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*.

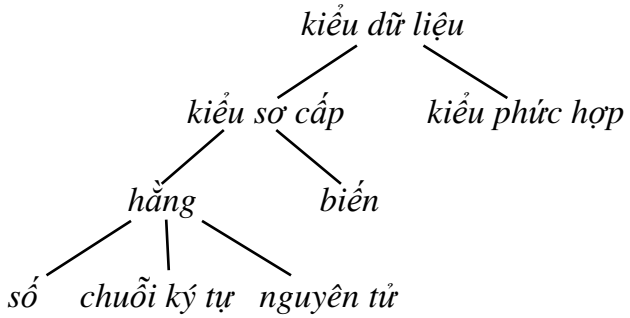
Prolog quy ước viết sau mỗi mệnh đề một dấu chấm để kết thúc như sau :

- 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

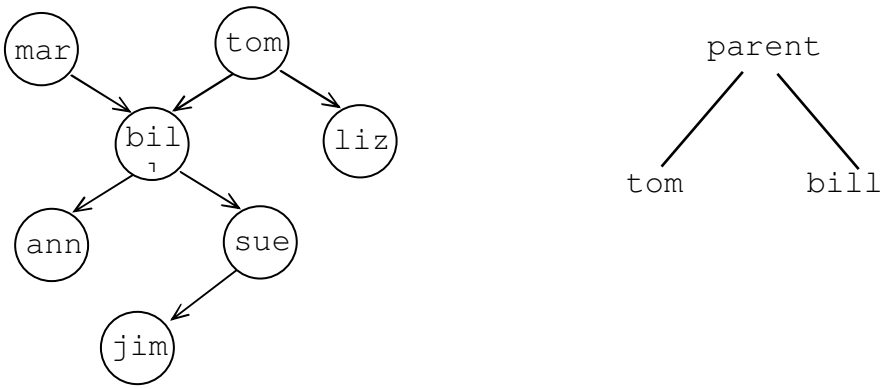
Ví dụ III.1 : Quan hệ gia đình

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`) :

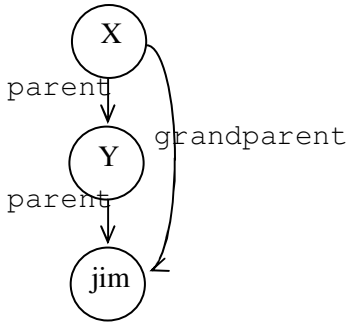
```
X = mary
Y = bill ->;
X = tom
Y = bill ->;
```

```
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.

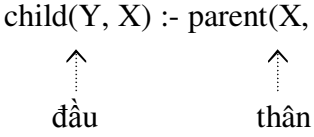
Có sự khác nhau cơ bản giữa sự kiện và luật. Một sự kiện, chẳng hạn :

```
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).
```

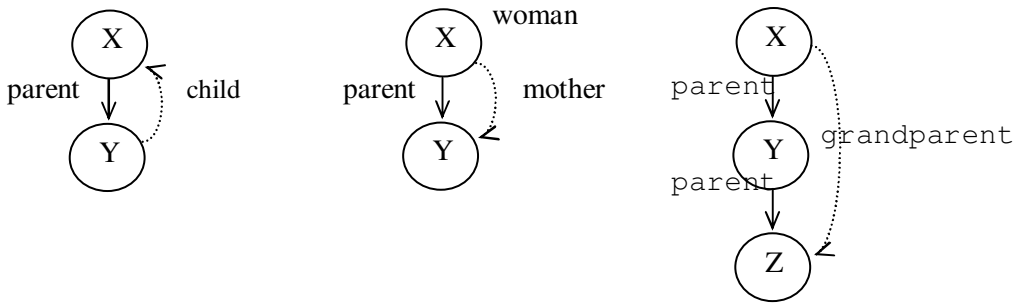
được hiểu là :

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 :

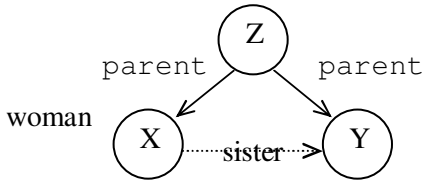
```
grandparent(X, Z) :-
  parent(X, Y),
  parent(Y, Z).
```

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

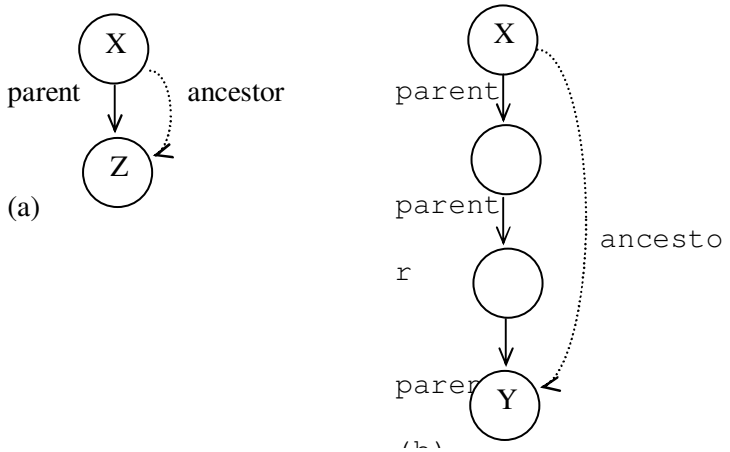
Bây giờ ta tiếp tục thêm một quan hệ mới vào chương trình. Quan hệ này chỉ sử dụng quan hệ `parent`, và chỉ có hai luật. Luật thứ nhất định nghĩa các tổ tiên trực tiếp, luật thứ hai định nghĩa các tổ tiên gián tiếp.

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*.

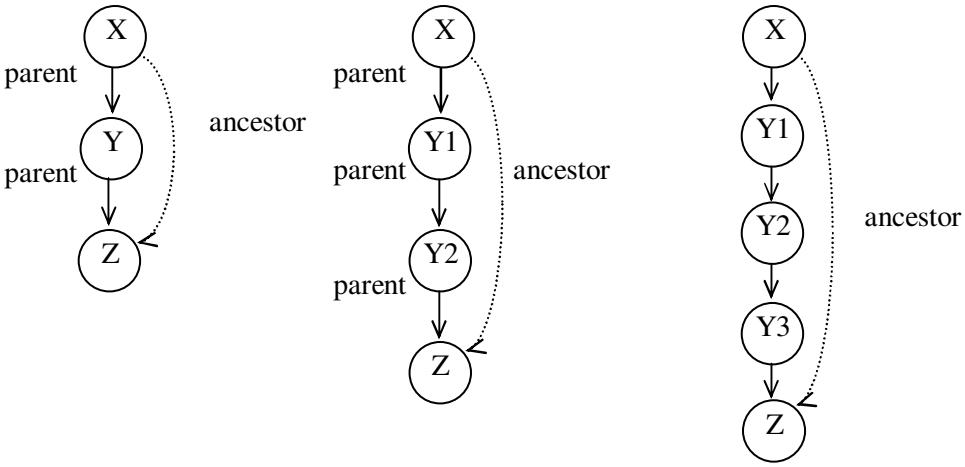
Kể cả luật 1, ta có quan hệ tổ tiên được định nghĩa như sau :

```
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.

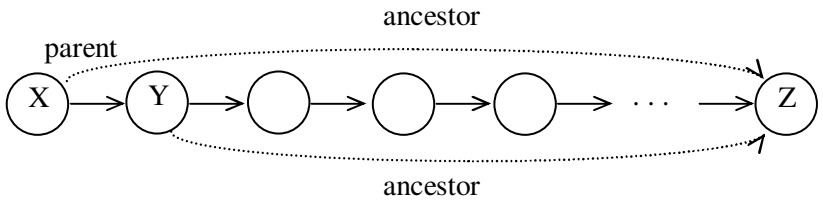
```
ancestor(X, Z) :-
   parent(X, Z).

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`.

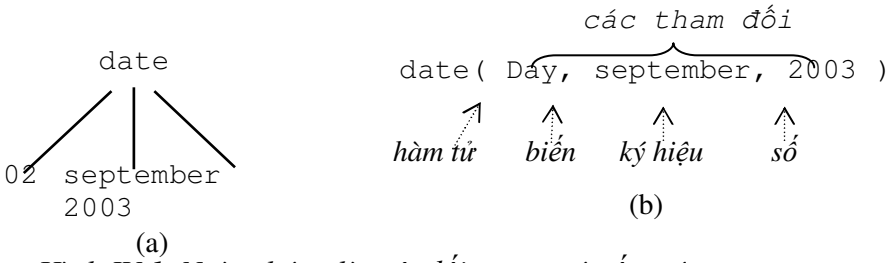
Ngày 2/9/1952 sẽ được viết như sau : `date(2, september, 1952)`

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 :

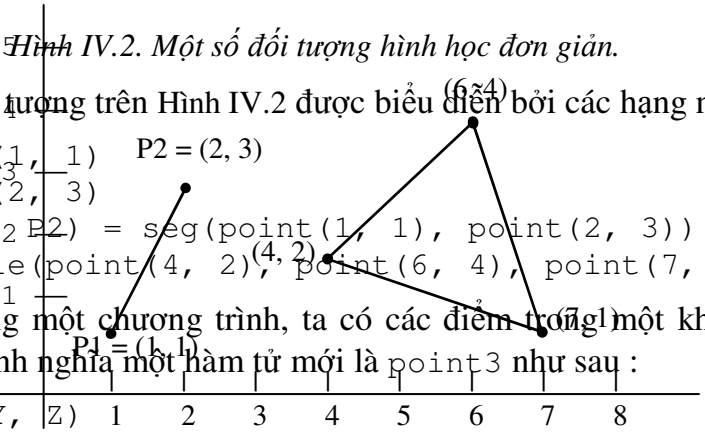
Cấu trúc (đơn giản) của một cuốn sách gồm ba thành phần *tiêu đề* và *tác giả* cũng là các cấu trúc (con), *năm xuất bản* là một biến :

```
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 :

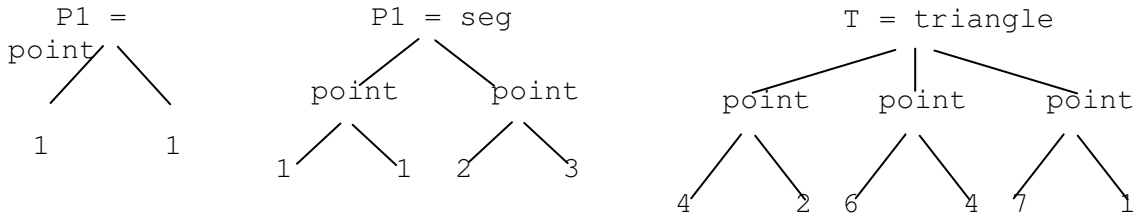
```
point3(X, Y, Z)
```

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 :

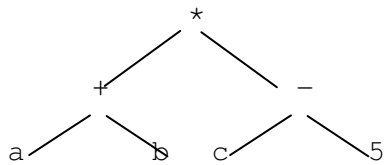


Hình IV.3. Biểu diễn dạng cây của các đối tượng.

(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

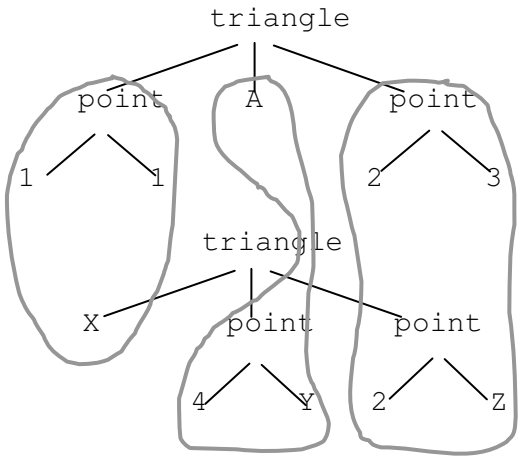
   (b) tất cả các thành phần là khớp nhau từng đôi một.

   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.

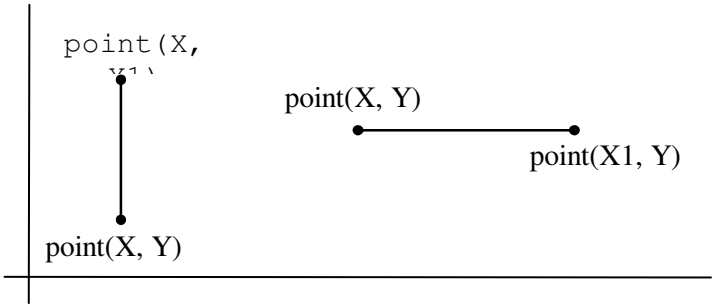
Một đoạn thẳng là thẳng đứng nếu hoành độ (abscissa) của hai mút là bằng nhau, tương tự, là nằm ngang nếu tung độ (ordinate) của hai mút là bằng nhau.

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
```

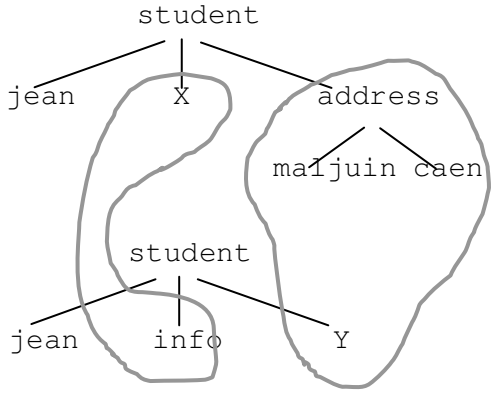
Câu trả lời có nghĩa là mọi đường thẳng có phương trình X = 2 là thẳng đứng. Chú ý rằng ở đây, ta không nhận được tên biến như mong muốn (là Y), mà tuỳ theo phiên bản cài đặt cụ thể, Prolog sẽ tạo ra một tên biến khi thực hiện chương trình, `_0104` trong ví dụ trên, nhằm tránh đặt lại tên biến của NSD 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

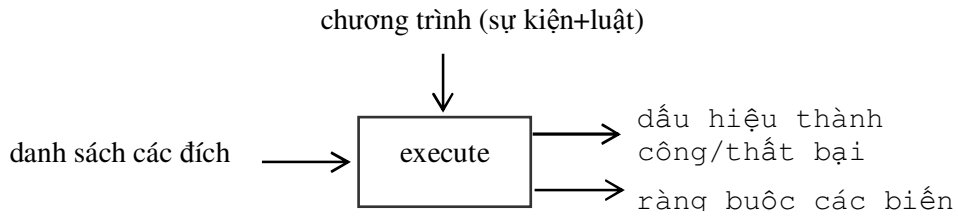
Đích	*Nghĩa lôgich*
`p(a).`	Có phải `p(a)` đúng (thoả mãn) ?
`p(a), Q(b).`	Có phải cả `p(a)` và `Q(b)` đều đúng ?
`P(X).`	Cho biết giá trị của `X` để `P(X)` là đúng ?
`P(X), Q(X, Y).`	Cho biết các giá trị của `X` và của `Y` để `P(X)` và `Q(X, Y)` đều là đúng ?

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).
```

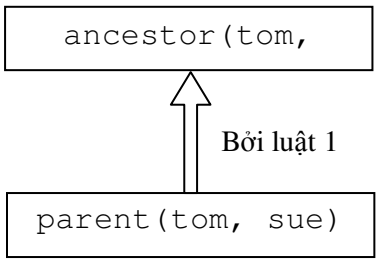
Do đích là `ancestor(tom, sue)`, các biến phải được ràng buộc như sau :

```
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 :

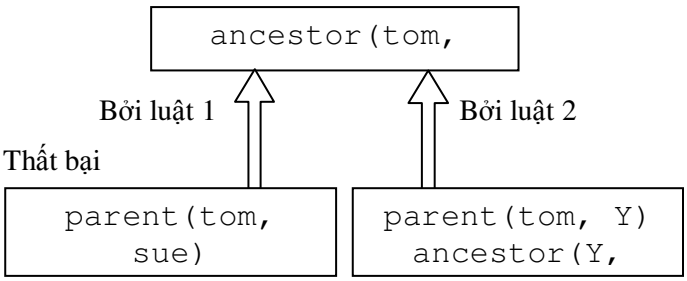
```
X = tom, Z = sue
```

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 :

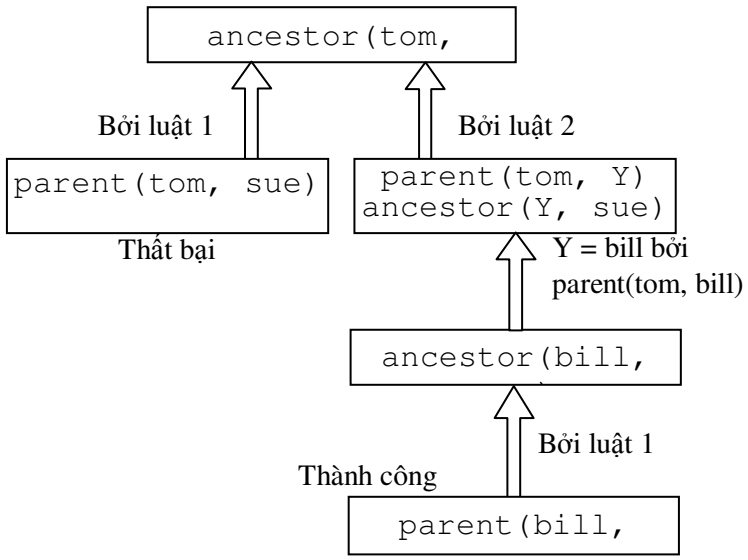
```
ancestor(bill, sue)
```

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

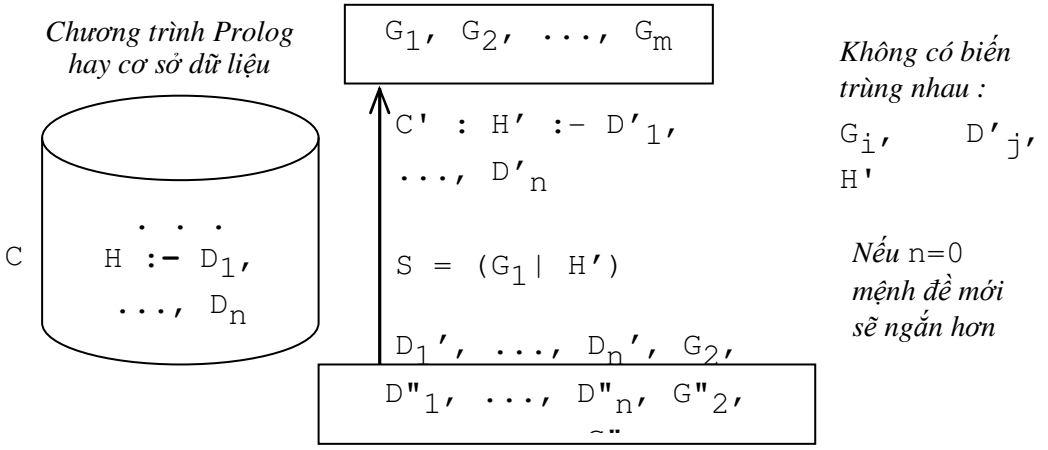
Chú ý rằng nếu C là một sự kiện, khi đó, n=0 và danh sách mới sẽ ngắn hơn danh sách cũ. Trường hợp danh sách mới rỗng, kết quả thành công.

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 :

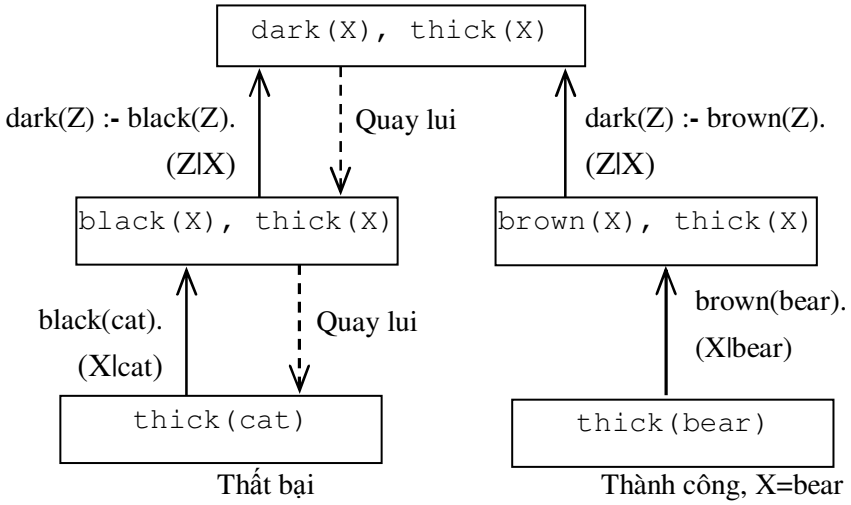
```
dark(Z) :- brown(Z).
```

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).
```

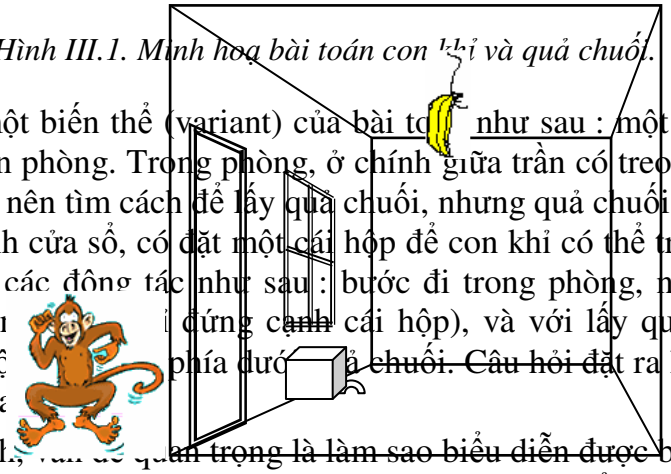
Do những tiến bộ của kỹ thuật lập trình, người ta quan tâm đến nghĩa khai báo để bỏ qua những chi tiết thủ tục, tận dụng những chi tiết khai báo làm lời giải đơn giản hơn và dễ hiểu hơn. Không phải là NLT, mà chính hệ thống phải quản lý những chi tiết thủ tục. Prolog là ngôn ngữ nhằm vào mục đích này. 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.



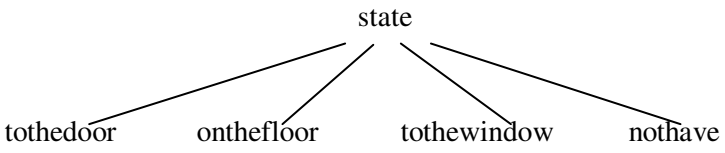
Hình III.1. Minh hoạ bài toán con khỉ và quả chuối.

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).

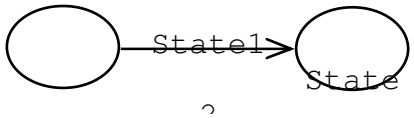
(3) Đẩy cái hộp (pushing).

(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 :

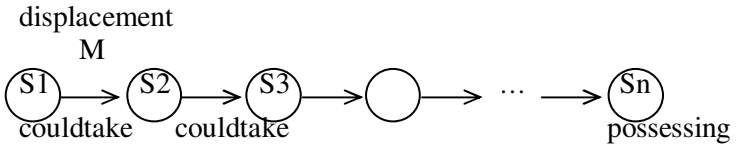
```
displacement(
   state(P1, onthefloor, G, H),
   walking(P1, P2),              % di chuyển từ P1 đến P2
   state(P2, onthefloor, G, H).
```

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 :

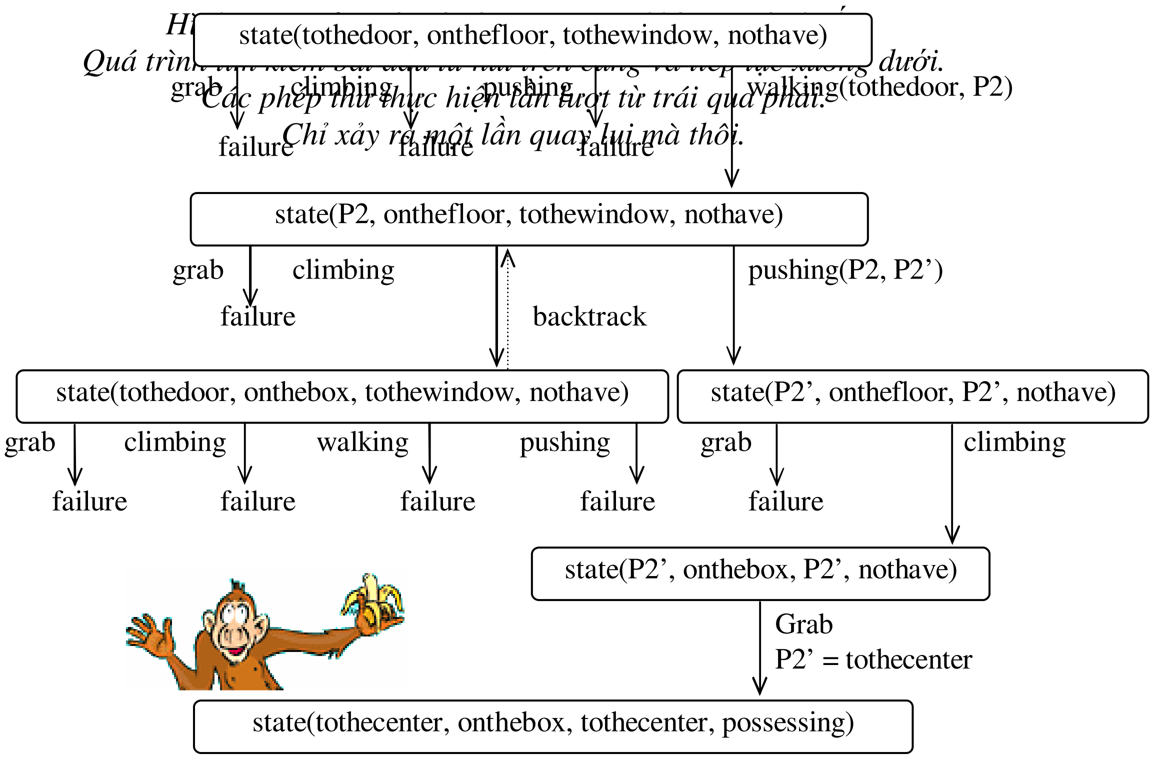
```
displacement(
    state(tothecenter, onthebox, tothecenter, nothave),
    grab,                                   % với lấy quả chuối
    state(tothecenter, onthebox, tothecenter,
    possessing)).
displacement(
    state(P, onthefloor, P, H),
    climbing,                               % trèo lên 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.
state(tothedoor, onthefloor, tothewindow, nothave)
grab
failure
climbing
failure
pushing
failure
walking(tothedoor, P2)
state(P2, onthefloor, tothewindow, nothave)
grab
failure
climbing
backtrack
pushing(P2, P2')
state(tothedoor, onthebox, tothewindow, nothave)
grab
failure
climbing
failure
walking
failure
pushing
failure
state(P2', onthefloor, P2', nothave)
grab
failure
climbing
state(P2', onthebox, P2', nothave)
Grab
P2' = tothecenter
state(tothecenter, onthebox, tothecenter, possessing)

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

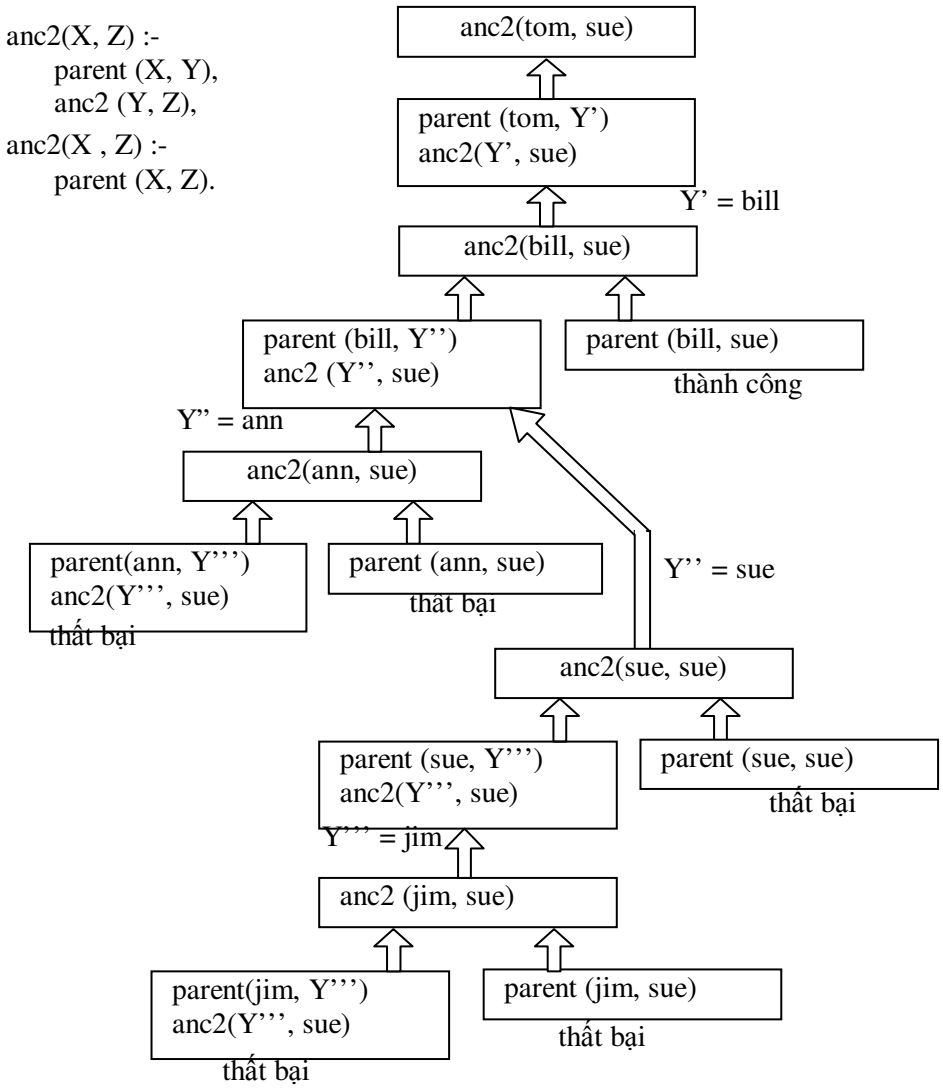
```
anc3(X, Z) :-
   parent(X, Z).
anc3 (X, Z) :-
   anc3 (X, Y),
   parent(Y, Z).
```

% 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
```



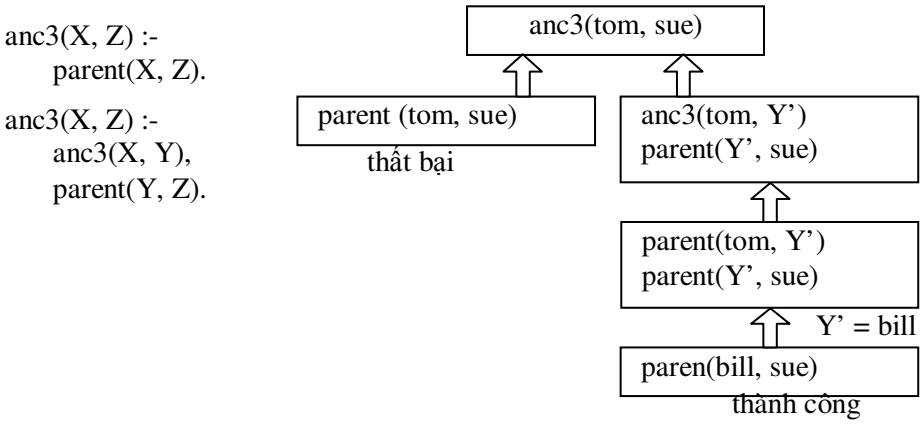
Hình III.6. Biến thể a của quan hệ tổ tiên trả lời câu hỏi "Tom có phải là một tổ tiên của Sue ?"

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.

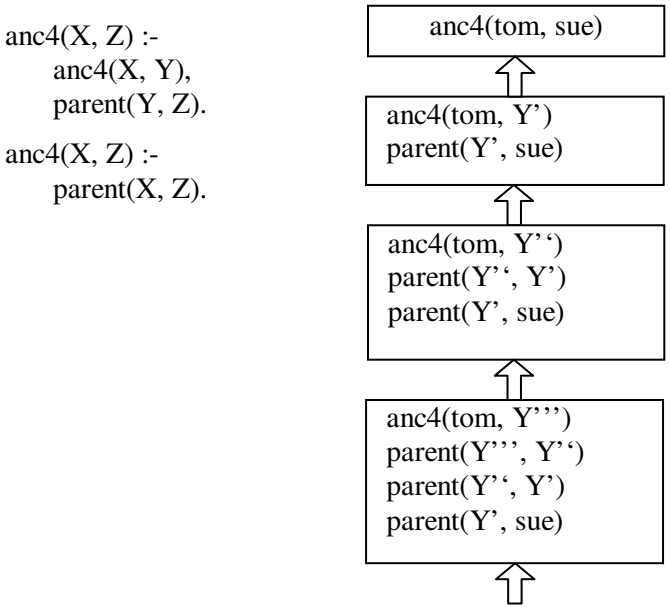
Hình 2.13 (a, b, c) mô tả quá trình thực hiện của `anc2`, `anc3` và `anc4`. Ta thấy `anc4` không có hy vọng và `anc2` kém hiệu quả hơn so với `anc1` do thực hiện nhiều lần tìm kiếm và quay lui hơn trong cây.

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
```

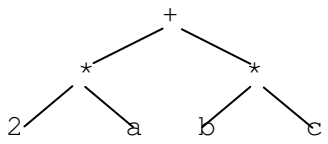
với `+` và `*` là các *phép toán* (operator), còn `a`, `b` và `c` là các *toán hạng* (operand), hay tham đối (argument). Biểu thức trên còn được viết dưới dạng *tiền tố* (prefix) nhờ các hàm tử `+` và `*` như sau :

```
+( *(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)`

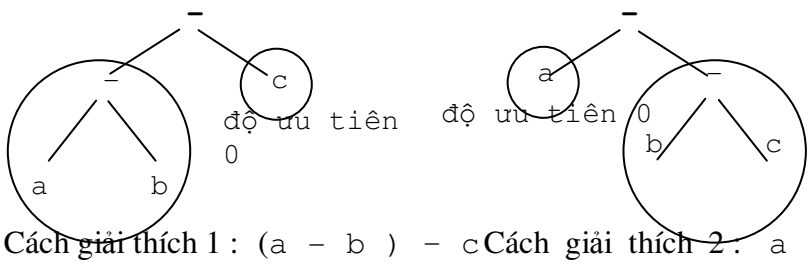
Các quy tắc trên cho phép loại bỏ tính nhập nhằng của các biểu thức chứa các phép toán có cùng độ ưu tiên. Chẳng hạn :

```
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.

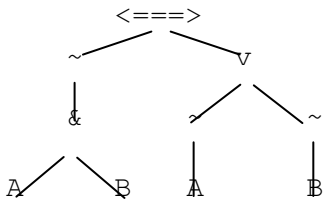
Để minh hoạ, ta xét ví dụ viết một chương trình xử lý các biểu thức lôgich (boolean). Giả sử ta cần biểu diễn một trong các định lý tương đương của Morgan, được viết dưới dạng toán học như sau :

~ (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

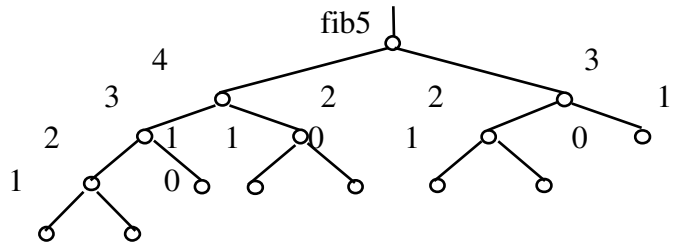
Lời giải các bài toán sử dụng đệ quy trong các ngôn ngữ lập trình nói chung thường ngắn gọn, dễ hiểu và dễ quản lý được chương trình. Tuy nhiên, trong một số trường hợp, sử dụng đệ quy lại xảy ra vấn đề về độ phức tạp tính toán, không những tốn kém bộ nhớ mà còn tốn kém thời gian.

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 :

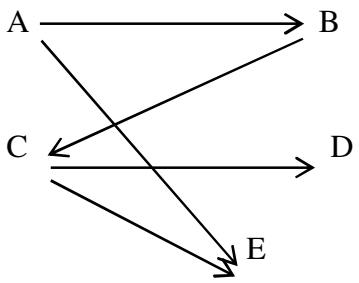
```
fibo(0, 0).
fibo(N, F) :-
    N >= 1,
    fib1(N, 1, 0, F).
fib1(1, F, _, F).
fib1(N, F2, F1, FN) :-
```

```
    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.