) :-
     N =< 0.
?- barres([3, 4, 6, 5, 9]).
***
****
******
```

```
*****
*********
No
```

Ví dụ III.8 :

Đọc nội dung một tệp vào danh sách các số nguyên :

```
readmyfile( File, List) :-
      see( File),
      readlist( List),
      seen,
      !.
readlist( [X | L ]) :-
      get0(X),
      X =\= -1,
      !,
      read_list( L).
readlist( [ ] ).
```

III.2.3. Nạp chương trình Prolog vào bộ nhớ

Các chương trình Prolog thường được lưu cất trong các tệp có tên hậu tố (hay phần mở rộng của tên) là « **.pl** » . Để nạp chương trình (load) vào bộ nhớ và biên dịch (compile, Prolog sử dụng vị từ :

```
?- consult(file_name).
```

trong đó, `file_name` là một nguyên tử.

Ví dụ III.9 :

Đích sau đây nạp và biên dịch chương trình nằm trong tệp `myexp.pl` :

```
?- consult('myexp.pl').

Yes
```

Prolog cho phép viết gọn trong một danh sách như sau :

```
?- ['myexp.pl' ].
```

Để nạp và biên dịch đồng thời nhiều tệp chương trình khác nhau, có thể liệt kê trong một danh sách như sau :

```
?- ['file1.pl', 'file2.pl'].
```

Sau khi các chương trình đã được nạp vào bộ nhớ, NSD bắt đầu thực hiện chương trình. NSD có thể xem nội dung toàn bộ chương trình nhờ vị từ :

```
?- listing.
```

hoặc xem một mệnh đề nào đó :

```
?- listing(displaylist).
displaylist( [ ]).
```

```
displaylist( [X | L ] ) :-
        write( X ),
        tab( 1),
        displaylist( L), nl.
Yes
```

III.3. Ứng dụng chế độ làm việc với các tệp

III.3.1. Định dạng các hạng

Giả sử một bản ghi cơ sở dữ liệu, là một sự kiện có dạng cấu trúc hàm tử của Prolog, có nội dung như sau :

```
family( individual(tom, smith, date(7, may, 1960),
     work(microsoft, 30000)),
     individual( ann, smith, date(9, avril, 1962),
     inactive),
     [individual( roza, smith, date(16, june, 1991),
     inactive),
      individual( eric, smith, date(23, march, 1993),
     inactive) ] ).
```

Ta cần in ra nội dung bản ghi sử dụng vị từ `write(F)` theo quy cách như sau :

```
parents
 tom smith, birth day may 7,1960, work microsoft, salary
 30000
 ann smith, birth day avril 9, 1962, out of work

children
 roza smith, birth day june 16, 1991, out of work
 eric smith, birth day march 23, 1993, out of work
```

Ta xây dựng thủ tục `writefamily( F)` như sau :

```
writefamily(family(Husband, Wife, Children)) :-
     nl, write(parents),nl, nl,
     writeindividual(Husband) ,nl,
     writeindividual(Wife), nl, nl,
     write(children), nl, nl,
     writeindividual(Children).

writeindividual(individual(Firstname, Name, date(D, M,
     Y), Work)) :-
     tab(4), write(Firstname),
     tab(1), write(Name),
     write(', birth day '), write(M), tab(1),
     write(D), tab(1), write(', '), write(Y), write(',
```

```
        '),
        writework(Work).
writeindividual([ ]).
writeindividual([ P | L] ):-
        writeindividual( P), nl,
        writeindividual( L).
writework(inactive):-
        write('out of work').
writework(work(Soc, Sal)):-
        write(' work '), write(Soc),
        write(', salaire '), write(Sal).
```

Thực hiện đích`X = ..., writefamily(X)`, ta nhận được kết quả như sau

```
?- X = family(individual( tom, smith, date(7, may,
1960), work(microsoft, 30000) ),individual( ann, smith,
date(9, avril, 1962), inactive),[individual( roza,
smith, date(16, june, 1991), inactive),individual( eric,
smith, date(23, march, 1993), inactive) ] ),
writefamily(X).
parents
 tom smith, birth day may 7 , 1960,  work microsoft, salaire
 30000
 ann smith, birth day avril 9 , 1962, out of work
children
 roza smith, birth day june 16 , 1991, out of work
 eric smith, birth day march 23 , 1993, out of work
X = family(individual(tom, smith, date(7, may, 1960),
work(microsoft, 30000)), individual(ann, smith, date(9,
avril, 1962), inactive), [individual(roza, smith,
date(16, june, 1991), inactive), individual(eric, smith,
date(23, march, 1993), inactive)])
Yes
```

III.3.2. Sử dụng tệp xử lý các hạng

Để đọc dữ liệu trên tệp, người ta sử dụng dãy đích sau :

```
..., see( F), fileprocess, see( user), ...
```

Thủ tục `fileprocess` đọc và xử lý lần lượt từng hạng của `F` cho đến khi đọc hết tệp. Mô hình thủ tục như sau :

```
filetreat :-
        read( Term),
treat( Term).
treat( end_of_file) :- !.   % Kết thúc tệp
        treat( Term) :-
```

```
        treatment( Term), % Xử lý hạng hiện hành
        filetreat.        % Xử lý phần còn lại của tệp
```

Trong thủ tục trên, `treatment( Terme)` thể hiện mọi thao tác có thể tác động lên hạng. Chẳng hạn thủ tục dưới đây liệt kê từng hạng của tệp kể từ dòng thứ `N` trở đi cho đến hết tệp, kèm theo thứ tự có mặt của hạng đó trong tệp :

```
viewfile( N) :-
     read( Term),
     viewterm( Term, N).
viewterm( end_of_file, _ ) :- !.
viewterm( Term, N) :-
     write( N), tab( 2),
     write( Term), nl,
     N1 is N + 1,
     viewfile( N1).
?- see('exp.txt'), viewfile(1), see( user), seen.
1  parent(pam, bob)
2  parent(tom, bob)
3  parent(tom, liz)
4  parent(bob, ann)
5  parent(bob, pat)
…
Yes
```

Sau đây là một mô hình khác để xử lý tệp. Giả sử `file1` là tệp dữ liệu nguồn chứa các hạng có dạng :

```
object( NoObject, Description, Price, FurnisherName).
```

Mỗi hạng mô tả một phần tử của danh sách các đối tượng. Giả sử rằng tệp cần xây dựng `file2` chứa các đối tượng do cùng một nhà cung cấp cấp hàng. Trong tệp này, tên nhà cung cấp được viết một lần ở đầu tệp, mà không xuất hiện trong các đối tượng, có dạng `object( No, Desc, Price)`. Thủ tục tạo tệp như sau :

```
createfile(Furnisher) :-
    write(Furnisher), write( '.'), nl,
    creatremaining(Furnisher).
creatremaining( Fournisseur) :-
    read( Objet),
    treat( Objet, Furnisher).
treat( end_of_file) :- !.
treat(object( No, Desc, Price, Furn), Furn) :-
    write( object( No, Desc, Price) ),
    write( '.'), nl,
    creatremaining(Furn).
```

```
treat( _ , Furnisher) :-
    creatremaining(Furnisher).
```

Giả sử `file1` là tệp

```
see(' file1.txt'),tell(' file2.txt'), createfile(suzuki),
seen, see(user), told, tell(user).
```

Ví dụ III.10 :

Sao chép nội dung một tệp lên một tệp khác :

```
copie :-
    repeat,
    read(X),
    mywrite(X),
    X == end_of_file, !.
mywrite( end_of_file).
mywrite( X) :-
    write( X), write( '.'), nl.
```

Đích sau cho phép côpy từ tệp nguồn `f1.txt` vào tệp đích `f2.txt` :

```
?- tell('f2.txt'), see('f1.txt'), copie, seen, told.
Yes
```

Trong thủ tục `copie` có sử dụng vị từ `repeat`. Vị từ `repeat` luôn luôn thành công, tạo ra một vòng lặp vô hạn. Vị từ `repeat` được định nghĩa như sau :

```
repeat.
repeat :- repeat.
```

III.3.3. Thao tác trên các ký tự

Một số vị từ xử lý ký tự của Prolog như sau :

Tên vị từ	*Ý nghĩa*
`put(Char)`	Đưa `Char` ra dòng ra hiện hành, `Char` hoặc là một giá trị nguyên trong khoảng 0..255, hoặc một ký tự
`put(File, Char)`	Đưa `Char` ra tệp `File`
`get_char(Char)`	Đọc từ tệp `File` và hợp nhất `Char` với ký tự tiếp theo.
`get_char(File, Char)`	Hợp nhất `Char` với ký tự tiếp theo trong tệp `File`.
`get0(Char)`	Đọc ký tự tiếp theo
`get0(File, Char)`	Đọc ký tự tiếp theo trong tệp `File`.
`get(-Char)`	Đọc ký tự khác khoảng trống từ dòng vào và hợp nhất với `Char`.
`get(File, Char)`	Đọc ký tự khác khoảng trống tiếp theo trong tệp `File`.
`skip(Char)`	Đọc vào và bỏ qua các ký tự đọc được cho đến khi gặp đúng ký tự khớp được với `Char`.

`skip(File, Char)`	Đọc vào từ tệp `File` và bỏ qua các ký tự đọc được cho đến khi gặp đúng ký tự khớp được với `Char`.

Ví dụ III.11 :

```
% Đưa ra liên tiếp các ký tự A, B và C có mã ASCII lần lượt là 65, 66, 67
?- put( 65), put( 66), put( 67).
ABC
yes
% Đọc và ghi các ký tự
?- get0(X).
|: a          % Gõ vào một ký tự rồi Enter (↵), không gõ dấu chấm
X = 97
Yes.
?- get0(X).
^D
X = -1.
Yes.
```

Ví dụ III.12 :

Sau đây ta xây dựng thủ tục `del_space` đọc vào một câu gồm nhiều từ cách nhau bởi các khoảng trống và trả về đúng câu đó sau khi đã loại bỏ các khoảng trống thừa, chỉ giữ lại một khoảng trống giữa các từ mà thôi.

Thủ tục hoạt động tương tự các thủ tục xử lý tệp, bằng cách đọc lần lượt từng ký tự rồi đưa ra màn hình. Thủ tục sử dụng kỹ thuật nhát cắt để xử lý tình huống ký tự đọc vào hoặc là một khoảng trống, hoặc là một chữ cái, hoặc là một dấu chấm kết thúc. Sau đây là thủ tục `del_space` :

```
del_space :-
     get0( C),
     put( C),
     follow( C).
follow( 46) :- !.      % 46 là mã ASCII của dấu chấm
follow( 32) :- !,      % 32 là mã ASCII của dấu khoảng
     trống
     get( C),     % Bỏ qua các dấu khoảng trống tiếp theo
     put( C),
     follow( C).
follow( Letter) :-
     del_space.
```

Chạy thử như sau :

```
?- del_space.
|: The   robot          try   to      cast   the   balls
to   the   basket.
```

```
The robot try to cast the balls to the basket.
Yes
```

III.3.4. Thao tác trên các nguyên tử

Prolog có vị từ `name/2` cho phép đặt tương ứng các nguyên tử với các mã ASCII :

```
name( A, L)
```

Vị từ thoả mãn khi `L` là danh sách các của các ký tự của `A`. Ví dụ :

```
?- name(mic29, [109, 105, 99, 50, 57 ]).
Yes
?- name( aikieutuido, L).
L = [ 97, 105, 107, 105, 101, 117, 116, 117, 105 |... ]
Yes
?- name(X, [ 97, 105, 107, 105, 101, 117, 116, 117, 105,
100, 111 ]).
X = aikieutuido
Yes
```

Hai chức năng chính của vị từ `name` như sau :

1. Chuyển một nguyên tử thành một danh sách các ký tự (mã ASCII).
2. Tạo một nguyên tử từ một danh sách các ký tự.

Ví dụ III.13 :

Xây dựng thủ tục quản lý các cuộc gọi dịch vụ xe taxi chở hành khách nhờ các nguyên tử sau :

Tên các cuộc gọi `call1`, `call2`, ...
Tên các lái xe `chauffeur1`, `chauffeur2`, ...
Tên các xe taxi `taxi1`, `taxi2`, ...

Vị từ `taxi( X )` kiểm tra một nguyên tử có biểu diễn đúng một taxi theo cách biểu diễn như trên không :

```
taxi( T ) :-
     name( T, Tlist),
     name( taxi, L),
     append( L, _ , Tlist).
```

Một cách tương tự, ta có thể xây dựng các vị từ `chauffer` và `taxi`.

Ví dụ III.14 :

Sau đây ta xây dựng thủ tục cho phép tạo ra một nguyên tử bằng cách tổ hợp các ký tự. Thủ tục `readsentence( Wordlist)` sẽ đọc một câu thuộc ngôn ngữ tự nhiên rồi gán cho `Wordlist` danh sách các giá trị mã biểu diễn trong của

các ký tự trong câu. Tiếp theo, mỗi câu được xem là một danh sách các từ, mỗi từ được chuyển thành một nguyên tử.

```
readsentence( WordList) :-
     get0( Char),
     readchain( Char, WordList).
readchain( 46,[ ] ) :- !.   % dấu chấm kết thúc câu
readchain( 32, WordList) :-
     readsentence(WordList).     % Bỏ qua các dấu khoảng trống
readchain( L, [ W | WordList ] ) :-
     readletter( L, Letters, Nextchar ),   % Đọc các ký tự của
     từ tiếp theo
     name( W, Letters),
     readchain( Nextchar, WordList).
readletter( 46, [ ], 46) :- !.   % kết thúc từ là một dấu chấm
readletter( 32, [ ], 32) :- !.   % kết thúc từ là một dấu khoảng
     trống
readletter( C, [ C | Letters] , Nextchar) :-
     get0( Char),
     readletter( Char, Letters, Nextchar).
```

Chạy chương trình, ta có các kết quả như sau :

```
?- readsentence( WordList).
|: The robot ASIMO try to cast the balls to the basket.
WordList = ['The', robot, 'ASIMO', try, to, cast, the,
balls, to|...]
Yes
?- readsentence( WordList).
|: " Ai đi trăm suối ngàn rừng "      % dấu Enter ↵ sau dấu nháy kép
|: .                                  % dấu chấm kết thúc câu
WordList = [ '" Ai', đi, trăm, suối, ngàn, 'rừng "\n' ]
Yes
```

Trong thủ tục, ta đã giả thiết rằng kết thúc câu vào là một dấu chấm và nếu có dấu chấm câu trong câu, thì tuỳ theo cách xuất hiện mà nó được xem như là một từ hoặc dính vào với từ.

Thủ tục đọc ký tự đầu tiên là `Char`, rồi chuyển cho thủ tục `readchain`.

Thủ tục `readchain` xử lý 3 trường hợp như sau :

(1) Nếu `Char` là một dấu chấm, thì quá trình đọc câu vào kết thúc.

(2) Nếu `Char` là một khoảng trống, áp dụng thủ tục `readsentence` cho phần còn lại của câu.

(3) Nếu `Char` là một ký tự : trước tiên đọc từ `W` được bắt đầu bởi ký tự `Char`, sau đó sử dụng `readsentence` để đọc phần còn lại của câu và tạo ra danh sách `WordList`. Kết quả được tích luỹ trong `[ W | WordList ]`.

Thủ tục `readletter( L, Letters, Nextchar )` đọc các ký tự của một từ, trong đó :

(1) `L` là chữ cái hiện hành (đã được đọc) của từ đang đọc.

(2) `Letters` là danh sách các chữ cái, bắt đầu bởi `L` cho đến hết từ.

(3) `Nextchar` là ký tự theo sau từ đang đọc, có thể không phải là một chữ cái.

Nhờ cách biểu diễn các từ của câu trong một danh sách, người ta có thể sử dụng Prolog để xử lý ngôn ngữ tự nhiên, như tìm hiểu nghĩa của câu theo một quy ước nào đó, v.v.. thuộc lĩnh vực trí tuệ nhân tạo.

III.3.5. Một số vị từ xử lý cơ sở dữ liệu

Sau đây là một số vị từ chuẩn cho phép xử lý trên các luật và sự kiện của một cơ sở dữ liệu Prolog.

`assert(P)`

Thêm `P` vào cơ sở dữ liệu. Ví dụ cho cơ sở dữ liệu lúc ban đầu :

```
personal(tom).
personal(ann).
```

Sau khi thực hiện đích :

```
?- assert(personal(bob)).
```

cơ sở dữ liệu lúc này trở thành :

```
personal(tom).
personal(ann).
personal(bob).
```

Do NSD không biết `assert` đã thêm `P` vào đầu hay cuối của cơ sở dữ liệu, Prolog cho phép sử dụng hai dạng khác là :

`asserta(P)` Thêm `P` vào đầu cơ sở dữ liệu.

`assertz(P)` Thêm `P` vào cuối cơ sở dữ liệu.

Sử dụng vị từ :

```
assert((P :- B, C, D)).
```

có thể làm thay đổi nội dung một mệnh đề trong chương trình. Tuy nhiên, người ta khuyên không nên sử dụng lệnh này.

`retract(P)`

Loại bỏ `P` khỏi cơ sở dữ liệu. Ví dụ cho cơ sở dữ liệu lúc ban đầu :

```
personal(tom).
personal(ann).
personal(bob).
```

Sau khi thực hiện đích :

```
?- retract(personal(ann)).
```

cơ sở dữ liệu lúc này chỉ còn :

```
personal(tom).
personal(bob).
```

Có thể sử dụng biến trong `retract` như sau :

```
?- retract(personal(X)).
X = tom ;
X = bob ;
No
```

Lúc này cơ sở dữ liệu đã rỗng.

`abolish(Term, Arity)`

Loại bỏ tất cả các hạng `Term` có cấp `Arity` khỏi cơ sở dữ liệu. Ví dụ :

```
?- abolish(personal, 2).
```

Loại bỏ tất cả các hạng `Term` có cấp `Arity=2`.

Ví dụ III.15

Xây dựng bộ siêu diễn dịch Prolog trong Prolog, việc xoá một đích được viết lại như sau :

```
prove(Goal) :- call(Goal).
```

hoặc :

```
prove(Goal) :- Goal.
```

hoặc viết các mệnh đề :

```
prove(true).
prove((Goal1, Goal2)) :-
      prove(Goal1),
      prove(Goal2).
prove(Goal) :-
      clause(Goal, Body),
      prove(Body).
```

Tóm tắt chương 5 :

Kỹ thuật nhát cắt và phủ định

- Nhát cắt ngăn cản sự quay lui, không những làm tặng hiệu quả chạy chương trình mà còn làm tối ưu tính biểu hiện của ngôn ngữ.
- Để tặng hiệu quả chạy chương trình, người lập trình sử dụng nhát cắt để chỉ ra cho Prolog biết những con đường dẫn đến thất bại.
- Nhát cắt cho phép tạo ra các kết luận loại trừ nhau dạng :

  If Condition Thì Conclusion_1 nếu Conclusion_2
- Nhát cắt cho phép định nghĩa phép phủ định : `not Goal` thoả mãn nếu `Goal` thất bại.
- Prolog có hai đích đặc biệt : `true` luôn luôn đúng và `fail` luôn luôn sai.
- Cần thận trọng khi sử dụng kỹ thuật nhát cắt, nhát cắt có thể làm sai lệch sự tương ứng giữa nghĩa khai báo và nghĩa thủ tục của một chương trình.
- Phép phủ định `not` trong Prolog không hoàn toàn mang ý nghĩa lôgich, cần chú ý khi sử dụng `not`.

Sử dụng các cấu trúc

Các ví dụ đã trình bày trong chương này minh hoạ những đặc trưng rất tiêu biểu của kỹ thuật lập trình Prolog :

- Trong Prolog, tập hợp các sự kiện đủ để biểu diễn một cơ sở dữ liệu.
- Kỹ thuật đặt câu hỏi và so khớp của Prolog là những phương tiện mềm dẻo cho phép truy cập tù cơ sở dữ liệu những thông tin có cấu trúc.
- Cần sử dụng phương pháp trừu tượng hoá dữ liệu như là một kỹ thuật lập trình cho phép sử dụng các cấu trúc dữ liệu phức tạp một cách đơn giản, làm chương trình trở nên dễ hiểu. Trong Prolog, phương pháp trừu tượng hoá dữ liệu rất dễ triển khai.
- Những cấu trúc toán học trừu tượng như ôtômat cũng rất dễ cài đặt trong Prolog.
- Người ta có thể tiếp cận đến nhiều lời giải khác nhau cho một bài toán nhờ sử dụng nhiều cách biểu diễn dữ liệu khác nhau, như trường hợp bài toán tám quân hậu. Cách biểu diễn dữ liệu sử dụng nhiều thông tin tiết kiệm được tính toán, mặc dù làm cho chương trình trở nên rườm rà, khó cô đọng.
- Kỹ thuật tổng quát hoá một bài toán, tuy trừu tượng, nhưng lại làm tăng khả năng hướng đến lời giải, làm đơn giản hoá phát biểu bài toán.

Làm việc với tệp

Cùng với chế độ tương tác câu hỏi-trả lời, quá trình vào ra và chế độ làm việc với tệp đã làm phong phú môi trường làm việc của Prolog.

- Các tệp trong Prolog đều hoạt động theo kiểu tuần tự. Prolog phân biệt dòng vào hiện hành và dòng ra hiện hành.
- Thiết bị cuối (terminal) của NSD gồm màn hình và bàn phím được xem như một tệp giả có tên là `user`.
- Prolog có nhiều vị từ có sẵn để xử lý các dòng vào-ra.
- Khi làm việc với tệp, chế độ đọc ghi là xử lý từng ký tự hoặc từng hạng.

Bài tập chương 5

1. Cho chương trình :

```
p( 1 ).
p( 2 ) :- !.
p( 3 ).
```

Cho biết các câu trả lời của Prolog tù các câu hỏi sau :

(a) `?- p( X ).`

(b) `?- p( X ), p( Y ).`

(c) `?- p( X ), !, p( Y ).`

2. Quan hệ sau đây cho biết một số có thể là dương, bằng không, hoặc âm :

```
sign( Number, positive) :-
     Number > 0.

sign( 0, null).

sign( Number, negative) :-
     Number < 0.
```

Hãy sử dụng kỹ thuật nhát cắt để viết lại chương trình trên hiệu quả hơn.

3. Thủ tục `separate(Number, Positive, Negative)` xếp các phần tử trong danh sách `Number` lần lượt thành hai danh sách, danh sách `Positive` chỉ chứa các số dương, hoặc bằng không, danh sách `Negative` chỉ chứa các số âm. `Ví dụ` :

```
separate( [ 3, -1, 0, 5, -2 ], [ 3, 0, 5 ], [ -1, -2 ] )
```

Hãy định nghĩa thủ tục trên theo hai cách, một cách không sử dụng kỹ thuật nhát cắt, một cách có sử dụng kỹ thuật nhát cắt.

4. Cho hai danh sách, `Accept` và `Reject`, hãy viết danh sách các đích sử dụng kỹ thuật quay lui và các quan hệ `member` và `not` để tìm các phần tử có mặt trong `Accept` nhưng không có mặt trong `Reject`.
5. Định nghĩa thủ tục `difference( Set1, Set2, SetDiff)` tìm hiệu hai tập hợp `Set1` và `Set2` với quy ước các tập hợp được biểu diễn bởi các danh sách.. Chẳng hạn :

   ```
   difference( [ a, b, c, d ], [ b, d, e, f ], [ a, c ] )
   ```
6. Hãy định nghĩa vị từ `unifiable( List1, Term, List2)` để kiểm tra so khớp, trong đó `List2` là danh sách tất cả các phần tử của `List1` có thể so khớp với `Term` nhưng không thực hiện phép thế trên các biến đã được so khớp. `Ví dụ` :

   ```
   ?- unifiable( [ X, bibo, t( Y ) ], t(a), List ).
   List = [X, t( Y )]
   ```

   Chú ý rằng `X` và `Y` vẫn là các biến tự do không thực hiện phép thế `t(a)` cho `X`, hay phép thế `a` cho `Y`. Muốn vậy, thực hiện hướng dẫn sau :
   Sử dụng phép phủ định `not( Term1 = Term2)`. Nếu quan hệ `Term1 = Term2` được thoả mãn, khi đó, `not( Term1 = Term2)` sẽ thất bại, và phép thế biến không xảy ra.
7. Bài toán mã đi tuần. Giả sử các ô của bàn cờ vua 8×8 được biểu diễn bởi các cặp toạ độ có dạng `X/Y`, với `X` và `Y` nằm trong khoảng 1 và 8.
   (a) Định nghĩa quan hệ `jump( case1, case2 )`, bằng cách sử dụng luật đi của quân mã, và giả sử rằng `case1` luôn luôn bị ràng buộc. Ví dụ :

   ```
   ?- jump( 1/1, C ).
   C = 3/2;
   C = 2/3;
   No
   ```

   (b) Định nghĩa quan hệ `mvt_ knight( path )`, với `path` là một danh sách gồm các ô biểu diễn lộ trình các bước nhảy hợp lý của quân mã trên bàn cờ rỗng.

   (c) Sử dụng quan hệ `mvt_ knight`, viết một câu hỏi để tìm tất cả các lộ trình bốn bước nhảy hợp lý của quân mã, xuất phát từ ô có toạ độ `2/1`, để đến biên bên phải của bàn cờ (`Y = 8`) và để đến ô `5/4` sau hai bước nhảy.
8. Cho `f` một tệp chứa các hạng, hãy định nghĩa thủ tục `findterm(Term)` để đưa ra màn hình hạng đầu tiên của `f` khớp được với `Term` ?
9. Cho `f` một tệp chứa các hạng, hãy định nghĩa thủ tục `findallterm(Term)` để đưa ra màn hình tất cả các hạng của `f` khớp được với `Term` ? Kiểm tra tính chất biến `Term` không thể được gán giá trị khi thực hiện tìm kiếm.

10. Hãy mở rộng thủ tục `del_space` đã được trình bày trong phần lý thuyết để có thể xử lý loại bỏ các dấu cách thừa nằm trước dấu phẩy (comma) và chỉ giữ lại một dấu cách nằm ngay sau dấu phẩy.
11. Tương tự bài 3 cho các dấu chấm câu khác như dấu chấm (period), dấu chấm phẩy (semicolon), dấu chấm hỏi (question mark), v.v...
12. Định nghĩa quan hệ `firstchar( Atom, Char)` cho phép kiểm tra `Char` có phải là ký tự đầu tiên của `Atom` không (`Atom` bắt đầu bởi `Char`) ?
13. Định nghĩa thủ tục cho phép đổi một danh từ tiếng Anh từ số ít (singular) sang số nhiều (plural) được thực hiện như sau :

```
?- plural ( table, X ).
X = tables
Yes
```

14. Áp dụng thủ tục `readsentence` đã được trình bày trong phần lý thuyết để xây dựng thủ tục :

```
?- find( Keyword, Sentence ).
```

cho phép tìm trong tệp đang đọc một câu có chứa từ khoá `Keyword`. Câu `Sentence` phải ở dạng mới được đọc vào chưa xử lý, nghĩa là được biểu diễn bởi một chuỗi ký tự, hoặc bởi một nguyên tử.

THÔNG TIN VỀ TÁC GIẢ
GIÁO TRÌNH "LẬP TRÌNH LÔGIC"

1 Thông tin về tác giả :

+ Họ và tên : **Phan Huy Khánh**
+ Quê quán : Nghệ An
+ Cơ quan công tác :
  Khoa Công nghệ Thông tin,
  Trường Đại học Bách khoa, Đại học Đà Nẵng
+ Email:
  khanhph@vnn.vn, phk@ud.edu.vn

2 Phạm vi và đối tượng sử dụng :

+ Giáo trình dùng tham khảo cho các ngành Công nghệ Thông tin
+ Có thể dùng ở các trường có đào tạo các chuyên ngành Công nghệ Thông tin
+ Từ khóa :
  Sự kiện, luật, luật đệ quy, mệnh đề, hạng, vị từ, đích, đệ quy, danh sách, hợp nhất, nhát cắt.
+ Yêu cầu kiến thức trước khi học môn này :
  Tin học đại cương, Toán rời rạc, Cấu trúc dữ liệu và thuật toán
+ Đã xuất bản :
  "*Lập trình logic trong Prolog*", Nhà Xuất bản Đại học Quốc gia Hà Nội, 2004