

## CHƯƠNG 4

---

# Cấu trúc danh sách

Chương này trình bày khái niệm về *danh sách*, một trong những cấu trúc đơn giản nhất và thông dụng nhất, cùng với những chương trình tiêu biểu minh họa cách vận dụng danh sách trong Prolog. Cấu trúc danh sách tạo nên một môi trường lập trình thuận tiện của ngôn ngữ Prolog.

## I. Biểu diễn cấu trúc danh sách

Danh sách là kiểu cấu trúc dữ liệu được sử dụng rộng rãi trong các ngôn ngữ lập trình phi số. Một danh sách là một dãy bất kỳ các đối tượng. Khác với kiểu dữ liệu tập hợp, các đối tượng của danh sách có thể trùng nhau (xuất hiện nhiều lần) và mỗi vị trí xuất hiện của đối tượng đều có ý nghĩa.

Danh sách là cách diễn đạt ngắn gọn của kiểu dữ liệu hạng phức hợp trong Prolog. Hàm tử của danh sách là dấu chấm ".". Do việc biểu diễn danh sách bởi hàm tử này có thể tạo ra những biểu thức mập mờ, nhất là khi xử lý các danh sách gồm nhiều phần tử lồng nhau, cho nên Prolog quy ước đặt dãy các phần tử của danh sách giữa các cặp móc vuông.

Chẳng hạn `.(a,.(b,[ ])).` Là danh sách `[ a, b ]`.

Danh sách các phần tử `anne`, `tennis`, `tom`, `skier` (tên người) được viết :

`[ anne, tennis, tom, skier ]`

chính là hàm tử :

`.( anne, .( tennis, .( tom, .( skier, [ ] ) ) ) )`

Cách viết dạng cặp móc vuông chỉ là xuất hiện bên ngoài của một danh sách. Như đã thấy ở mục trước, mọi đối tượng cấu trúc của Prolog đều có biểu diễn cây. Danh sách cũng không nằm ngoại lệ, cũng có cấu trúc cây.

Làm cách nào để biểu diễn danh sách bởi một đối tượng Prolog chuẩn ? Có hai khả năng xảy ra là danh sách có thể rỗng hoặc không. Nếu danh sách rỗng, nó được viết dưới dạng một nguyên tử :

[ ]

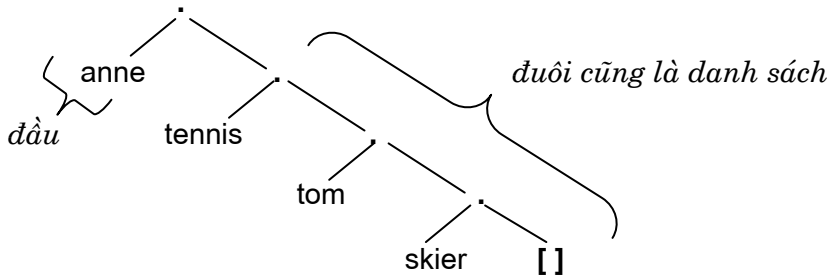
Nếu danh sách khác rỗng, có thể xem nó được cấu trúc từ hai thành phần (pair syntax) :

1. Thành phần thứ nhất, được gọi là *đầu* (head) của danh sách.
2. Thành phần thứ hai, phần còn lại của danh sách (trừ ra phần đầu), được gọi là *đuôi* (tail) của danh sách, cũng là một danh sách.

Trong ví dụ trên thì đầu là anne, còn đuôi là danh sách :

[ tennis, tom, skier ]

Nói chung, đầu của danh sách có thể là một đối tượng bất kỳ của Prolog, có thể là cây hoặc biến, nhưng đuôi phải là một danh sách. Hình I.1. Biểu diễn dạng cây của danh sách mô tả cấu trúc cây của danh sách đã cho :



Hình I.1. Biểu diễn dạng cây của danh sách

Vì đuôi tail là một danh sách, nên tail có thể rỗng, hoặc lại có thể được tạo thành từ một đầu head và một đuôi tail khác.

Chú ý rằng danh sách rỗng xuất hiện trong số các hạng, vì rằng phần tử cuối cùng có thể xem là danh sách chỉ gồm một phần tử duy nhất có phần đuôi là một danh sách rỗng:

[ skier ]

Ví dụ trên đây minh họa nguyên lý cấu trúc dữ liệu tổng quát trong Prolog áp dụng cho các danh sách có độ dài tùy ý.

?- L1 = [ a, b, c ].

?- L2 = [ a, a, a ].

$$L1 = [a, b, c]$$

$$L2 = [a, a, a]$$

$$?- \text{Leisure1} = [\text{tennis}, \text{music}, []].$$

$$?- \text{Leisure2} = [\text{sky}, \text{eating}],$$

$$?- L = [\text{anne}, \text{Leisure1}, \text{tom}, \text{Leisure2}].$$

$$\text{Leisure1} = [\text{tennis}, \text{music}]$$

$$\text{Leisure2} = [\text{sky}, \text{eating}]$$

$$L = [\text{anne}, [\text{tennis}, \text{music}], \text{tom}, [\text{sky}, \text{eating}]]$$

Như vậy, các phần tử của một danh sách có thể là các đối tượng có kiểu bất kỳ, kể cả kiểu danh sách. Thông thường, người ta xử lý đuôi của danh sách như là một danh sách. Chẳng hạn, danh sách :

$$L = [a, b, c]$$

có thể viết :

$$\text{tail} = [b, c] \text{ và } L = .(a, \text{tail})$$

Để biểu diễn một danh sách được tạo thành từ đầu (Head) và đuôi (Tail), Prolog sử dụng ký hiệu  $|$  (split) để phân cách phần đầu và phần đuôi như sau :

$$L = [a | \text{Tail}]$$

Ký hiệu  $|$  được dùng một cách rất tổng quát bằng cách viết một số phần tử tùy ý của danh sách trước  $|$  rồi danh sách các phần tử còn lại. Danh sách bây giờ được viết lại như sau :

$$[a, b, c] = [a | [b, c]] = [a, b | [c]] = [a, b, c | []]$$

Sau đây là một số cách viết danh sách :

<i>Kiểu hai thành phần</i>	<i>Kiểu liệt kê phần tử</i>
$[]$	$[]$
$[a   []]$	$[a]$
$[a   b   []]$	$[a, b]$
$[a   X]$	$[a   X]$
$[a   b   X]$	$[a, b   X]$
$[X_1   [\dots [X_n   []] \dots]]$	$[X_1, \dots, X_n]$

Ta có thể định nghĩa danh sách theo kiểu đệ quy như sau :

$$\text{List} \rightarrow []$$

$$\text{List} \rightarrow [\text{Element} | \text{List}]$$

## II. Một số vị từ xử lý danh sách của Prolog

SWI-Prolog có sẵn một số vị từ xử lý danh sách như sau :

Vị từ	Ý nghĩa
<code>append(List1, List2, List3)</code>	Ghép hai danh sách List1 và List2 thành List3.
<code>member(Elem, List)</code>	Kiểm tra Elem có là phần tử của danh sách List hay không, nghĩa là Elem hợp nhất được với một trong các phần tử của List.
<code>nextto(X, Y, List)</code>	Kiểm tra nếu phần tử Y có đứng ngay sau phần tử X trong danh sách List hay không.
<code>delete(List1, Elem, List2)</code>	Xoá khỏi danh sách List1 những phần tử hợp nhất được với Elem để trả về kết quả List2.
<code>select(Elem, List, Rest)</code>	Lấy phần tử Elem ra khỏi danh sách List để trả về những phần tử còn lại trong Rest, có thể dùng để chèn một phần tử vào danh sách.
<code>nth0(Index, List, Elem)</code>	Kiểm tra phần tử thứ Index (tính từ 0) của danh sách List có phải là Elem hay không.
<code>nth1(Index, List, Elem)</code>	Kiểm tra phần tử thứ Index (tính từ 1) của danh sách List có phải là Elem hay không.
<code>last(List, Elem)</code>	Kiểm tra phần tử đứng cuối cùng trong danh sách List có phải là Elem hay không.
<code>reverse(List1, List2)</code>	Nghịch đảo thứ tự các phần tử của danh sách List1 để trả về kết quả List2.
<code>permutation(List1, List2)</code>	Hoán vị danh sách List1 thành danh sách List2.
<code>flatten(List1, List2)</code>	Chuyển danh sách List1 chứa các phần tử bất kỳ thành danh sách phẳng List2. Ví dụ : <code>flatten([a, [b, [c, d], e]], X)</code> . cho kết quả <code>X = [a, b, c, d, e]</code> .
<code>sumlist(List, Sum)</code>	Tính tổng các phần tử của danh sách List chứa toàn số để trả về kết quả Sum.
<code>numlist(Low, High, List)</code>	Nếu Low và High là các số sao cho $Low \leq High$ , thì trả về danh sách <code>List = [Low, Low+1, ..., High]</code> .

Chú ý một số vị từ xử lý danh sách có thể sử dụng cho mọi ràng buộc, kể cả khi các tham đối đều là biến.

Trong Prolog, tập hợp được biểu diễn bởi danh sách, tuy nhiên, thứ tự các phần tử trong một tập hợp là không quan trọng, các đối tượng dù xuất hiện

nhiều lần chỉ được xem là một phần tử của tập hợp. Các phép toán về danh sách có thể áp dụng cho các tập hợp. Đó là :

- Kiểm tra một phần tử có mặt trong một danh sách tương tự việc kiểm tra một phần tử có thuộc về một tập hợp không ?
- Ghép hai danh sách để nhận được một danh sách thứ ba tương ứng với phép hợp của hai tập hợp.
- Thêm một phần tử mới, hay loại bỏ một phần tử.

Prolog có sẵn một số vị từ xử lý tập hợp như sau :

Vị từ	Ý nghĩa
is_set(Set)	Kiểm tra Set có phải là một tập hợp hay không
list_to_set(List, Set)	Chuyển danh sách List thành tập hợp Set giữ nguyên thứ tự các phần tử của List (nếu List có các phần tử trùng nhau thì chỉ lấy phần tử gặp đầu tiên). Ví dụ : list_to_set([a,b,a], X) cho kết quả $X = [a,b]$ .
intersection(Set1, Set2, Set3)	Phép giao của hai tập hợp Set1 và Set2 là Set3.
subtract(Set, Delete, Result)	Trả về kết quả phép hiệu của hai tập hợp Set và Delete là Result (là tập Set sau khi đã xóa hết các phần tử của Delete có mặt trong đó).
union(Set1, Set2, Set3)	Trả về kết quả phép hợp của hai tập hợp Set1 và Set2 là Set3.
subset(Subset, Set)	Kiểm tra tập hợp Subset có là tập hợp con của Set hay không.

### III. Các thao tác cơ bản trên danh sách

#### III.1. Xây dựng lại một số vị từ có sẵn

Sau đây ta sẽ trình bày một số thao tác cơ bản trên danh sách bằng cách xây dựng lại một số vị từ có sẵn của Prolog.

##### III.1.1. Kiểm tra một phần tử có mặt trong danh sách

Prolog kiểm tra một phần tử có mặt trong một danh sách như sau :

member(X, L)

trong đó,  $X$  là một phần tử và  $L$  là một danh sách. Đích  $\text{member}(X, L)$  được thoả mãn nếu  $X$  xuất hiện trong  $L$ . Ví dụ :

?-  $\text{member}(b, [a, b, c])$

Yes

?-  $\text{member}(b, [a, [b, c]])$

No

?-  $\text{member}([b, c], [a, [b, c]])$

Yes

Từ các kết quả trên, ta có thể giải thích quan hệ  $\text{member}(X, L)$  như sau :

Phần tử  $X$  thuộc danh sách  $L$  nếu :

1.  $X$  là đầu của  $L$ , hoặc nếu
2.  $X$  là một phần tử của đuôi của  $L$ .

Ta có thể viết hai điều kiện trên thành hai mệnh đề, mệnh đề thứ nhất là một sự kiện đơn giản, mệnh đề thứ hai là một luật :

$\text{member}(X, [X | \text{Tail}])$ .

$\text{member}(X, [\text{Head} | \text{Tail}]) :- \text{member}(X, \text{Tail})$ .

hoặc :

$\text{member}(X, [X|\text{T}])$ .

$\text{member}(X, [_|\text{T}]) :- \text{member}(X, \text{T})$ .

### III.1.2. Ghép hai danh sách

Để ghép hai danh sách, Prolog có hàm :

$\text{append}(L1, L2, L3)$ .

trong đó,  $L1$  và  $L2$  là hai danh sách,  $L3$  là kết quả của phép ghép  $L1$  và  $L2$ . Ví dụ :

?-  $\text{append}([a, b], [c, d], [a, b, c, d])$ .

Yes

?-  $\text{append}([a, b], [c, d], [a, b, a, c])$ .

No

Hàm  $\text{append}$  hoạt động phụ thuộc tham đối đầu tiên  $L1$  theo cách như sau :

1. Nếu tham đối đầu tiên là danh sách rỗng, thì tham đối thứ hai và thứ ba phải là một danh sách duy nhất, gọi là  $L$ . Ta viết trong Prolog như sau :

$\text{append}([], L, L)$ .

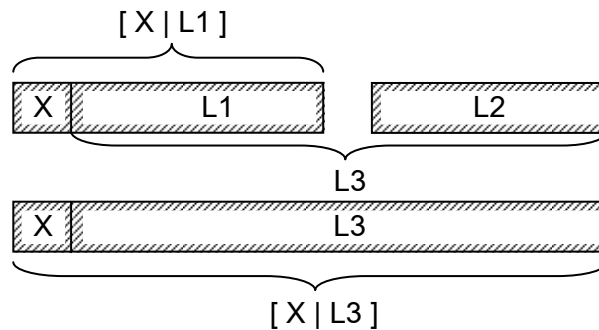
2. Nếu tham đối đầu tiên của append là danh sách khác rỗng, thì nó gồm một đầu và một đuôi như sau

$[X | L1]$

Kết quả phép ghép danh sách là danh sách  $[X | L3]$ , với  $L3$  là phép ghép của  $L1$  và  $L2$ . Ta viết trong Prolog như sau :

`append([X | L1], L2, [X | L3]) :- append(L1, L2, L3).`

Hình 4.2 dưới đây minh hoạ phép ghép hai danh sách  $[X | L1]$  và  $L2$ .



Hình III.1. Ghép hai danh sách  $[X | L1]$  và  $L2$  thành  $[X | L3]$ .

Ta có các ví dụ sau :

?- `append([a, b, c], [1, 2, 3], L).`

`L = [a, b, c, 1, 2, 3]`

?- `append([a, [b, c], d], [a, [], b], L).`

`L = [a, [b, c], d, a, [], b]`

Thủ tục append được sử dụng rất mềm dẻo theo nhiều cách khác nhau.

Chẳng hạn Prolog đưa ra bốn phương án để phân tách một danh sách đã cho thành hai danh sách mới như sau :

?- `append(L1, L2, [a, b, c]).`

`L1 = []`

`L2 = [a, b, c];`

`L1 = [a]`

`L2 = [b, c];`

`L1 = [a, b]`

`L2 = [c];`

`L1 = [a, b, c]`

`L2 = [];`

Yes

Sử dụng `append`, ta cũng có thể tìm kiếm một số phần tử trong một danh sách. Chẳng hạn, từ danh sách các tháng trong năm, ta có thể tìm những tháng đứng trước một tháng đã cho, giả sử tháng năm (May) :

```
?- append( Before, [ May | After ] ,
           [ jan, fev, mar, avr, may, jun, jul, aut, sep, oct, nov, dec ] ).
Before = [ jan, fev, mar, avr ]
After = [ jun, jul, aut, sep, oct, nov, dec ]
Yes
```

Tháng đứng ngay trước và tháng đứng ngay sau tháng năm nhận được như sau :

```
?- append( _, [ Month1, may, Month2 | _ ] ,
           [ jan, fev, mar, avr, may, jun, jul, aut, sep, oct, nov, dec ] ).
Month1 = avr
Month2 = jun
Yes
```

Bây giờ cho trước danh sách :

```
L1 = [ a, b, z, z, c, z, z, z, d, e ]
```

Ta cần xóa các phần tử đứng sau ba chữ z liên tiếp, kể cả ba chữ z :

```
?- L1 = [ a, b, z, z, c, z, z, z, d, e ],
   append( L2, [ z, z, z | _ ], L1 ).
L1 = [ a, b, z, z, c, z, z, z, d, e ]
L2 = [ a, b, z, z, c ]
```

Trước đây ta đã định nghĩa quan hệ `member( X, L )` để kiểm tra một phần tử X có mặt trong một danh sách L không. Bây giờ bằng cách sử dụng `append`, ta có thể định nghĩa lại `member` như sau :

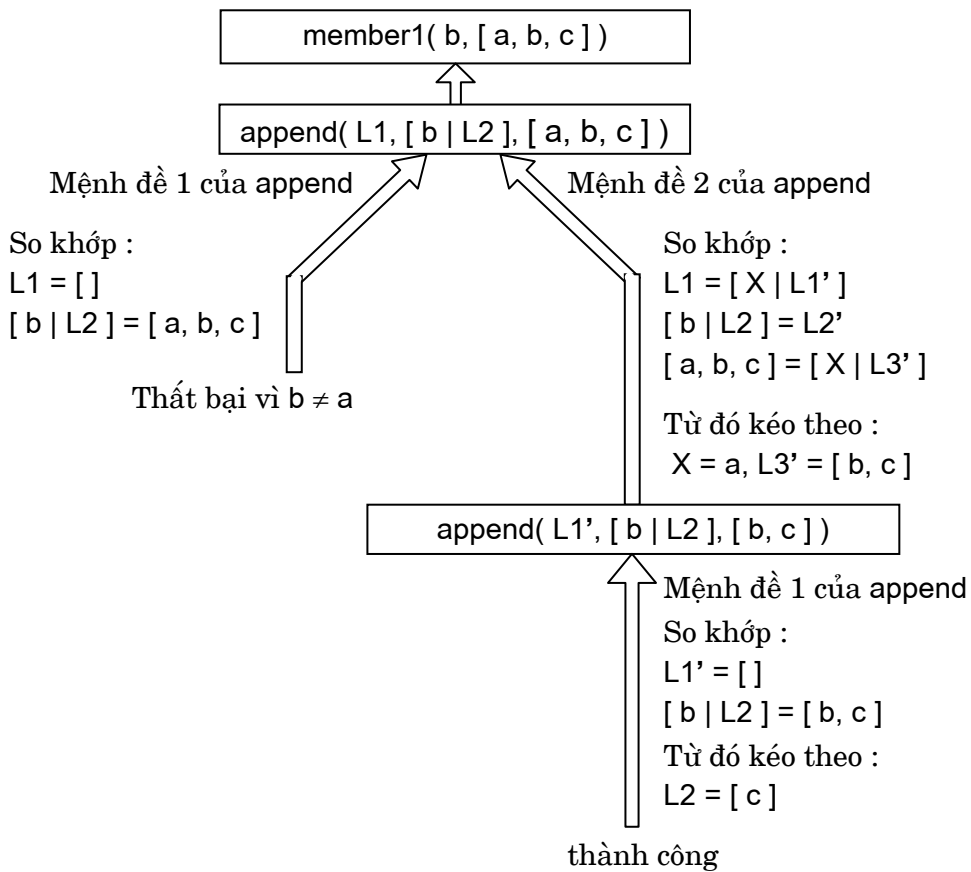
```
member1( X, L ) :- append( L1, [ X | L2 ], L ).
```

Mệnh đề này có nghĩa : nếu X có mặt trong danh sách L thì L có thể được phân tách thành hai danh sách, với X là đầu của danh sách thứ hai. Định nghĩa `member1` hoàn toàn tương đương với định nghĩa `member`.

Ở đây ta sử dụng hai tên khác nhau để phân biệt hai cách cài đặt Prolog. Ta cũng có thể định nghĩa lại `member1` bằng cách sử dụng biến nặc danh (anonymous variable) :

```
member1( X, L ) :- append( _ , [ X | _ ], L ).
```





Hình III.2. Thủ tục member1 tìm tuần tự một đối tượng trong danh sách đã cho.

So sánh hai cách cài đặt khác nhau về quan hệ thành viên, ta nhận thấy nghĩa thủ tục trong định nghĩa member được thể hiện rất rõ :

Trong member, để kiểm tra phần tử X có mặt trong một danh sách L không,

1. Trước tiên kiểm tra phần tử đầu của L là đồng nhất với X, nếu không,
2. Kiểm tra rằng X có mặt trong phần đuôi của L.

Nhưng trong trường hợp định nghĩa member1, ta thấy hoàn toàn nghĩa khai báo mà không có nghĩa thủ tục.

Để hiểu được cách member1 hoạt động như thế nào, ta hãy xem xét quá trình Prolog thực hiện câu hỏi :

?- member1( b, [ a, b, c ] ).

Cách tìm của thủ tục member1 trên đây tương tự member, bằng cách duyệt từng phần tử, cho đến khi tìm thấy đối tượng cần tìm, hoặc danh sách đã cạn.

### III.1.3. Bổ sung một phần tử vào danh sách

Phương pháp đơn giản nhất để bổ sung một phần tử vào danh sách là đặt nó ở vị trí đầu tiên, để nó trở thành đầu. Nếu X là một đối tượng mới, còn L là danh sách cần bổ sung thêm, thì danh sách kết quả sẽ là :

$[X | L]$

Người ta không cần viết thủ tục để bổ sung một phần tử vào danh sách. Bởi vì việc bổ sung có thể được biểu diễn dưới dạng một sự kiện nếu cần :

`insert( X, L, [ X | L ] ).`

### III.1.4. Loại bỏ một phần tử khỏi danh sách

Để loại bỏ một phần tử X khỏi danh sách L, người ta xây dựng quan hệ :

`remove( X, L, L1 )`

trong đó, L1 đồng nhất với L, sau khi X bị loại bỏ khỏi L. Thủ tục remove có cấu trúc tương tự member. Ta có thể lập luận như sau

1. Nếu phần tử X là đầu của danh sách, thì kết quả là đuôi của danh sách.
2. Nếu không, tìm cách loại bỏ X khỏi phần đuôi của danh sách.

`remove( X, [ X | Tail ], Tail ).`

`remove( X, [ Y | Tail ], [ Y | Tail1 ] ) :-`

`remove( X, Tail, Tail1 ).`

Tương tự thủ tục member, thủ tục remove mang tính không xác định. Nếu có nhiều phần tử là X có mặt trong danh sách, thì remove có thể xoá bất kỳ phần tử nào, do quá trình quay lui. Tuy nhiên, mỗi lần thực hiện, remove chỉ xoá một phần tử là X mà không đụng đến những phần tử khác. Ví dụ :

?- `remove( a, [ a, b, a, a ], L ).`

`L = [ b, a, a ];`

`L = [ a, b, a ];`

`L = [ a, b, a ]`

No

Thủ tục remove thất bại nếu danh sách không chứa phần tử cần xoá. Người ta có thể sử dụng remove trong một khía cạnh khác, mục đích để bổ sung một phần tử mới vào bất cứ đâu trong danh sách.

Ví dụ, nếu ta muốn đặt phần tử a vào tại mọi vị trí bất kỳ trong danh sách [ 1, 2, 3 ], chỉ cần đặt câu hỏi : Cho biết danh sách L nếu sau khi xoá a, ta nhận được danh sách [ 1, 2, 3 ] ?

?- remove( a, L, [ 1, 2, 3 ] ).

L = [ a, 1, 2, 3 ];

L = [ 1, a, 2, 3 ];

L = [ 1, 2, a, 3 ];

L = [ 1, 2, 3, a ]

No

Một cách tổng quát, phép toán chèn insert một phần tử X vào một danh sách List được định nghĩa bởi thủ tục remove bằng cách sử dụng một danh sách lớn hơn LargerList làm tham đối thứ hai :

insert( X, List, LargerList ) :-  
remove( X, LargerList, List ).

Ta đã định nghĩa quan hệ thuộc về trong thủ tục member1 bằng cách sử dụng thủ tục append. Tuy nhiên, ta cũng có thể định nghĩa lại quan hệ thuộc về trong thủ tục mới member2 bởi thủ tục remove bằng cách xem một phần tử X thuộc về một danh sách List nếu X bị xoá khỏi List :

member2( X, List ) :-  
remove( X, List, \_ ).

### III.1.5. Nghịch đảo danh sách

Sử dụng append, ta có thể viết thủ tục nghịch đảo một danh sách như sau :

reverse ( [ ], [ ] ).

reverse ( [ X | Tail ], R ) :-

reverse ( Tail, R1 ),

append(R1, [X], R).

?- reverse( [ a, b, c, d, e, f ], L).

L = [f, e, d, c, b, a]

Yes

Sau đây là một thủ tục khác để nghịch đảo một danh sách nhưng có sử dụng hàm hỗ trợ trong thân thủ tục :

```

revert(List, RevList) :-
    rev(List, [ ], RevList).
rev([ ], R, R).
rev([H|T], S, R) :-
    rev(T, [H|S], R).
?- revert( [ a, b, c, d, e, f ], R).
R = [f, e, d, c, b, a]
Yes

```

Sử dụng reverse, ta có thể kiểm tra một danh sách có là đối xứng (palindrome) hay không :

```

palindrome(L) :-
    reverse( L, L ).
?- palindrome([ a, b, c, d, c, b, a ]).
Yes

```

### III.1.6. Danh sách con

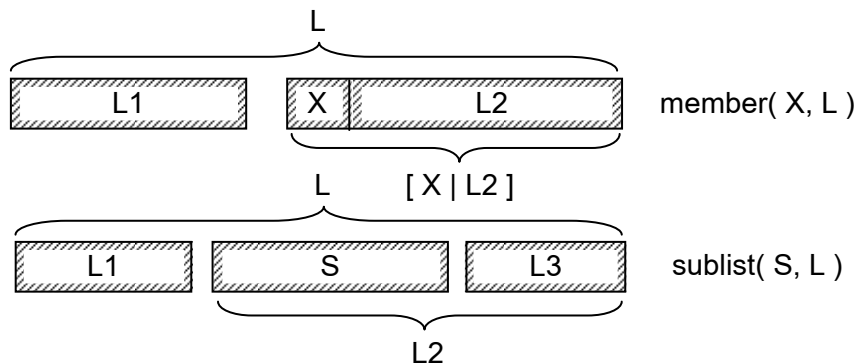
Ta xây dựng thủ tục sublist nhận hai tham đối là hai danh sách L và S sao cho S là danh sách con của L như sau :

```

?- sublist( [ c, d, e ], [ a, b, c, d, e, f ] )
Yes
?- sublist( [ c, e ], [ a, b, c, d, e, f ] )
No

```

Nguyên lý để xây dựng thủ tục sublist tương tự thủ tục member1, mặc dù ở đây quan hệ danh sách con tổng quát hơn.



Hình III.3. Các quan hệ member và sublist.

Quan hệ danh sách con được mô tả như sau :

S là một danh sách con của L nếu :

1. Danh sách L có thể được phân tách thành hai danh sách L1 và L2, và nếu
2. Danh sách L2 có thể được phân tách thành hai danh sách S và L3.

Như đã thấy, việc phân tách các danh sách có thể được mô tả bởi quan hệ ghép append.

Do đó ta viết lại trong Prolog như sau :

```
sublist( S, L ) :-
    append( L1, L2, L ), append( S, L3, L2 ).
```

Ta thấy thủ tục sublist rất mềm dẻo và do vậy có thể sử dụng theo nhiều cách khác nhau. Chẳng hạn ta có thể liệt kê mọi danh sách con của một danh sách đã cho như sau :

```
?- sublist( S, [ a, b, c ] ).
S = [ ];
S = [ a ];
S = [ a, b ];
S = [ a, b, c ];
S = [ b ];
...
```

### III.2. Hoán vị

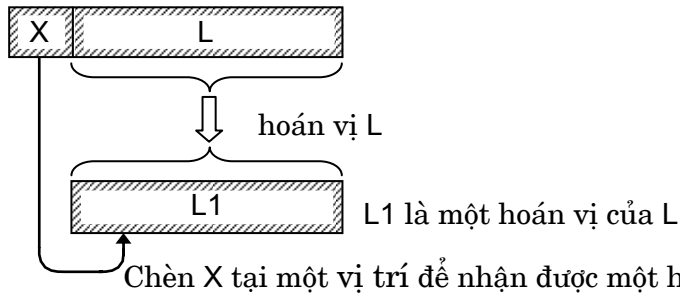
Đôi khi, ta cần tạo ra các hoán vị của một danh sách. Ta xây dựng quan hệ permutation có hai tham biến là hai danh sách, mà một danh sách là hoán vị của danh sách kia. Ta sẽ tận dụng phép quay lui như sau :

```
?- permutation( [ a, b, c ], P ).
P = [ a, b, c ];
P = [ a, c, b ];
P = [ b, a, c ];
...
```

Nguyên lý hoạt động của thủ tục swap dựa trên hai trường hợp phân biệt, tùy theo danh sách thứ nhất :

1. Nếu danh sách thứ nhất rỗng, thì danh sách thứ hai cũng phải rỗng.

2. Nếu danh sách thứ nhất khác rỗng, thì nó sẽ có dạng  $[X | L]$  và được tiến hành hoán vị như sau : trước tiên hoán vị  $L$  để nhận được  $L1$ , sau đó chèn  $X$  vào tất cả các vị trí trong  $L1$ .



Hình III.4. Một cách xây dựng hoán vị permutation của danh sách  $[X | L]$ .

Ta nhận được hai mệnh đề tương ứng với thủ tục như sau :

```
permutation( [ ], [ ] ).
permutation( [ X | L ], P ) :-
    permutation( L, L1 ), insert( X, L1, P ).
```

Một phương pháp khác là loại bỏ phần tử  $X$  khỏi danh sách đầu tiên, hoán vị phần còn lại của danh sách này để nhận được danh sách  $P$ , sau đó thêm  $X$  vào phần đầu của  $P$ . Ta có chương trình khác permutation2 nhæ sau :

```
permutation2( [ ], [ ] ).
permutation2( L, [ X | P ] ) :-
    remove( X, L, L1 ), permutation2( L1, P ).
```

Từ đây, ta có thể khai thác thủ tục hoán vị, chẳng hạn (chú ý khi chạy Arity Prolog cần gõ vào một dấu chấm phẩy ; sau ->) :

```
?- permutation( [ red, blue, green ], P ).
P = [ red, blue, green ];
P = [ red, green, blue ];
P = [ blue, red, green ];
P = [ blue, green, red ];
P = [ green, red, blue ];
P = [ green, blue, red ];
Yes
```

Hoặc nếu sử dụng permutation theo cách khác như sau :

```
?- permutation( L, [ a, b, c ] ).
```

Prolog sẽ ràng buộc liên tiếp cho  $L$  để đưa ra 6 hoán vị khác nhau có thể. Tuy nhiên, nếu NSD yêu cầu một giải pháp khác, Prolog sẽ không bao giờ

trả lời "No", mà rơi vào một vòng lặp vô hạn do phải tìm kiếm một hoán vị mới mà thực ra không tồn tại. Trong trường hợp này, thủ tục permutation2 chỉ tìm thấy một hoán vị thứ nhất, sau đó ngay lập tức rơi vào một vòng lặp vô hạn. Vì vậy, cần chú ý khi sử dụng các quan hệ hoán vị này.

### III.3. Một số ví dụ về danh sách

#### III.3.1. Sắp xếp các phần tử của danh sách

Xây dựng thủ tục sắp xếp các phần tử có của một danh sách bằng phương pháp chèn như sau :

```

ins(X, [ ], [ X ]).
ins(X, [H|T], [ X,H|T ]) :-
    X @=< H.
ins(X, [ H|T ], [ H|L ]) :-
    X @> H, ins( X, T, L ).
?- ins(8, [ 1, 2, 3, 4, 5 ], L).
L = [1, 2, 3, 4, 5, 8]
Yes
?- ins(1, L, [ 1, 2, 3, 4, 5 ]).
L = [2, 3, 4, 5]
Yes
ins_sort([ ], [ ]).
ins_sort([H|T], L) :-
    ins_sort(T, L1),
    ins(H, L1, L).
?- ins_sort([3, 2, 6, 4, 7, 1], L).
L = [1, 2, 3, 4, 6, 7]
Yes

```

#### III.3.2. Tính độ dài của một danh sách

Xây dựng thủ tục tính độ dài hay đếm số lượng các phần tử có mặt trong một danh sách đã cho như sau :

```
length( L, N ).
```

Xảy ra hai trường hợp :

1. Nếu danh sách rỗng, thì độ dài  $N = 0$ .

2. Nếu danh sách khác rỗng, thì nó được tạo thành từ danh sách có dạng :  
 [ head | queue ]  
 và có độ dài bằng 1 cộng với độ dài của queue.

Ta có chương trình Prolog như sau :

```
length( [], 0 ).
length( [ _ | Queue ], N ) :-
    length(Queue, N1 ),
    N is 1 + N1.
```

Kết quả chạy Prolog như sau :

```
?- length( [ a, b, c, d, e ], N ).
N = 5
Yes

?- length( [ a, [ b, c ], d, e ], N ).
N = 4
Yes
```

Ta thấy rằng trong mệnh đề thứ hai, hai đích của phần thân là không thể hoán đổi cho nhau, vì rằng N1 phải được ràng buộc trước khi thực hiện đích :

N is 1 + N1

Chẳng hạn, nếu gọi trace, quá trình thực hiện length( [ 1, 2, 3 ], N ) như sau :

```
(0) gọi  length([1, 2, 3], N) ->
(1) gọi  length([2, 3], N')  ->
(2) gọi  length([3], N'')    ->
(3) gọi  length([ ], N''')    -> N''' = 0
(4) gọi  N'' is 1 + 0          -> N'' = 1
(5) gọi  N' is 1 + 1           -> N' = 2
(6) gọi  N is 1 + 2            -> N = 3
```

Với is, ta đã đưa vào một quan hệ nhạy cảm với thứ tự thực hiện các đích, và do vậy không thể bỏ qua yếu tố thủ tục trong chương trình.

Điều gì sẽ xảy ra nếu ta không sử dụng is trong chương trình. Chẳng hạn :

```
length1( [], 0 ).
```



```
length1( [ _ | Queue ], N ) :-
    length1( Queue, N1 ),
    N = 1 + N1.
```

Lúc này, nếu gọi :

```
?- length1( [ a, [ b, c ], d, e ], N ).
```

Prolog trả lời :

```
N = 1 + (1 + (1 + (1 + 0)))
Yes
```

Phép cộng do không được khởi động một cách tường minh nên sẽ không bao giờ được thực hiện. Tuy nhiên, ta có thể hoán đổi hai đích của mệnh đề thứ hai trong length1 :

```
length1( [ ], 0 ).
length1( [ _ | Queue ], N ) :-
    N = 1 + N1,
    length1( Queue, N1 ).
```

Kết quả chạy chương trình sau khi hoán đổi vẫn y hệt như cũ. Bây giờ, ta lại có thể rút gọn mệnh đề về chỉ còn một đích :

```
length1( [ ], 0 ).
length2( [ _ | Queue ], 1 + N ) :-
    length2( Queue, N ).
```

Kết quả chạy chương trình lần này vẫn y hệt như cũ. Prolog không đưa ra trả lời như mong muốn, mà là :

```
?- length1([ a, b, c, d], N).
N = 1+ (1+ (1+ (1+0)))
Yes
```

### III.3.3. Tạo sinh các số tự nhiên

Chương trình sau đây tạo sinh và liệt kê các số tự nhiên :

```
% Natural Numbers
```

```
nat(0).
nat(N) :- nat(M), N is M + 1.
```

Khi thực hiện các đích con trong câu hỏi :

```
?- nat(N), write(N), nl, fail.
```

các số tự nhiên được tạo sinh liên tiếp nhờ kỹ thuật quay lui. Sau khi số tự nhiên đầu tiên  $\text{nat}(N)$  được in ra nhờ  $\text{write}(N)$ , hằng fail bắt buộc thực hiện quay lui. Khi đó, luật thứ hai được vận dụng để tạo sinh số tự nhiên tiếp theo và cứ thế tiếp tục cho đến khi NSD quyết định dừng chương trình ( $^C$ ).

## Tóm tắt chương 4

- Danh sách là một cấu trúc hoặc rỗng, hoặc gồm hai phần : phần đầu là một phần tử và phần còn lại là một danh sách.
- Prolog quản lý các danh sách theo cấu trúc cây nhị phân. Prolog cho phép sử dụng nhiều cách khác nhau để biểu diễn danh sách.  
`[ Object1, Object2, ... ]`  
 hoặc `[ Head | Tail ]`  
 hoặc `[ Object1, Object2, ... | Others ]`  
 Với Tail và Others là các danh sách.
- Các thao tác cơ bản trên danh sách có thể lập trình được là : kiểm tra một phần tử có thuộc về một danh sách hay không, phép ghép hai danh sách, bổ sung hoặc loại bỏ một phần tử ở đầu hoặc cuối danh sách, trích ra một danh sách con...

## Bài tập chương 4

1. Viết một thủ tục sử dụng `append` để xóa ba phần tử cuối cùng của danh sách `L`, tạo ra danh sách `L1`. Hướng dẫn : `L` là phép ghép của `L1` với một danh sách của ba phần tử (đã bị xóa khỏi `L`).
2. Viết một dãy các đích để xóa ba phần tử đầu tiên và ba phần tử cuối cùng của một danh sách `L`, để trả về danh sách `L2`.
3. Định nghĩa quan hệ :  
`last_element( Object, List )`  
 sao cho `Object` phải là phần tử cuối cùng của danh sách `List`. Hãy viết thành hai mệnh đề, trong đó có một mệnh đề sử dụng `append`, mệnh đề kia không sử dụng `append`.
4. Định nghĩa hai vị từ :

`even_length( List )` và `odd_length( List )`

được thỏa mãn khi số các phần tử của danh sách List là chẵn hay lẻ tương ứng. Ví dụ danh sách :

`[ a, b, c, d ]` có độ dài chẵn,

`[ a, b, c ]` có độ dài lẻ.

5. Cho biết kết quả Prolog trả lời các câu hỏi sau :

?- `[1,2,3] = [1|X]`.

?- `[1,2,3] = [1,2|X]`.

?- `[1 | [2,3]] = [1,2,X]`.

?- `[1 | [2,3,4]] = [1,2,X]`.

?- `[1 | [2,3,4]] = [1,2|X]`.

?- `b(o,n,j,o,u,r) =.. L`.

?- `bon(Y) =.. [X,jour]`.

?- `X(Y) =.. [bon,jour]`.

6. Viết chương trình Prolog kiểm tra một danh sách có phải là một tập hợp con của một danh sách khác không ? Chương trình hoạt động như sau :

?- `subset2([4,3],[2,3,5,4])`.

Yes

7. Viết chương trình Prolog để lấy ra các phần tử từ một danh sách. Chương trình cũng có thể chèn các phần tử vào một danh sách hoạt động như sau :

?- `takeout(3,[1,2,3],[1,2])`.

Yes

?- `takeout(X,[1,2,3],L)`.

`X = 1`

`L = [2, 3] ;`

`X = 2`

`L = [1, 3] ;`

`X = 3`

`L = [1, 2] ;`

No

?- `takeout(4,L,[1,2,3])`.

4

`L = [4, 1, 2, 3] ;`

`L = [1, 4, 2, 3] ;`

`L = [1, 2, 4, 3] ;`

L = [1, 2, 3, 4] ;  
No

8. Viết vị từ Prolog `getEltFromList(L,N,E)` cho phép lấy ra phần tử thứ N trong một danh sách. Thất bại nếu danh sách không có đủ N phần tử. Chương trình hoạt động như sau :

?- `getEltFromList([a,b,c],0,X)`.

No

?- `getEltFromList([a,b,c],2,X)`.

X = b

?- `getEltFromList([a,b,c],4,X)`.

No

9. Viết chương trình Prolog tìm phần tử lớn nhất và phần tử nhỏ nhất trong một danh sách các số. Chương trình hoạt động như sau :

?- `maxmin([3,1,5,2,7,3],Max,Min)`.

Max = 7

Min = 1

Yes

?- `maxmin([2],Max,Min)`.

Max = 2

Min = 2

Yes

10. Viết chương trình Prolog chuyển một danh sách phức hợp, là danh sách mà mỗi phần tử có thể là một danh sách con chứa các danh sách con phức hợp khác, thành một danh sách phẳng là danh sách chỉ chứa các phần tử trong tất cả các danh sách con có thể, giữ nguyên thứ tự lúc đầu. Chương trình hoạt động như sau :

`flatten([[1,2,3],[4,5,6]], Flatlist)`.

Flatlist = [1,2,3,4,5,6]

Yes

`flatten([[1,[hallo,[aloha]]],2,[],3],[4,[],5,6]], Flatlist)`

Flatlist = [1, hallo, aloha, 2, 3, 4, 5, 6]

Yes

11. Viết các chương trình Prolog thực hiện các vị từ xử lý tập hợp cho ở phần lý thuyết (mục II).

12. Sử dụng vị từ forall để viết chương trình Prolog kiểm tra hai danh sách có rời nhau (disjoint) không ? Chương trình hoạt động như sau :
- ?- disjoint([a,b,c],[d,g,f,h]).  
Yes  
?- disjoint([a,b,c],[f,a]).  
No
13. Vị từ forall(Cond, Action) thực hiện kiểm tra sự so khớp tương ứng giữa Cond, thường kết hợp với vị từ member, và Action. Ví dụ dưới đây kiểm tra việc thực hiện các phép toán số học trong danh sách L là đúng đắn.
- ?- forall(member(Result = Formula, [2 = 1 + 1, 4 = 2 \* 2]), Result == Formula).  
Result = \_G615  
Formula = \_G616  
Yes
14. Sử dụng vị từ forall để viết chương trình Prolog kiểm tra một danh sách có là một tập hợp con của một danh sách khác hay không ? Chương trình hoạt động như sau :
- ?- subset3([a,b,c],[c,d,a,b,f]).  
Yes  
?- subset3([a,b,q,c],[d,a,c,b,f])  
No
15. Sử dụng vị từ append ghép hai danh sách để viết các chương trình Prolog thực hiện các việc sau :
- prefixe(L1, L2)     danh sách L1 đứng trước (prefixe list) danh sách L2.  
suffixe(L1, L2)     danh sách L1 đứng sau (suffixe list) danh sách L2.  
isin(L1, L2)        các phần tử của danh sách L1 có mặt trong danh sách L2.
16. Sử dụng phương pháp Quicksort viết chương trình Prolog sắp xếp nhanh một danh sách các số đã cho theo thứ tự tăng dần.
17. Đọc hiểu chương trình sau đây rồi dựng lại thuật toán :
- ```
/* Missionarys & Cannibals */
/* Tránh vòng lặp */
INotExist(_, []).
INotExist(X, [T|Q]) :-
    X \== T, INotExist(X, Q).
```

```

/* Kiểm tra tính hợp lý của trạng thái */
valid(MG,CG,MD,CD) :-
    MG>=0, CG>=0, MD>=0, CD>=0, MG=0, MD>=CD.
valid(MG,CG,MD,CD) :-
    MG>=0, CG>=0, MD>=0, CD>=0, MG>=CG, MD=0.
valid(MG,CG,MD,CD) :-
    MG>=0, CG>=0, MD>=0, CD>=0, MG>=CG, MD>=CD.

/* Xây dựng cung và kiểm tra */
sail(1,0). sail(0,1). sail(1,1). sail(2,0). sail(0,2).
arc([left,MGi,CGi,MDi,CDi],[droite,MGf,CGf,MDf,CDf]) :-
    sail(Mis,Can),
    MGf is MGi-Mis, MDf is MDi+Mis,
    CGf is CGi-Can, CDf is CDi+Can,
    valid(MGf,CGf,MDf,CDf).
arc([right,MGi,CGi,MDi,CDi],[left,MGf,CGf,MDf,CDf]) :-
    sail(Mis,Can),
    MGf is MGi+Mis, MDf is MDi-Mis,
    CGf is CGi+Can, CDf is CDi-Can,
    valid(MGf,CGf,MDf,CDf).

/* Phép đệ quy */
cross(A,A,[A],Non).
cross(X,Y,Ch,Non) :-
    arc(X,A), INotExist(A,Non),
    cross(A,Y,ChAY,[A|Non]), Ch=[X|ChAY].

/* Đi qua */
traverse(X,Y,Ch) :-
    cross(X,Y,Ch,[X]).

```

## CHƯƠNG 5

---

# Kỹ thuật lập trình Prolog

## I. Nhát cắt

### I.1. Khái niệm nhát cắt

Như đã thấy, một trình Prolog được thực hiện nhờ các mệnh đề và các đích. Sau đây ta sẽ xét một kỹ thuật khác của Prolog cho phép ngăn chặn sự quay lui là *nhát cắt* (cut).

Prolog tự động quay lui khi cần tìm một tìm kiếm một mệnh đề khác để thoả mãn đích. Điều này rất có ích đối với người lập trình khi cần sử dụng nhiều phương án giải quyết vấn đề. Tuy nhiên, nếu không kiểm soát tốt quá trình này, việc quay lui sẽ trở nên kém hiệu quả. Vì vậy, Prolog sử dụng kỹ thuật nhát cắt kiểm soát quay lui, hay cấm quay lui, để khắc phục khiếm khuyết này.

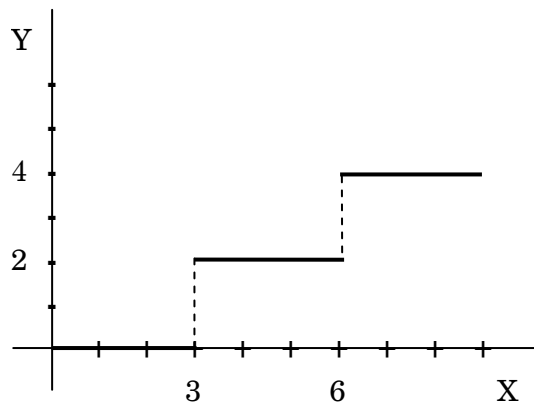
Trong ví dụ sau đây, một chương trình Prolog sử dụng kỹ thuật quay lui kém hiệu quả. Ta cần xác định các vị trí mà từ đó chương trình bắt đầu quá trình quay lui. Ta xét hàm bậc thang

Ta có ba quy tắc xác định quan hệ giữa hai trục X và Y như sau :

1. Nếu  $X < 3$  thì  $Y = 0$
2. Nếu  $X \leq 3$  và  $X < 6$  thì  $Y = 2$
3. Nếu  $X \leq 6$  thì  $Y = 4$

Ta viết thành quan hệ nhị phân  $f(X, Y)$  trong Prolog như sau :

|                               |          |
|-------------------------------|----------|
| $f(X, 0) :- X < 3.$           | % luật 1 |
| $f(X, 2) :- 3 \leq X, X < 6.$ | % luật 2 |
| $f(X, 4) :- 6 \leq X.$        | % luật 3 |



Hình I.1. Hàm bậc thang có hai bậc.

Khi chạy chương trình, giả sử rằng biến  $X$  của hàm  $f(X, Y)$  đã được nhận một giá trị số để có thể thực hiện phép so sánh trong thân hàm. Từ đây, xảy ra hai khả năng sử dụng kỹ thuật nhát cắt như sau :

## I.2. Kỹ thuật sử dụng nhát cắt

### I.2.1. Kỹ thuật nhát cắt thứ nhất

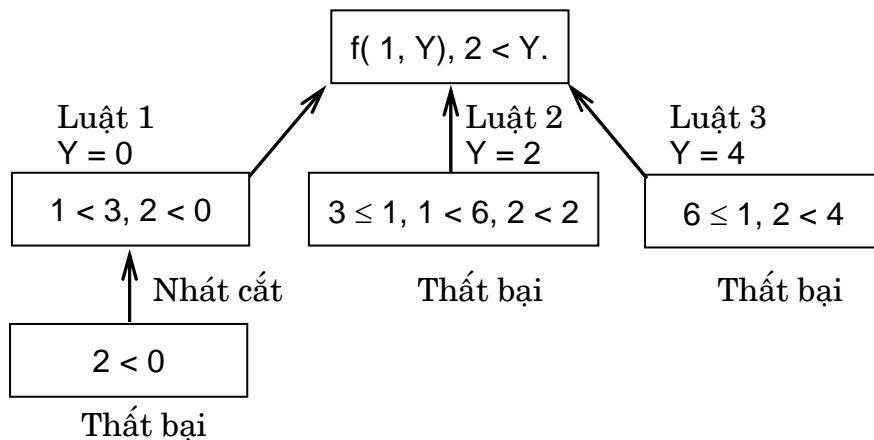
Giả sử ta đặt ra câu hỏi :

?-  $f(1, Y), 2 < Y$ .

Lúc này,  $Y$  nhận giá trị 0, đích thứ hai trở thành :

$2 < 0$

và gây ra kết quả No (thất bại) cho cả danh sách các đích còn lại, vì Prolog còn tiếp tục tiến hành thêm hai quá trình quay lui vô ích khác :



Hình I.2. Tại vị trí «Nhát cắt», các luật 2 và 3 đã biết trước thất bại.



Cả ba luật định nghĩa quan hệ  $f$  có tính chất loại trừ lẫn nhau, chỉ có duy nhất một trong chúng là có thể thành công. Người lập trình biết điều này nhưng Prolog lại không biết, cho nên cứ tiếp tục áp dụng tất cả các luật mặc dù đi đến thất bại. Trong ví dụ trên, luật 1 được áp dụng tại vị trí «Nhất cắt» và gây ra thất bại. Để tránh sự quay lui không cần thiết bắt đầu từ vị trí này, chúng ta cần báo cho Prolog biết một cách tường minh, bằng cách sử dụng một nhất cắt, ký hiệu bởi một dấu chấm than «!» thực chất là một *đích giả* (pseudo goal) được chèn vào giữa các đích thật khác. Chương trình hàm bậc thang được viết lại như sau :

```
f( X, 0 ) :- X < 3, !.      % luật 1
f( X, 2 ) :-
    3 =< X, X < 6, !. % luật 2
f( X, 4 ) :-
    6 =< X.                % luật 3
```

Nhất cắt ! sẽ cấm mọi quá trình quay lui từ vị trí xuất hiện của nó trong chương trình. Nếu bây giờ ta yêu cầu thực hiện đích :

?- f( 1, Y ), 2 < Y.

Prolog chỉ thực hiện nhánh trái nhất ứng với luật 1 trong hình trên, trả về kết quả thất bại vì xảy ra  $2 < 0$  mà không tiếp tục quay lui thực hiện các nhánh tương ứng với luật 2 và 3, do đã gặp nhất cắt !. Chương trình mới sử dụng nhất cắt chạy hiệu quả hơn chương trình cũ. Khi xảy ra thất bại, Prolog sẽ nhanh chóng dừng, mà không mất thời gian để thực hiện những việc vô ích khác. Sử dụng nhất cắt trong một chương trình làm thay đổi nghĩa thủ tục nhưng không làm thay đổi nghĩa khai báo. Tuy nhiên sau đây ta sẽ thấy rằng nhất cắt có thể làm mất đi nghĩa khai báo.

### 1.2.2. Kỹ thuật nhất cắt thứ hai

Giả sử bây giờ ta gọi thực hiện đích :

?- f( 7, Y ).

Y=4

Yes

Quá trình thực hiện được mô tả như sau : trước khi nhận được kết quả, về nguyên tắc, Prolog phải sử dụng cả ba luật để có quá trình xoá đích.

Thủ luật 1  $7 < 3$  thất bại, quay lui, thủ luật 2 (nhát cắt chưa được sử dụng).

Thủ luật 2  $3 \leq 7$  thoả mãn, nhưng  $7 < 6$  thất bại, quay lui, thủ luật 3 (nhát cắt chưa được sử dụng).

Thủ luật 3  $6 \leq 7$  thoả mãn.

Đến đây, ta lại thấy xuất hiện chương trình thực hiện kém hiệu quả. Khi xảy ra đích  $X < 3$  (nghĩa là  $7 < 3$ ) thất bại, đích tiếp theo  $3 \leq X$  ( $3 \leq 7$ ) thoả mãn, Prolog tiếp tục kiểm tra đích trong luật 3. Nhưng ta biết rằng nếu một đích thứ nhất thất bại, thì đích thứ hai bắt buộc phải được thoả mãn vì nó là phủ định của đích thứ nhất. Việc kiểm tra lần nữa sẽ trở nên dư thừa vì đích tương ứng với nó có thể bị xoá. Như vậy việc kiểm tra đích  $6 \leq X$  của luật 3 là không cần thiết. Với nhận xét này, ta có thể viết lại chương trình hàm bậc thang tiết kiệm hơn như sau :

*Nếu  $X < 3$  thì  $Y = 0$ ,*

*Nếu không, nếu  $X < 6$  thì  $Y = 2$ ,*

*Nếu không  $Y = 4$ .*

Bằng cách loại khỏi chương trình những điều kiện mà biết chắc chắn sẽ đúng, ta nhận được chương trình mới như sau :

$f(X, 0) :- X < 3, !.$

$f(X, 2) :- X < 6, !.$

$f(X, 4).$

Chương trình này cho kết quả tương tự hai chương trình trước đây nhưng thực hiện nhanh hơn do đã loại bỏ hoàn toàn những quay lui không cần thiết.

?-  $f(1, Y).$

$Y = 0$

Yes

?-  $f(5, Y).$

$Y = 2$

Yes

?-  $f(7, Y).$

$Y = 4$

Yes

Nhưng vấn đề gì sẽ xảy ra nếu bây giờ ta lại loại bỏ hết các nhát cắt ra khỏi chương trình ? Chẳng hạn :

$f(X, 0) :- X < 3.$

$f(X, 2) :- X < 6.$

$f(X, 4).$

Với lời gọi :

?-  $f(1, Y).$

$Y = 0;$

$Y = 2;$

$Y = 4;$

No

Prolog đưa ra nhiều câu trả lời nhưng không đúng. Như vậy, việc sử dụng nhất cắt đã làm thay đổi đồng thời nghĩa thủ tục và nghĩa khai báo. Kỹ thuật nhất cắt có thể được mô tả như sau :

Ta gọi «đích cha» là đích tương ứng với phần đầu của mệnh đề chứa nhất cắt. Ngay khi gặp nhất cắt, Prolog xem rằng một đích đã được thoả mãn một cách tự động, và giới hạn sự lựa chọn các mệnh đề trong phạm vi giữa lời gọi đích cha và thời điểm thực hiện nhất cắt. Tất cả các mệnh đề tương ứng với các đích con chưa được kiểm tra so khớp giữa đích cha và nhất cắt đều được bỏ qua.

Để minh hoạ, ta xét mệnh đề có dạng :

$H :- G1, G2, \dots Gm, !, \dots, Bn.$

Giả sử rằng mệnh đề này được khởi động bởi một đích  $G$  hợp nhất được với  $H$ , khi đó,  $G$  là đích cha. Cho đến khi gặp nhất cắt, Prolog đã tìm được các lời giải cho các đích con  $G1, G2, \dots Gm$ .

Ngay sau khi thực hiện nhất cắt, các đích con  $G1, G2, \dots Gm$  bị «vô hiệu hoá», kể cả các mệnh đề tương ứng với các đích con này cũng bị bỏ qua. Hơn nữa, do  $G$  hợp nhất với  $H$  nên Prolog không tiếp tục tìm kiếm để so khớp  $H$  với đầu (head) của các mệnh đề khác.

Chẳng hạn, áp dụng nguyên lý trên cho ví dụ sau :

$C :- P, Q, R, ! S, T, U.$

$C :- V.$

$A :- B, C, D.$

?-  $A.$

Giả sử  $A, B, C, D, P, \dots$  đều là các hạng. Tác động của nhất cắt khi thực hiện đích  $C$  như sau : quá trình quay lui xảy ra bên trong danh sách các đích

P, Q, R, nhưng ngay khi thực hiện nhất cắt, mọi con đường dẫn đến các mệnh đề trong danh sách P, Q, R đều bị bỏ qua. Mệnh đề C thứ hai :

$C :- V.$

cũng bị bỏ qua. Tuy nhiên, việc quay lui vẫn có thể xảy ra bên trong danh sách các đích S, T, U. Đích cha của mệnh đề chứa nhất cắt là C ở trong mệnh đề :

$A :- B, C, D.$

Như vậy, nhất cắt chỉ tác động đối với mệnh đề C, mà không tác động đối với A. Việc quay lui tự động trong danh sách các đích B, C, D vẫn được thực hiện, độc lập với nhất cắt hiện diện trong C.

### 1.2.3. Ví dụ sử dụng kỹ thuật nhất cắt

#### 1. Tìm số max

Xây dựng chương trình tìm số lớn nhất trong hai số có dạng :

$\text{max}(X, Y, \text{MaX})$

trong đó,  $\text{Max} = X$  nếu X lớn hơn hoặc bằng Y, và  $\text{Max} = Y$  nếu X nhỏ hơn hoặc bằng Y. Ta xây dựng hai quan hệ như sau :

$\text{max}(X, Y, X) :- X \geq Y.$

$\text{max}(X, Y, Y) :- X < Y.$

Hai quan hệ trên loại trừ lẫn nhau. Nếu quan hệ thứ nhất thoả mãn, thì quan hệ thứ 2 chỉ có thể thất bại và ngược lại. Áp dụng dạng điều kiện quen thuộc «nếu-thì-nếu không thì» để làm gọn chương trình lại như sau :

*Nếu  $X \geq Y$  thì  $\text{Max} = X$ ,*

*Nếu không thì  $\text{Max} = Y$ .*

Sử dụng kỹ thuật nhất cắt, chương trình được viết lại như sau :

$\text{max}(X, Y, X) :- X \geq Y, !.$

$\text{max}(X, Y, Y).$

## 2. Kiểm tra một phần tử có thuộc danh sách đã cho không

Ta đã xây dựng quan hệ :

`membre( X, L).`

để kiểm tra phần tử  $X$  có nằm trong danh sách  $L$  hay không. Chương trình như sau :

`membre( X, [X | L]).`

`membre( X, [X | L]) :- membre( X, L).`

Tuy nhiên, chương trình này hoạt động một cách «không đơn định». Nếu  $X$  xuất hiện nhiều lần trong danh sách, thì bất kỳ phần tử nào bằng  $X$  cũng được tìm thấy. Bây giờ ta chuyển `membre` thành một quan hệ đơn định chỉ tác động đối với phần tử  $X$  đầu tiên. Việc thay đổi rất đơn giản như sau : chỉ việc cấm quay lui ngay khi  $X$  được tìm thấy, nghĩa là khi mệnh đề đầu tiên được thoả mãn :

`membre( X, [ X | L ]) :- !.`

`membre( X, [ X | L ]) :- membre( X, L).`

Khi đó, trong ví dụ sau, Prolog chỉ đưa ra một lời giải :

?- `membre( X, [a, a, b, c]).`

$X = a$  ;

No

## 3. Thêm một phần tử vào danh sách mà không bị trùng lặp

Thông thường, khi muốn thêm một phần tử mới, chẳng hạn  $X$ , vào danh sách  $L$ , người ta muốn trước đó,  $L$  không chứa phần tử này. Giả sử quan hệ cần xây dựng :

`ajoute( X, L, L1)`

có  $X$  là phần tử mới cần thêm vào danh sách  $L$ ,  $L1$  là kết quả có chứa đúng một  $X$ . Ta lập luận như sau :

*Nếu  $X$  thuộc danh sách  $L$ , thì  $L1 = L$ ,*

*Nếu không,  $L1$  là  $L$  đã được thêm  $X$  vào.*

Cách đơn giản nhất là chèn phần tử  $X$  vào ngay đầu danh sách sao cho nó là phần tử đầu (head) của  $L1$ . Ta có chương trình như sau :

`ajoute( X, L, L) :- membre( X, L), !.`

ajoute( X, L, [ X | L ] ).

Sau đây là các vận dụng chương trình :

?- ajoute( a, [ b, c ], L).

L = [ a, b, c ]

?- ajoute( X, [ b, c ], L).

L = [ b, c ]

X = b

?- ajoute( a, [ b, c, X ], L).

X = \_G333

L = [a, b, c, \_G333]

?- ajoute( a, [ a, b, c ], L).

L = [a, b, c];

Trong ví dụ này, nhờ sử dụng kỹ thuật nhất cắt, người lập trình dễ dàng thêm một phần tử mới vào danh sách mà không làm trùng lặp phần tử đó. Nếu không sử dụng kỹ thuật nhất cắt, việc thêm một phần tử mới vào một danh sách có thể làm trùng lặp phần tử.

Như vậy, kỹ thuật nhất cắt không những làm tối ưu hiệu quả lập trình, mà còn rất cần thiết để đặc tả đúng đắn mối quan hệ giữa các đối tượng.

#### **4. Sử dụng nhất cắt để phân loại dữ liệu**

Giả sử ta cần quản lý một CSDL chứa kết quả các trận đấu của các hội viên một câu lạc bộ quần vợt. Các trận đấu không được sắp xếp một cách có hệ thống, mà mỗi hội viên có thể đấu với bất cứ ai. Kết quả các trận đấu được biểu diễn bởi các sự kiện như sau :

bat( tom, jim).

bat( ann, tom).

bat( pat, jim).

Ta cần định nghĩa quan hệ :

classe(Player, Category).

để phân thứ hạng cho mỗi người chơi quần vợt trong ba hạng như sau :

champion            người luôn thắng trong tất cả các trận đấu

combative          người có cả bàn thắng và có cả bàn thua

dilettante          người luôn thua trong tất cả các trận đấu

Từ kết quả những trận đấu đã có được cho trong các sự kiện, ta thấy Ann và Pat được xếp hạng quán quân (champion), Tom được xếp hạng trung bình (combative), còn Jim thì được xếp hạng yếu kém (dilettante). Ta có thể dễ dàng xây dựng các luật xếp hạng như sau :

*X được xếp hạng trung bình nếu  
tồn tại Y sao cho X thắng Y, và  
tồn tại Z sao cho Z thắng X.*

*X được xếp hạng quán quân nếu  
X thắng Y, và  
X không bị thua bất kỳ đối thủ nào.*

Luật xếp hạng quán quân có chứa phép phủ định (not) mà cho đến lúc này, ta chưa tìm hiểu cách biểu diễn như thế nào trong Prolog. Luật xếp hạng yếu kém cũng xây dựng tương tự luật xếp hạng quán quân. Ta có thể sử dụng sơ đồ *if-then-else* để xử lý đồng thời hai tình huống như sau :

*Nếu X thắng và X bị thua khi đấu với bất kỳ ai  
thì X được xếp hạng trung bình  
nếu không, nếu X thắng bất kỳ ai  
thì X được xếp hạng quán quân  
nếu không, nếu X luôn bị thua  
thì X được xếp hạng yếu kém.*

Từ sơ đồ trên ta có thể chuyển sang Prolog sử dụng kỹ thuật nhất cắt để xử lý khả năng loại trừ nhau giữa ba thứ hạng.

```
classe( X, combative) :-
    bat( X, _ ),
    bat( _, X ), !.

classe( X, champion) :-
    bat( X, _ ), !.

classe( X, dilettante) :-
    bat( _, X ).
```

Chú ý rằng không nhất thiết phải sử dụng nhất cắt trong mệnh đề champion vì bản chất của ba thứ hạng.

### I.3. Phép phủ định

#### I.3.1. Phủ định bởi thất bại

Trong Prolog, ta có thể nói được câu : «Marie thích tất cả loài động vật trừ loài rắn» hay không ?

Đối với vế thứ nhất, ta có thể dễ dàng dịch ra thành : *Dù X là gì, Marie thích X nếu X là loài động vật :*

```
enjoy( marie, X ) :-  
    animal( X ).
```

Tuy nhiên cần loại trừ loài rắn. Lúc này ta cần dịch ra như sau :

*Nếu X là loài rắn, thì «Marie thích X» là sai,  
Nếu không, nếu X là loài động vật thì Marie thích X.*

Những gì không đúng thì có thể sử dụng đích đặc biệt fail (thất bại) để luôn luôn sai, và cũng làm cho đích cha thất bại. Chương trình được viết lại như sau :

```
enjoy( marie, X ) :-  
    serpent( X ), !, fail.  
  
enjoy( marie, X ) :-  
    animal( X ).
```

Luật thứ nhất xử lý tình huống Marie không thích loài rắn : nếu X là loài rắn, thì nhất cắt sẽ ngăn sự quay lui (và do đó, luật thứ hai không được thực hiện), và đích fail sẽ gây ra thất bại. Ta có thể sử dụng dấu ; để viết cô đọng hai luật thành một luật như sau :

```
enjoy( marie, X ) :-  
    serpent( X ), !, fail;  
    animal( X ).
```

Một cách tương tự, ta định nghĩa quan hệ khác nhau :

```
différent( X, Y )
```

thỏa mãn nếu X và Y là khác nhau. Do sự khác nhau có thể được diễn giải theo nhiều cách nên ta cần chỉ rõ như sau :

- X và Y không phải là các trục hằng (literal) đồng nhất,
- X và Y không thể khớp với nhau,
- Các giá trị của các biểu thức số học X và Y không thể bằng nhau.



Ta nói rằng X và Y khác nhau do chúng không thể khớp được với nhau :

*Nếu X và Y là đồng nhất, thì différent( X, Y ) thất bại,*

*Nếu không, différent( X, Y ) thành công.*

Ta sử dụng nhất cắt và đích fail để viết quan hệ này thành hai luật :

différent( X, X ) :- !, fail.

différent( X, Y ).

Hoặc viết lại thành một luật như sau :

différent( X, Y ) :-

    X = Y, !, fail;

    true.

Chú ý rằng đích true (đúng) luôn luôn thành công.

Từ đây, ta có thể định nghĩa vị từ not(Goal) cho phép kiểm tra đích không thoả mãn như sau :

*Nếu Goal thoả mãn, thì not(Goal) thất bại,*

*Nếu không, not(Goal) thành công.*

Chương trình Prolog :

not( P ) :-

    P, !, fail;

    true.

Hầu hết các phiên bản Prolog hiện nay đều có vị từ not

not(2 = 3).

Yes

?- not(2 = 2).

No

Sử dụng vị từ not, ta có thể định nghĩa lại các quan hệ enjoy, différent và classe như sau :

enjoy( marie, X ) :-

    animal( X ),

    not (serpent( X )).

différent( X, Y ) :-

    not( X = Y ).

classe( X, combatif ) :-

    bat( X, \_ ),

    bat( \_ , X ).

```

classe( X, champion) :-
    bat( X _ ),
    not bat( _ , X ).

classe( X, dilettante) :-
    bat( _ , X ),
    not bat( X, _ ).

```

### 1.3.2. Sử dụng kỹ thuật nhất cắt và phủ định

Ưu điểm của kỹ thuật nhất cắt có thể tóm tắt như sau :

1. Nâng cao tính hiệu quả của một chương trình nhờ nguyên tắc thông báo một cách tường minh cho Prolog tránh không đi theo những con đường dẫn đến thất bại.
2. Kỹ thuật nhất cắt cho phép xây dựng những luật có tính chất loại trừ nhau có dạng :

*Nếu điều kiện P xảy ra thì kết luận là Q,  
Nếu không, thì kết luận là R.*

Tuy nhiên sử dụng nhất cắt có thể làm mất sự tương ứng giữa nghĩa khai báo và nghĩa thủ tục của một chương trình. Nếu trong chương trình không xuất hiện nhất cắt, thì việc thay đổi thứ tự các mệnh đề và các đích chỉ làm ảnh hưởng đến hiệu quả chạy chương trình mà không làm thay đổi nghĩa khai báo. Còn khi có mặt nhất cắt trong một chương trình, thì lại xảy ra vấn đề, lúc này có thể có thể nhiều kết quả khác nhau. Ví dụ :

p :- a, b.  
p :- c.

Xét về mặt nghĩa khai báo, chương trình trên có nghĩa : p đúng nếu và chỉ nếu cả a và b đều đúng, hoặc c đúng. Từ đó ta xây dựng biểu thức logic như sau :

$$p \Leftrightarrow (a \wedge b) \vee c$$

Nghĩa khai báo không còn đúng nữa nếu ta thay đổi mệnh đề thứ nhất bằng cách thêm vào một nhất cắt :

p :- a, !, b.  
p :- c.

Biểu thức logic tương ứng như sau :

$$p \Leftrightarrow (a \wedge b) \vee (\sim a \wedge c)$$

Nếu ta đảo thứ tự hai mệnh đề :

$$p :- c.$$

$$p :- a, !, b.$$

thì ta lại có cùng nghĩa như ban đầu :

$$p \Leftrightarrow c \vee (a \wedge b)$$

Người ta phải thận trọng khi sử dụng kỹ thuật nhất cắt do nhất cắt làm thay đổi nghĩa thủ tục và làm tăng nguy cơ xảy ra sai sót trong chương trình. Như đã xét trong các ví dụ trước đây, việc loại bỏ nhất cắt có thể làm thay đổi nghĩa khai báo của một chương trình. Tuy nhiên trong một số trường hợp, nhất cắt không ảnh hưởng đến nghĩa khai báo. Người ta gọi những nhất cắt không làm thay đổi ngữ nghĩa của chương trình là *nhất cắt xanh* (green cuts). Đúng trên quan điểm lập trình dễ đọc và dễ hiểu (readability), các nhất cắt xanh là an toàn và người ta thường hay sử dụng chúng. Thậm chí, người ta có thể bỏ qua sự có mặt của chúng khi đọc chương trình. Người ta nói nhất cắt xanh làm *rõ ràng* (explicit) *tính tiền định* (determinism) vốn không rõ ràng (implicit). Thông thường nhất cắt xanh được đặt ngay sau phép kiểm tra tiền định.

Ví dụ sử dụng nhất cắt xanh tìm số min :

$$\text{minimum}(X, Y, X) :-$$

$$X \leq Y, !.$$

$$\text{minimum}(X, Y, Y) :-$$

$$X > Y, !.$$

Ví dụ sử dụng nhất cắt xanh kiểm tra kiểu của cây nhị phân các số nguyên :

$$\text{int\_bin\_tree}(\text{ab}(X, G, D)) :-$$

$$\text{integer}(X),$$

$$\text{int\_bin\_tree}(G),$$

$$\text{int\_bin\_tree}(D).$$

$$\text{int\_bin\_tree}(X) :-$$

$$\text{integer}(X).$$

Trong các trường hợp khác, các nhất cắt ảnh hưởng đến nghĩa khai báo được gọi là *nhất cắt đỏ* (red cuts). Sự có mặt của các nhất cắt đỏ thường làm cho chương trình trở nên khó đọc, khó hiểu. Để sử dụng được chúng, NSD

phải hết sức chú ý. Ví dụ sử dụng nhất cắt để tìm số min thay đổi ngữ nghĩa :

```
minimum_cut( X, Y, X ) :-
    X <= Y, !.
minimum_cut( X, Y, Y ).
```

Trong một số trường hợp, một câu hỏi có thể không liên quan đến ngữ nghĩa của chương trình. Ví dụ vị từ kiểm tra một phần tử có thuộc danh sách không :

```
member_cut(X, [ X | _ ] ) :- !.
member_cut(X, [ _ | L ] ) :- member_cut(X, L ).
```

Với câu hỏi `member_cut(X, [ 1, 2, 3 ] )` sẽ không cho kết quả `X = 2`.

?- `member_cut(X, [ 1, 2, 3 ] )`.

`X = 1 ;`

No

Thông thường, đích fail được dùng cặp đôi với nhất cắt (cut-fail). Người ta thường định nghĩa phép phủ định một đích (not) bằng cách gây ra sự thất bại của đích này, thực chất là cách sử dụng nhất cắt có hạn chế. Để chương trình dễ hiểu hơn, thay vì sử dụng cặp đôi cut-fail, người ta sử dụng not. Tuy nhiên, phép phủ định not cũng không phải không gây ra những phiền phức cho người dùng. Nhiều khi sử dụng not không hoàn toàn chính xác với phép phủ định trong Toán học. Chẳng hạn nếu trong chương trình có định nghĩa quan hệ man, mà ta đưa ra một câu hỏi đại loại như :

?- `not( man( marie) )`.

Khi đó, Prolog sẽ trả lời No nếu đã có định nghĩa `man( marie)`, trả lời Yes nếu chưa có định nghĩa như vậy. Tuy nhiên, khi trả lời No, không phải Prolog nói rằng «Marie không phải là một người», mà nói rằng «Không tìm thấy trong chương trình thông tin để chứng minh Marie là một người». Khi thực hiện phép not, Prolog không chứng minh trực tiếp mà tìm cách chứng minh điều ngược lại. Nếu chứng minh được, Prolog suy ra rằng đích not thành công. Cách lập luận như vậy được gọi là *giả thuyết về thế giới khép kín* (hypothesis of the enclosed world). Theo giả thuyết này, thế giới khép kín có nghĩa là những gì tồn tại (đúng) đều nằm trong chương trình hoặc được suy ra từ chương trình. Những gì không nằm trong chương trình, hoặc không thể suy ra từ chương trình, thì sẽ là không đúng (sai), hay điều phủ định là

đúng. Vì vậy, cần chú ý khi sử dụng phủ định do thông thường, người ta đã không giả thiết rằng thế giới là khép kín. Trong chương trình, do thiếu khai báo mệnh đề :

man( marie).

nên Prolog không chứng minh được rằng Marie là một người.

Sau đây là một ví dụ khác sử dụng phép phủ định not :

r( a).

q( b).

p( X ) :- not( r( X )).

Nếu đặt câu hỏi :

?- q( X ), p( X ).

thì Prolog sẽ trả lời :

X=b

Yes

Nhưng nếu đặt câu hỏi :

?- p( X ), q( X ).

thì Prolog sẽ trả lời :

No

Để hiểu được vì sao cùng một chương trình nhưng với hai cách đặt câu hỏi khác nhau lại có hai cách trả lời khác nhau, ta cần tìm hiểu cách Prolog lập luận. Trong trường hợp thứ nhất, biến X được ràng buộc giá trị là b khi thực hiện đích q( X ). Tiếp tục thực hiện đích con p( X ), nhờ ràng buộc X=b, đích not( r( X ) ) thoả mãn vì đích r( b ) không thoả mãn, Prolog trả lời Yes.

Trái lại trong trường hợp thứ hai, do Prolog thực hiện đích con p( X ) trước nên sự thất bại của not( r( X ) ), tức r( X ) thành công với ràng buộc X=a, dẫn đến câu trả lời No.

## II. Sử dụng các cấu trúc

Kiểu dữ liệu cấu trúc, danh sách, kỹ thuật so khớp, quay lui và nhất cắt là những điểm mạnh trong lập trình Prolog. Chương này sẽ tiếp tục trình bày một số ví dụ tiêu biểu về :

- Truy cập thông tin cấu trúc từ một cơ sở dữ liệu.

- Mô phỏng một ô tômat hữu hạn không đơn định và máy Turing.
- Lập kế hoạch đi du lịch
- Bài toán tám quân hậu

Đồng thời, ta cũng trình bày cách Prolog trừu tượng hoá dữ liệu.

## II.1. Truy cập thông tin cấu trúc từ một cơ sở dữ liệu

Sau đây là một ví dụ cho phép biểu diễn và thao tác các dữ liệu cấu trúc. Từ đó, ta cũng hiểu cách sử dụng Prolog như một ngôn ngữ truy vấn cơ sở dữ liệu.

Trong Prolog, một cơ sở được biểu diễn dưới dạng một tập hợp các sự kiện. Chẳng hạn, một cơ sở dữ liệu về các gia đình sẽ mô tả mỗi gia đình (family) như một mệnh đề. Mỗi gia đình sẽ gồm ba phần tử lần lượt : chồng, vợ (individual) và các con (children). Do các phần tử này thay đổi tùy theo từng gia đình, nên các con sẽ được biểu diễn bởi một danh sách để có thể nhận được một số lượng tùy ý số con. Mỗi người trong gia đình được biểu diễn bởi bốn thành phần : tên, họ, ngày tháng năm sinh và việc làm. Thành phần việc làm có thể có giá trị "thất nghiệp" (inactive), hoặc chỉ rõ tên cơ quan công tác và thu nhập theo năm.

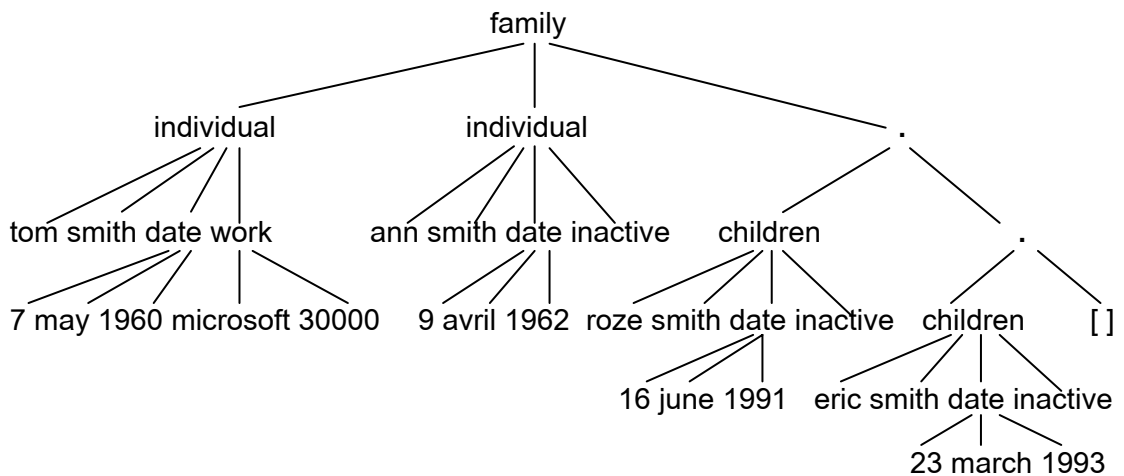
Giả sử cơ sở dữ liệu chứa mệnh đề đầu tiên như sau :

```
family(
    individual( tom, smith, date(7, may, 1960), work(microsoft, 30000) ),
    individual( ann, smith, date(9, avril, 1962), inactive),
    [ individual( roze, smith, date(16, june, 1991), inactive),
      individual( eric, smith, date(23, march, 1993), inactive) ] ).
```

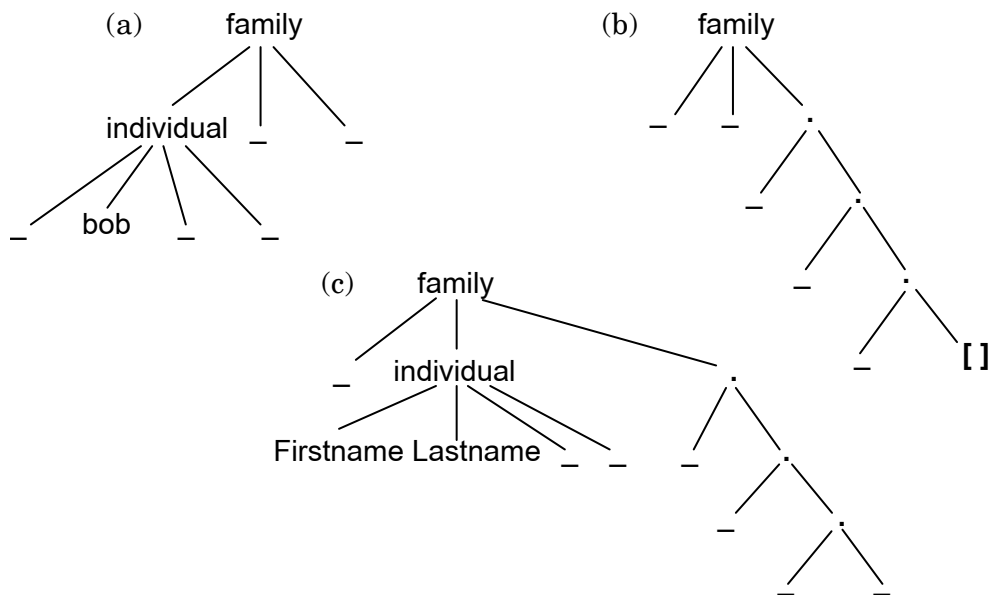
Dữ liệu về những gia đình khác tiếp tục được bổ sung dưới dạng các mệnh đề tương tự. Hình 5.1 dưới đây minh họa cách tổ chức cơ sở dữ liệu.

Prolog là một ngôn ngữ rất thích hợp cho việc khôi phục thông tin : người sử dụng có thể gọi các đối tượng mà không nhất thiết chỉ rõ tất cả các thành phần. Người sử dụng chỉ cần chỉ ra *cấu trúc* của các đối tượng mà họ quan tâm một cách tương trưng, không cần phải chỉ ra hết. Hình 1.2 minh họa những cấu trúc như vậy. Ví dụ, để biểu diễn những gia đình dòng họ Smith, trong Prolog viết :

```
family( individual( _ , smith, _ , _ ), _ , _ )
```



Hình II.1. Cấu trúc cây biểu diễn thông tin về một gia đình



Hình II.2. tính chất cấu trúc của các đối tượng Prolog cho phép biểu diễn :

(a) một gia đình Smith nào đó ; (b) những gia đình có đúng ba con ; (c) những gia đình có ít nhất ba con. Riêng trường hợp (c) còn cho phép biểu diễn tên của người vợ nhờ sự ràng buộc các biến Firstname và Lastname.

Những dấu gạch dưới dòng như đã biết là các biến nặc danh, người sử dụng không cần quan tâm đến giá trị của chúng. Một cách tương tự, những gia đình có ba con được biểu diễn bởi :

family( \_ , \_ , [ \_ , \_ , \_ ] )

Ta cũng có thể đặt câu hỏi tìm những người vợ trong những gia đình có ít nhất ba con :

?- family( \_, individual( Firstname, Lastname, \_, \_ ), [ \_, \_, \_ | \_ ] ).

Những ví dụ trên đây chỉ ra rằng ta có thể biểu diễn các đối tượng bởi cấu trúc của chúng mà không cần quan tâm đến nội dung, bằng cách bỏ qua những tham đối vô định.

Sau đây là một số mệnh đề được đưa thêm vào cơ sở dữ liệu các gia đình để có thể đặt các câu hỏi vấn tin khác nhau (có thể bổ sung thêm các gia đình mới bởi mệnh đề family) :

```
husban( X ) :-                               % X là một người chồng
    family( X , _ , _ ).

wife( X ) :-                                  % X là một người vợ
    family( _ , X , _ ).

chidren( X ) :-                               % X là một người con, chú ý các tên biến chữ hoa
    family( _ , _ , Chidren ),
    ismember( X, Chidren ).

ismember( X, [ X | L ] ).                    % có thể sử dụng mệnh đề member của Prolog
ismember( X, [ Y | L ] ) :-
    ismember( X, L ).

exist( Individual ) :-                       % mọi thành viên của gia đình
    husban( Individual ) ;
    wife( Individual ) ;
    chidren( Individual ).

dateofbirth( individual( _ , _ , Date , _ ), Date ).

salary( individual( _ , _ , _ , work( _ , S ) ), S ).    % thu nhập của người lao
động

salary( individual( _ , _ , _ , inactive ), 0 ).    % người không có nguồn thu
nhập
```

Bây giờ ta có thể đặt các câu hỏi như sau :

1. Tìm tên họ của những người có mặt trong cơ sở dữ liệu :  
?- exist( individual( Firstname, Lastname, \_ , \_ ) ).
2. Tìm những người con sinh năm 1991 :  
?- chidren( X ), dateofbirth( X, date( \_ , \_ , 1991 ) ).
3. Tìm những người vợ có việc làm :



- ?- wife( individual( Firstname, Lastname, \_ , work( \_ , \_ ) ) ).
4. Tìm những người không có việc làm sinh trước năm 1975 :
- ?- exist( individual( Firstname, Lastname, date( \_ , \_ , Year ) , inactive ) ),  
Year < 1975.
5. Tìm những người sinh trước năm 1975 có thu nhập dưới 10000 :
- ?- exist( Individual ),  
dateofbirth( Individual, date( \_ , \_ , Year ) ),  
Year < 1975,  
salary( Individual, Salary ),  
Salary < 10000.
6. Tìm những gia đình có ít nhất ba con :
- ?- family( individual( \_ , Name, \_ , \_ ), \_ , [ \_ , \_ , \_ | \_ ] ).

Để tính tổng thu nhập của một gia đình, ta có thể định nghĩa một quan hệ nhị phân cho phép tính tổng các thu nhập của một danh sách những người đang có việc làm dạng :

total( List\_of\_ individual, Sum\_of\_ salary )

Ta viết trong Prolog như sau :

```
total( [ ], 0 )                % danh sách rỗng
total( [ Individual | List ], Sum ) :-
    salary( Individual, S ),    % S là thu nhập của người đầu tiên
    total( List, Remain ),      % Remain là thu nhập của tất cả những người
                                còn lại
    Sum is S + Remain.
```

Như vậy, tổng thu nhập của một gia đình được tính bởi câu hỏi :

```
?- family( Husband, Wife, Children ),
    total( [ Husband, Wife | Children ], Income ).
```

Các phiên bản Prolog đều có thể tính độ dài (length) của một danh sách (xem mục III chương 1 trước đây, ta cũng đã tìm cách xây dựng quan hệ này). Bây giờ ta có thể áp dụng để tìm những gia đình có nguồn thu nhập nhỏ hơn 5000 tính theo đầu người :

```
?- family( Husband, Wife, Children ),
    total( [ Husband, Wife | Children ], Income )
    length( [ Husband, Wife | Children ], N ),
    Income / N < 5000.          % N là số người trong một gia đình
```

## II.2. Trừu tượng hoá dữ liệu

Trừu tượng hoá dữ liệu (data abstraction) được xem là cách tổ chức tự nhiên (một cách có thứ cấp) những thành phần khác nhau trong cùng những đơn vị thông tin, sao cho về mặt ý niệm, người sử dụng có thể hiểu được cấu trúc bên trong. Chương trình phải dễ dàng truy cập được vào từng thành phần dữ liệu. Một cách lý tưởng thì người sử dụng không nhìn thấy được những chi tiết cài đặt các cấu trúc này, người sử dụng chỉ quan tâm đến những đối tượng và quan hệ giữa chúng. Với mục đích đó, Prolog phải có cách biểu diễn thông tin phù hợp.

Để tìm hiểu cách Prolog giải quyết, ta quay lại ví dụ cơ sở dữ liệu gia đình trong mục trước đây. Mỗi gia đình là một nhóm các thông tin khác nhau về bản chất, mỗi người hay mỗi gia đình được xử lý như một đối tượng độc lập.

Giả thiết rằng mỗi gia đình được biểu diễn như Hình II.1. Bây giờ ta tiếp tục định nghĩa các quan hệ để có thể tiếp cận đến các thành phần của gia đình mà không cần biết chi tiết. Những quan hệ này được gọi là các *bộ chọn* (selector), vì chúng chọn những thành phần nào đó. Mỗi bộ chọn sẽ có tên là tên thành phần mà nó chọn ra, và có hai tham đối : đối tượng chứa thành phần được chọn và bản thân thành phần đó :

```
selector_relation( Object, Selected_component )
```

Sau đây là một số ví dụ về các bộ chọn :

```
husban( family( Husban, _ , _ ), Husban ).
```

```
wife( family( _ , Wife, _ ), Wife ).
```

```
chidren( family( _ , _ , ChidrenList ), ChidrenList ).
```

Ta cũng có thể định nghĩa những bộ chọn chọn ra những người con đặc biệt như con trưởng, con út và con thứ N trong gia đình :

```
eldest( Family, Eldest ) :-                               % người con trưởng
    chidren(Family, [ Eldest | _ ] ).
```

```
cadet( Family, Eldest ) :-                                 % người con út
    chidren( Family, [ Eldest | _ ] ).
```

Chọn ra một người con bất kỳ nào đó :

```
nth_child( N, Family, Chidren ) :-                         % người con thứ N trong gia đình
    chidren( Family, ChidrenList ),
```

% phần tử thứ N của một danh sách  
nth\_member( N, ChildrenList, Children ).

Từ biểu diễn cấu trúc minh hoạ trong Hình II.1, sau đây là một số bộ chọn nhận tham đối là một thành viên trong gia đình (individual) :

lastname( individual( \_ , Lastname, \_ , \_ ), Lastname ).    % tên gia đình (họ)  
firstname( individual( Firstname, \_ , Wife, \_ ), Firstname ).    % tên riêng  
born( individual( \_ , \_ , Date, \_ ), Date ).    % ngày sinh

Làm cách nào để có thể áp dụng các bộ chọn ? Mỗi khi các bộ chọn đã được định nghĩa, ta không cần quan tâm đến cách biểu diễn những thông tin có cấu trúc. Để tạo ra và để thao tác trên những thông tin cấu trúc, chỉ cần biết tên các bộ chọn và sử dụng chúng trong chương trình. Với phương pháp này, các biểu diễn phức tạp cấu trúc dữ liệu sẽ dễ dàng hơn so với phương pháp mô tả đã xét.

Ví dụ, người sử dụng không cần biết những người con trong gia đình được lưu giữ trong một danh sách. Giả sử rằng ta muốn hai người con Johan Smith và Eric Smith cùng thuộc một gia đình, và Eric là em thứ hai của Johan. Ta có thể sử dụng bộ chọn để định nghĩa hai cá thể, được gọi là Individual1 và Individual2, và định nghĩa gia đình như sau :

% Johan Smith  
lastname( Individual1, smith ), firstname( Individual1, johan ).  
% Eric Smith  
lastname( Individual2, smith ), firstname( Individual1, eric ),  
    husban( Family, Individual1 ).  
nth\_child( 2, Family, Individual2 ).

Việc sử dụng các bộ chọn làm thay đổi dễ dàng một chương trình Prolog. Giả sử ta muốn thay đổi dữ liệu của một chương trình, ta chỉ cần định nghĩa lại các bộ chọn, phần còn lại của chương trình vẫn hoạt động như cũ.

## II.3. Mô phỏng ô tômat hữu hạn

Ví dụ sau đây minh hoạ cách Prolog biểu diễn các mô hình toán học trừu tượng.

### II.3.1. Mô phỏng ô tômat hữu hạn không đơn định

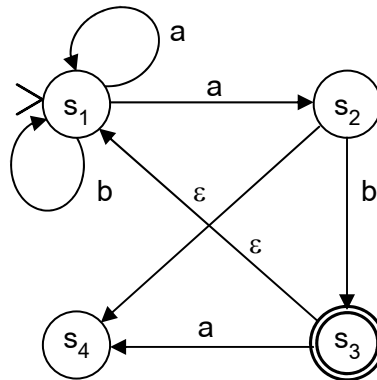
Một *ô tômat hữu hạn không đơn định* (Non-deterministic Finite Automaton, viết tắt NFA) là một máy trừu tượng có thể đọc một *câu vào* (input string) là một *xâu* (hay *chuỗi*) ký tự nào đó và có thể quyết định có *thừa nhận* (accept) hay *không thừa nhận* (rejecting). Ô tômat có một số hữu hạn *trạng thái* (state) và luôn ở một trạng thái nào đó để có thể *chuyển tiếp* (transition) qua một trạng thái khác sau khi đọc (thừa nhận) một ký hiệu (symbol) hay ký tự thuộc một *bảng ký tự* (alphabet hay set of characters) hữu hạn nào đó. Một *xâu* đã cho được gọi là *được thừa nhận* bởi ô tômat, nếu sau khi đọc hết câu vào, ô tômat rơi vào một trong các trạng thái thừa nhận.

Người ta thường biểu diễn ô tômat hữu hạn bởi một đồ thị định hướng mô tả các chuyển tiếp trạng thái có thể. Mỗi cung định hướng của đồ thị được gắn nhãn là ký tự sẽ đọc. Mỗi nút của đồ thị là một trạng thái, trong đó, *trạng thái đầu* (initial state) được đánh dấu bởi  $\triangleright$ , và các *trạng thái thừa nhận* (accepted state) được đánh dấu bởi đường kép.

Hình 5.3 dưới đây minh hoạ một ô tômat hữu hạn không đơn định có bốn trạng thái  $s_1$ ,  $s_2$ ,  $s_3$  và  $s_4$ , trong đó,  $s_1$  là trạng thái đầu và ô tômat chỉ có một trạng thái thừa nhận duy nhất là  $s_3$ . Chú ý ô tômat có hai chuyển tiếp nối vòng (chu kỳ) tại trạng thái  $s_1$  (nghĩa là ô tômat không thay đổi trạng thái sau khi đọc xong hoặc ký tự a, hoặc ký tự b).

Mỗi chuyển tiếp của ô tômat được xác định bởi một quan hệ giữa trạng thái hiện hành, ký tự sẽ đọc và trạng thái sẽ đạt tới. Chú ý rằng mỗi chuyển tiếp có thể không đơn định. Trong Hình II.3, từ trạng thái  $s_1$ , sau khi đọc ký tự a, ô tômat có thể rơi vào hoặc trạng thái  $s_1$ , hoặc trạng thái  $s_2$ . Ta cũng thấy một số cung có nhãn  $\varepsilon$  (câu rỗng), tương ứng với "chuyển tiếp epsilon", ký hiệu  $\varepsilon$ -chuyển tiếp. Những cung này mô tả sự chuyển tiếp "không nhìn thấy được" của ô tômat: ô tômat chuyển qua một trạng thái mới khác mà

không hề đọc một ký tự nào. Nghĩa là phần câu vào vẫn không thay đổi, nhưng ô tômat đã thay đổi trạng thái.



Hình II.3. Một ô tômat hữu hạn không đơn định bốn trạng thái.

Người ta nói ô tômat thừa nhận câu vào nếu tồn tại một dãy các chuyển tiếp trong đồ thị sao cho :

1. Lúc đầu, ô tômat ở trạng thái đầu (ví dụ  $s_1$ ).
2. Ô tômat kết thúc việc đoán nhận câu vào và ở trạng thái thừa nhận ( $s_3$ ).
3. Các nhãn trên các cung của con đường chuyển tiếp từ trạng thái đầu đến trạng thái thừa nhận tương ứng với câu vào là xâu đã đọc.

Trong quá trình đoán nhận câu vào, ô tômat quyết định lựa chọn một trong số các chuyển tiếp có thể để tiếp tục. Đặc biệt, ô tômat có thể thực hiện hay không thực hiện một  $\varepsilon$ -chuyển tiếp, nếu trạng thái hiện hành cho phép. Ô tômat không thừa nhận câu vào nếu nó không rơi vào trạng thái thừa nhận dù đã đọc hết câu vào, hoặc không còn khả năng tiếp tục chuyển tiếp mà câu vào chưa kết thúc, hoặc có thể bị quẩn vô hạn.

Như đã biết, các ô tômat hữu hạn không đơn định trừu tượng có một tính chất thú vị : tại mỗi thời điểm, ô tômat có khả năng lựa chọn, trong số các chuyển tiếp có thể, một chuyển tiếp "tốt nhất" để thừa nhận câu vào.

Chẳng hạn, ô tômat cho ở Hình II.3 sẽ thừa nhận các xâu  $ab$  và  $aabaab$ , nhưng không thừa nhận các xâu  $abb$  và  $abba$ . Một cách tổng quát, ô tômat thừa nhận mọi xâu kết thúc bởi  $ab$ , nhưng không thừa nhận các xâu khác.

Trong Prolog, một ô tômat được định nghĩa bởi ba quan hệ :

1. Một quan hệ một ngôi *satisfaction* cho phép xác định các trạng thái thừa nhận của ô tômat.
2. Một quan hệ ba ngôi *trans* cho phép xác định các trạng thái chuyển tiếp, chẳng hạn :  
`trans( S1, X, S2 ).`  
 có nghĩa là ô tômat chuyển tiếp từ trạng thái S1 qua trạng thái S2 sau khi đọc ký tự X.
3. Một quan hệ hai ngôi *epsilon* chỉ ra phép chuyển tiếp rỗng từ trạng thái S1 qua trạng thái S2 :  
`epsilon( S1, S2 ).`

Ô tômat đã cho ở Hình II.3 được mô tả bởi các mệnh đề Prolog như sau :

`satisfaction( s3 ).`

`trans( s1, a, s1 ).`

`trans( s1, a, s2 ).`

`trans( s1, b, s1 ).`

`trans( s2, b, s3 ).`

`trans( s3, b, s4 ).`

`epsilon( s2, s4 ).`

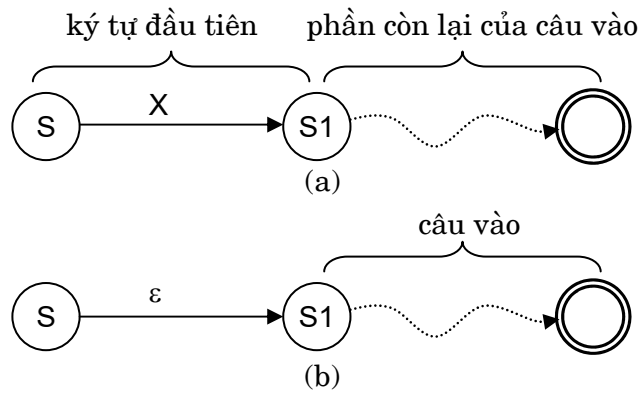
`epsilon( s3, s1 ).`

Để biểu diễn các xâu ký tự trong Prolog, ta sẽ sử dụng kiểu danh sách. Chẳng hạn xâu *aab* được biểu diễn bởi `[ a, b, a ]`. Xuất phát từ một câu vào, ô tômat vừa mô tả trên đây sẽ mô phỏng quá trình đoán nhận, bằng cách đọc lần lượt các phần tử của danh sách, để thừa nhận hay không thừa nhận.

Theo định nghĩa, ô tômat hữu hạn không đơn định sẽ thừa nhận câu vào nếu, xuất phát từ trạng thái đầu, sau khi đọc hết câu (xử lý hết mọi phần tử của danh sách), ô tômat rơi vào trạng thái thừa nhận. Quan hệ hai ngôi *accept* sau đây cho phép mô phỏng quá trình đoán nhận một câu vào từ một trạng thái đã cho :

`accept( State, InputString )`

Quan hệ *accept* là đúng nếu *State* là trạng thái đầu và *InputString* là một câu vào.



Hình II.4. Ôtômat thừa nhận câu vào :

(a) đọc ký tự đầu tiên X ; (b) thực hiện một  $\epsilon$ -chuyển tiếp.

Ba mệnh đề cho phép định nghĩa quan hệ này, tương ứng với ba trường hợp như sau :

1. Xâu rỗng [ ] được thừa nhận tại trạng thái S nếu ô tômat đang ở tại trạng thái S và S là một trạng thái thừa nhận.
2. Một xâu khác rỗng được thừa nhận tại trạng thái S nếu đầu đọc đang ở tại vị trí đọc ký tự đầu tiên của xâu để sau khi đọc, ô tômat chuyển qua trạng thái S1 và xuất phát từ trạng thái S1 này, ô tômat thừa nhận toàn bộ phần còn lại của câu vào (xem minh hoạ ở Hình II.4 (a) ).
3. Một xâu khác rỗng được thừa nhận tại trạng thái S nếu ô tômat có thể thực hiện một  $\epsilon$ -chuyển tiếp từ trạng thái S qua trạng thái S1 và xuất phát từ trạng thái S1 này, ô tômat thừa nhận toàn bộ phần còn lại của câu vào (xem minh hoạ ở Hình II.4 (b) ).

Ta có thể viết trong Prolog như sau :

```
accept( S, [ ] ) :-                               % thừa nhận xâu rỗng
    satisfaction( S ).

accept( S, [ X | Remainder ] ) :-                  % thừa nhận sau khi đọc ký tự đầu tiên
    trans( S, X, S1 ),
    accept( S1, Remainder ).

accept( S, InputString ) :-                        % thừa nhận bởi  $\epsilon$ -chuyển tiếp
    epsilon( S, S1 ),
    accept( S1, InputString ).
```

Bây giờ, ta có thể yêu cầu ô tômat nhận biết chuỗi aaab bởi câu hỏi sau :

```
?- accept( s1, [ a, a, a, b ] ).
```

Yes

Tuy nhiên, ô tômat không thừa nhận chuỗi abbb :

```
?- accept( s1, [ a, b, b, b ] ).
```

ERROR: Out of local stack

Ta cũng thấy rằng các chương trình Prolog thường giải quyết các bài toán tổng quát hơn những gì mà NLT tạo ra chúng. Ví dụ, để yêu cầu Prolog cho biết trạng thái đầu nào thì chuỗi ab được thừa nhận :

```
?- accept( S, [ a, b ] ).
```

S = s1

Yes

Thú vị hơn nữa, ta có thể yêu cầu Prolog cho biết những chuỗi ba ký tự nào thì được thừa nhận bởi ô tômat :

```
?- accept( s1, [ X1, X2, X3 ] ).
```

X1 = a

X2 = a

X3 = b

Yes

Nếu ta muốn kết quả trả về là một chuỗi, ta chỉ cần đặt câu hỏi :

```
?- InputString = [ _ , _ , _ ], accept( s1, InputString ).
```

InputString = [a, a, b]

Yes

Đi xa hơn, tại sao ta không thể yêu cầu Prolog cho biết những trạng thái đầu nào của ô tômat cho phép nhận biết những chuỗi có bảy ký tự, v.v... ?

Cần phải có những thay đổi trên các quan hệ satisfaction, trans và epsilon nếu ta muốn ô tômat thực hiện những xử lý tổng quát hơn. Ô tômat đã cho ở Hình II.4 không chứa các nối vòng  $\epsilon$ -chuyển tiếp. Bây giờ nếu ta thêm một chuyển tiếp :

```
epsilon( s1, s3 ).
```

thì ta đã tạo ra một nối vòng trên chuỗi rỗng  $\epsilon$  làm rối loạn chức năng đoán nhận của ô tômat. Lúc này với câu hỏi :

```
?- accept( s1, [ a ] ).
```



sẽ gây ra một vòng lặp quần vô hạn tại trạng thái  $s_1$ , trong khi ôôtômat cố gắng tìm một con đường đến trạng thái thừa nhận  $s_3$ .

### II.3.2. Mô phỏng ôôtômat hữu hạn đơn định

Một ôôtômat hữu hạn là *đơn định* (Deterministic Finite Automaton, viết tắt DFA) nếu chuyển tiếp của ôôtômat được xác định đơn định : ôôtômat chỉ có thể chuyển qua một và chỉ một trạng thái tiếp theo sau khi đọc một ký tự và không có các « chuyển tiếp epsilon ». Thay vì sử dụng thuật ngữ quan hệ ba ngôi, người ta thường sử dụng thuật ngữ *hàm chuyển tiếp* delta  $\delta(s, a) = s'$  để mô tả các hoạt động đoán nhận câu của ôôtômat đơn định.

DFA được viết trong Prolog như sau :

```
parse(L) :-
    start(S),
    trans(S,L).

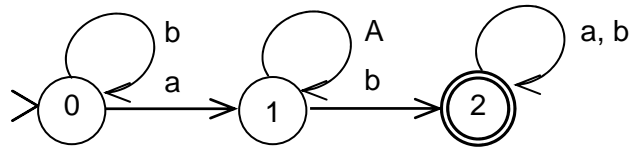
trans(X,[A|B]) :-
    delta(X,A,Y),                % X ---A---> Y
    write(X),
    write(' '),
    write([A|B]),
    nl,
    trans(Y,B).

trans(X,[]) :-
    final(X),
    write(X),
    write(' '),
    write([]), nl.
```

DFA sau đây thừa nhận ngôn ngữ  $(a,b)^*ab(a,b)^*$  :

```
start(0).
final(2).
delta(0,a,1).
delta(0,b,0).
delta(1,a,1).
delta(1,b,2).
delta(2,a,2).
delta(2,b,2).
```

Sơ đồ biểu diễn ô tômat như sau :



Hình II.5. Ô tômat hữu hạn đơn định có ba trạng thái.

Sau đây là một số hoạt động đoán nhận của ô tômat :

?- parse([b,b,a,a,b,a,b]).

0 [b, b, a, a, b, a, b]

0 [b, a, a, b, a, b]

0 [a, a, b, a, b]

1 [a, b, a, b]

1 [b, a, b]

2 [a, b]

2 [b]

2 []

Yes

?- parse([b,b,a]).

0 [b, b, a]

0 [b, a]

0 [a]

No

## II.4. Ví dụ : lập kế hoạch đi du lịch bằng máy bay

Trong mục này, ta sẽ xây dựng một chương trình Prolog cho phép lập kế hoạch để đi du lịch bằng máy bay. Dẫu rằng đơn giản, những ví dụ này trả lời được những câu hỏi mang tính thực tiễn sau đây :

- Những ngày nào trong tuần có chuyến bay trực tiếp từ Paris đi Ljubljana ?
- Làm cách nào để đi từ Ljubljana đến Grenoble ngày thứ Năm ?

- Tôi phải đi du lịch Milan, Ljubljana và Zurich xuất phát từ Paris ngày thứ Ba và phải quay về trong ngày thứ Sáu. Làm sao để có thể sắp xếp các chuyến đi của tôi sao cho mỗi ngày không đi máy bay quá một lần ?

Chương trình Prolog được dựa trên một cơ sở dữ liệu chứa những thông tin về các chuyến bay. Mỗi chuyến bay là một quan hệ ba ngôi cho biết lịch trình bay timetable như sau :

timetable( Place1, Place2, Fly\_List).

Danh sách các chuyến bay có dạng như sau :

Departure\_hour / Arrival\_hour / Fly\_Number / Day\_List

Danh sách các ngày có chuyến bay hoặc là một danh sách các ngày thứ trong tuần, hoặc là một nguyên tử all (cho tất cả các ngày). Chẳng hạn, sau đây là một quan hệ timetable :

```
timetable( paris , grenoble ,
           [ 9:40 / 10:50 / ba4732 / all ,
             11:40 / 12:50 / ba4752 / all ,
             18:40 / 19:50 / ba4822 / [ mo , tu , we , th , fr ] ] ).
```

Lịch trình bay được biểu diễn bởi các cấu trúc hai thành phần là giờ và phút phân cách nhau bởi phép toán : (dấu hai chấm).

Bài toán chính đặt ra là tìm những lộ trình chính xác giữa hai thành phố (nơi đi và nơi đến) và một ngày nào đó đã cho. Để lập trình, ta sử dụng một quan hệ có bốn tham đối như sau :

path( Place1 , Place2 , Day, Path )

trong đó, là một dãy các chuyến bay thoả mãn các tiêu chuẩn sau :

- (1) Nơi đi là Place1.
- (2) Nơi đến là Place2.
- (3) Tất cả những chuyến bay cùng ngày Day trong tuần.
- (4) Tất cả những chuyến bay của lộ trình Path thuộc về quan hệ timetable.
- (5) Có đủ thời gian để di chuyển giữa các chuyến bay.

Lộ trình được biểu diễn bởi một danh sách các đối tượng như sau :

Departure - Arrival : Fly\_number : Departure\_hour

Ta cũng sử dụng các vị từ hỗ trợ sau đây :

(1) fly( Place1, Place2 , Day , Fly\_number , Departure\_hour, Arrival\_hour )

Có nghĩa là tồn tại một chuyến bay số hiệu Fly\_number giữa Place1 và Place2 trong ngày Day, tương ứng với ngày đi và ngày đến đã cho.

(2) dephour( Path , Hour )

Giờ xuất phát của lộ trình Path là Hour.

(3) connecting( Hour1, Hour2 )

Có ít nhất 40 phút giữa Hour1 và Hour2, cho phép thực hiện việc di chuyển (nối tiếp giữa hai chuyến bay).

Vấn đề tìm một lộ trình giữa hai thành phố tương tự bài toán đoán nhận sâu của một ô tômat hữu hạn không đơn định đã xét trong mục trước. Những điểm chung là :

- Các trạng thái của ô tômat tương ứng với các thành phố.
- Một chuyển tiếp giữa hai trạng thái tương ứng với các chuyến bay giữa hai thành phố.
- Quan hệ trans của ô tômat tương ứng với quan hệ timetable.
- Để mô phỏng quá trình đoán nhận câu, ô tômat tìm được một lộ trình giữa trạng thái đầu và một trạng thái thừa nhận. Còn để mô phỏng việc lập kế hoạch đi du lịch, chương trình tìm được một lịch trình bay giữa thành phố xuất phát và thành phố đến.

Chính vì vậy, ta có thể định nghĩa một quan hệ về lộ trình path tương tự với quan hệ accept, chỉ có khác là quan hệ path không chứa chuyển tiếp rỗng.

Xây ra hai trường hợp như sau :

(1) Nếu có một chuyến bay trực tiếp giữa P1 và P2 thì lộ trình được rút gọn thành :

path( P1 , P2 , Day, [ P1 - P2 : FlyNum : DepH ] ) :-  
fly( P1 , P2 , Day , FlyNum , Dep , Arr ).

(2) Nếu không có một chuyến bay trực tiếp giữa P1 và P2 thì lộ trình sẽ phải bao gồm một chuyến bay giữa P1 và một thành phố trung gian

P3, rồi một chuyến bay giữa P3 và P2. Lúc này cần có đủ thời gian để di chuyển giữa hai chuyến bay, từ nơi đến của chuyến bay thứ nhất đến nơi xuất phát của chuyến bay thứ hai :

```
path( P1 , P2 , Day , [ P1 - P3 : FlyNum : Dep1 | Path ] ) :-
    path( P3 , P2 , Day , Path ),
    fly( P1 , P3 , Day , FlyNum1 , Dep1 , Arr1 ),
    dephour( Path , Dep2 ) ,
    connecting( Arr1 , Dep2 ).
```

Các quan hệ fly, connecting và dephour được xây dựng tương đối dễ dàng. Dưới đây là chương trình đầy đủ bao gồm cơ sở dữ liệu về lịch trình bay.

Ví dụ này đơn giản, không xảy ra trường hợp có lộ trình vô ích, nghĩa là một lộ trình không dẫn đến đâu. Ta cũng thấy rằng cơ sở dữ liệu về lịch trình bay còn nhỏ. Để có thể quản lý một cơ sở dữ liệu lớn hơn, nhất thiết phải sử dụng một chương trình lập kế hoạch thông minh hơn.

% Chương trình lập kế hoạch đi du lịch

:- op( 50 , xfy , : ).

```
fly( Place1, Place2 , Day , FlyNum , DepH , ArrH ) :-
    timetable( Place1 , Place2 , FlyList ) ,
    ismember( DepH / ArrH / FlyNum / DayList , FlyList ) ,
    flyday( Day , DayList ).
```

```
ismember( X , [ X | L ] ).
```

```
ismember( X , [ Y | L ] ) :-
    ismember( X , L ).
```

```
flyday( Day , DayList ) :-
    ismember( Day , DayList ).
```

```
flyday( Day , all ) :-
    ismember( Day , [ mo , tu , we , th , fr , sa , su ] ).
```

% Chuyến bay trực tiếp

```
path( P1 , P2 , Day , [ P1 - P2 : FlyNum : DepH ] ) :-
    fly( P1 , P2 , Day , FlyNum , DepH , _ ).
```

% Chuyến bay không trực tiếp

```
path( P1 , P2 , Day , [ P1 - P3 : FlyNum : Dep1 | Path ] ) :-
    path( P3 , P2 , Day , Path ),
    fly( P1 , P3 , Day , FlyNum1 , Dep1 , Arr1 ),
    dephour( Path , Dep2 ) ,
    connecting( Arr1 , Dep2 ).
```

```
dephour( [ P1 - P2 : FlyNum : Dep | _ ] , Dep ).
```

```

connecting( Hour1 : Mins1 , Hour2 : Mins2 ) :-
    60 * ( Hour2 - Hour1 ) + Mins2 - Mins1 >= 40.
% Mäüt cả sải dăi liău văo lăch trănă căic chuyăún bay
timetable( grenoble , paris ,
    [ 9 :40 / 10:50 / ba4733 / all ,
      13 :40 / 14:50 / ba4773 / all ,
      19:40 / 20:50 / ba4833 / [ mo , tu , we , th , fr , su ] ] ).
timetable( paris , grenoble ,
    [ 9:40 / 10:50 / ba4732 / all ,
      11:40 / 12:50 / ba4752 / all ,
      18:40 / 19:50 / ba4822 / [ mo , tu , we , th , fr ] ] ).
timetable( paris , ljubljana ,
    [ 13:20 / 16:20 / ju201 / [ fr ] ,
      13:20 / 16:20 / ju213 / [ su ] ] ).
timetable( paris , zurich ,
    [ 9:10 / 11:45 / ba614 / all ,
      14:45 / 17:20 / sr805 / all ] ).
timetable( paris , milan ,
    [ 8:30 / 11:20 / ba510 / all ,
      11:00 / 13:50 / az459 / all ] ).
timetable( ljubljana , zurich ,
    [ 11:30 / 12:40 / ju322 / [ tu , fr ] ] ).
timetable( ljubljana , paris ,
    [ 11:10 / 12:20 / yu200 / [ fr ] ,
      11:25 / 12:20 / yu212 / [ su ] ] ).
timetable( milan , paris ,
    [ 9:10 / 10 :00 / az458 / all ,
      12:20 / 13:10 / ba511 / all ] ).
timetable( milan , zurich ,
    [ 9:25 / 10:15 / sr621 / all ,
      12:45 / 13:35 / sr623 / all ] ).
timetable( zurich , ljubljana ,
    [ 13:30 / 14:40 / yu323 / [ tu , th ] ] ).
timetable( zurich , paris ,
    [ 9:00 / 9:40 / ba613 / [ mo , tu , we , th , fr , sa ] ,
      16:10 / 16:55 / sr806 / [ mo , tu , we , th , fr , su ] ] ).
timetable( zurich , milan ,
    [ 7:55 / 8:45 / sr620 / all ] ).

```

---

Sau đây là một số câu hỏi trên cơ sở dữ liệu về lịch trình hàng không :

- Những ngày nào trong tuần có một chuyến bay trực tiếp giữa Paris và Ljubljana ?

?- fly( paris , ljubljana , Day , \_ , \_ , \_ ).

Day = fr;

Day = su;

No

- Làm cách nào để có thể đi từ Ljubljana đến Grenoble ngày thứ năm ?

?- path( ljubljana , grenoble , th, C ).

C = [ ljubljana-paris:yu200:11:10, paris-grenoble:ba4822:18:40 ] ;

C = [ ljubljana-paris:yu212:11:25, paris-grenoble:ba4822:18:40 ] ;

C = [ ljubljana-zurich:ju322:11:30, zurich-paris:sr806:16:10,  
paris-grenoble:ba4822:18:40 ]

- Làm cách nào để xuất phát từ Paris, có thể du lịch Milan, Ljubljana và Zurich trong ngày thứ Ba, để trở về trong ngày thứ Sáu, sao cho mỗi ngày chỉ thực hiện không quá một chuyến bay ?

Đây là một câu hỏi tương đối lúng củng. Để trả lời, ta cần sử dụng quan hệ permutation đã trình bày trong chương 1, mục 3. Quan hệ này cho phép hoán vị tất cả các thành phố Milan, Ljubljana và Zurich sao cho tồn tại những chuyến bay thích hợp mỗi ngày :

?- permutation( [ milan , ljubljana , zurich ] , [ V1, V2, V3 ] ),  
fly( paris, V1, tu, FN1, Dep1, Arr1 ),  
fly( V1, V2, tu, FN2, Dep2, Arr2 ),  
fly( V2, V3, tu, FN3, Dep3, Arr3 ),  
fly( V3, paris, fr, FN4, Dep4, Arr4 ).

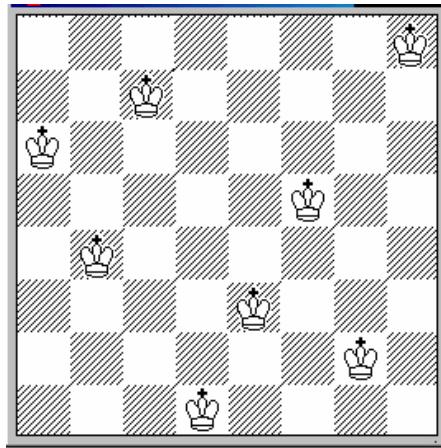
Kết quả -> V1 = ljubljana  
V2 = zurich  
V3 = milan  
FN1 = ju213  
Dep1 = 13:20  
Arr1 = 16:20  
FN2 = ju322  
Dep2 = 11:30  
Arr2 = 12:40  
FN3 = sr620  
Dep3 = 7:55  
Arr3 = 8:45  
FN4 = az458  
Dep4 = 9:10  
Arr4 = 10:0 ;

-> V1 = milan  
V2 = zurich  
V3 = ljubljana  
FN1 = ba510  
Dep1 = 8:30  
Arr1 = 11:20  
FN2 = sr621  
Dep2 = 9:25  
Arr2 = 10:15  
FN3 = yu323  
Dep3 = 13:30  
Arr3 = 14:40  
FN4 = yu200  
Dep4 = 11:10  
Arr4 = 12:20

## II.5. Bài toán tám quân hậu

Bài toán tám quân hậu do Carl Friedrich Gauss đưa ra vào năm 1850 nhưng không có lời giải hoàn toàn theo phương pháp giải tích. Sau đó bài toán này được nhiều người giải trọn vẹn trên MTĐT, theo nhiều cách khác nhau. Bài toán phát biểu như sau :

Hãy tìm cách đặt tám quân hậu lên một bàn cờ vua (có 8 x 8 ô, lúc đầu không chứa quân nào) sao cho không có quân nào ăn được quân nào ? Một quân hậu có thể ăn được bất cứ quân nào nằm trên cùng cột, hay cùng hàng, hay cùng đường chéo thuận, hay cùng đường chéo nghịch với nó.



Hình II.6. Một lời giải của bài toán tám quân hậu

Niclaus Wirth trình bày phương pháp *thử-sai* (trial-and-error) như sau :

- Đặt một quân hậu vào cột 1 (trên một hàng tùy ý);
- Đặt tiếp một quân hậu thứ hai sao cho 2 quân không ăn nhau;
- Tiếp tục đặt quân thứ 3, v.v...

Lời giải có dạng một vòng lặp theo giả ngữ Pascal như sau :

```
Xét-cột-đầu ;
repeat
  Thử_cột ;
  if An_toàn then begin
    Đặt_quân_hậu_vào ;
    Xét_cột_kế_tiếp;
  end else Quay_lại ;
until Đã_xong_với_cột_cuối or Đã_quay_lại_quá_cột_đầu ;
```



Với Prolog, chương trình sẽ có dạng một vị từ :

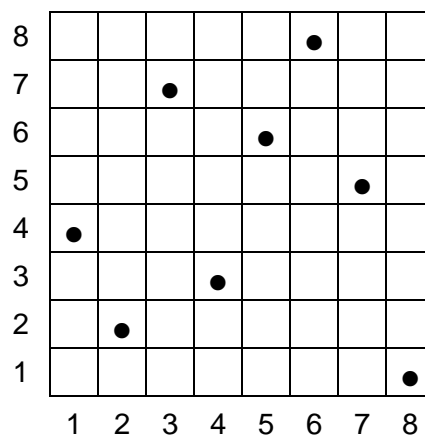
solution( Pos )

Vị từ này chỉ thoả mãn khi và chỉ khi Pos biểu diễn một cách bố trí tám quân hậu sao cho không có quân nào ăn được quân nào. Sau đây ta sẽ trình bày ba cách tiếp cận để lập trình Prolog dựa trên các cách biểu diễn khác nhau.

### II.5.1. Sử dụng danh sách tọa độ theo hàng và cột

Ta cần tìm cách biểu diễn các vị trí trên bàn cờ. Giải pháp trực tiếp nhất là sử dụng một danh sách tám phần tử mà mỗi phần tử tương ứng với ô đặt quân hậu. Mỗi phần tử là một cặp số nguyên giữa 1 và 8 chỉ tọa độ của quân hậu :

X / Y



Hình II.7. Một lời giải của bài toán tám quân hậu, biểu diễn bởi danh sách [ 1/4, 2/2, 3/7, 4/3, 5/6, 6/8, 7/5, 8/1 ] .

Ở đây, phép toán / không phải là phép chia, mà chỉ là cách tổ hợp hai tọa độ của một ô bàn cờ. Hình 5.6 trên đây là một lời giải khác của bài toán tám quân hậu được biểu diễn dưới dạng một danh sách như sau :

[ 1/4, 2/2, 3/7, 4/3, 5/6, 6/8, 7/5, 8/1 ]

Từ cách biểu diễn danh sách, ta cần tìm lời giải có dạng :

[ X1/Y1, X2/Y2, X3/Y3, ... , X8/Y8 ]

Ta cần tìm các giá trị của các biến  $X1, Y1, X2, Y2, X3, Y3, \dots, X8, Y8$ . Do các quân hậu phải nằm trên các cột khác nhau để không thể ăn lẫn nhau, nên ta có ngay giá trị của các tọa độ  $X$ , và lời giải lúc này có dạng :

[ 1/Y1, 2/Y2, 3/Y3, ... , 8/Y8 ]

Cho đến lúc này, bài toán tám quân hậu chỉ đặt ra đối với bàn cờ  $8 \times 8$ . Tuy nhiên, lời giải phải dự kiến được cho trường hợp tổng quát khi lập trình. Ở đây, ta sẽ thấy rằng chính trường hợp tổng quát lại đơn giản hơn bài toán ban đầu. Bàn cờ  $8 \times 8$  chỉ là một trường hợp riêng.

Để giải quyết cho trường hợp tổng quát, ta chuyển kích thước 8 quân hậu thành một số quân hậu bất kỳ nào đó (mỗi cột một quân hậu), kể cả số cột bằng không. Ta xây dựng quan hệ solution từ hai tình huống sau :

1. Danh sách các quân hậu là rỗng : danh sách rỗng cũng là một lời giải vì không xảy ra sự tấn công nào.  
solution( [ ] ).
2. Danh sách các quân hậu khác rỗng và có dạng như sau :  
[ X/Y | Others ]

Trong trường hợp thứ hai, quân hậu thứ nhất nằm trên ô  $X/Y$ , còn những quân hậu khác nằm trong danh sách Others. Nếu danh sách này là một lời giải, thì những điều kiện sau đây phải được thoả mãn :

1. Những quân hậu trong danh sách Others không thể tấn công lẫn nhau, điều này nói lên rằng Others cũng là một lời giải.
2. Vị trí  $X$  và  $Y$  của những quân hậu phải nằm giữa 1 và 8.
3. Một quân hậu tại vị trí  $X/Y$  không thể tấn công một quân hậu nào khác trong danh sách Others.

Đối với điều kiện thứ nhất, quan hệ solution phải được gọi một cách đệ quy.

Điều kiện thứ hai nói lên rằng  $Y$  phải thuộc về danh sách [ 1, 2, 3, 4, 5, 6, 7, 8 ]. Ở đây, ta không cần quan tâm đến vị trí  $X$ , vì nó phải tương hợp với danh sách kết quả trả về như ta đã xác định ngay từ đầu. Nghĩa là  $X$  phải thuộc về những giá trị đã được ấn định tương ứng.

Giả sử điều kiện thứ ba được giải quyết nhờ quan hệ noattack, chương trình Prolog cho quan hệ solution như sau :

```
solution( [ X/Y | Others ] ) :-
    solution( Others ),
    member( Y, [ 1, 2, 3, 4, 5, 6, 7, 8 ] ),
    noattack( X/Y, Others ).
```

Bây giờ ta cần tìm quan hệ noattack. Ta thấy :

1. Nếu danh sách Rlist rỗng, khi đó noattack là đúng, vì không có quân hậu nào tấn công quân hậu tại X/Y nào đó.  
noattack( \_, [ ] ).
2. Nếu danh sách Rlist khác rỗng, khi đó có dạng [ R1 | Rlist1 ] và hai điều kiện sau đây phải được thoả mãn :
  - (a) Quân hậu tại ô R không thể tấn công quân hậu tại ô R1, và
  - (b) Quân hậu tại ô R không thể tấn công quân hậu nào trong Rlist1.

Để một quân hậu không thể tấn công quân hậu khác, thì chúng không thể nằm trên cùng hàng, cùng cột và cùng đường chéo chính hoặc phụ. Ta biết chắc chắn rằng các quân hậu đã nằm trên các cột phân biệt nhau do mô hình lời giải đã ấn định. Bây giờ ta cần chỉ ra rằng :

- Các toạ độ Y của các quân hậu phải phân biệt nhau, và
- Các quân hậu không thể nằm trên cùng đường chéo chính hoặc phụ. nghĩa là khoảng cách giữa các ô trên trục X phải khác với các ô trên trục Y.

```
noattack( X/Y, [ X1/Y2 | Others ] ) :-
    Y      =\= Y1,
    Y1 - Y =\= X1 - X,
    Y1 - Y =\= X - X1,
    noattack( X/Y, Others ).
```

Dưới đây là chương trình Prolog đầy đủ thứ nhất có chứa danh sách lời giải là quan hệ model. Mô hình làm cho việc tìm lời giải cho bài toán tám quân hậu trở nên đơn giản hơn.

---

```
% chương trình thứ nhất giải bài toán tám quân hậu
% The problem of the eight queens - Program 1
% -----
```

```

solution( [ ] ).
solution( [ X/Y | Others ] ) :-
    solution( Others ),
    ismember( Y, [ 1, 2, 3, 4, 5, 6, 7, 8 ] ),
    noattack( X/Y, Others ).
noattack( _ , [ ] ).
noattack( X/Y, [ X1/Y1 | Others ] ) :-
    Y \= Y1,
    Y1 - Y \= X1 - X,
    Y1 - Y \= X - X1,
    noattack( X/Y, Others ).
ismember( X , [ X | L ] ).
ismember( X, [ Y | L ] ) :-
    ismember( X, L ).
model( [ 1/Y1, 2/Y2, 3/Y3, 4/Y4, 5/Y5, 6/Y6, 7/Y7, 8/Y8 ] ).

```

---

```

?- model( S ), solution( S ).
S = [1/4, 2/2, 3/7, 4/3, 5/6, 6/8, 7/5, 8/1] ;
S = [1/5, 2/2, 3/4, 4/7, 5/3, 6/8, 7/6, 8/1] ;
S = [1/3, 2/5, 3/2, 4/8, 5/6, 6/4, 7/7, 8/1] ;
S = [1/3, 2/6, 3/4, 4/2, 5/8, 6/5, 7/7, 8/1] ;
S = [1/5, 2/7, 3/1, 4/3, 5/8, 6/6, 7/4, 8/2] ;
S = [1/4, 2/6, 3/8, 4/3, 5/1, 6/7, 7/5, 8/2]
Yes

```

Sử dụng vị từ `not`, ta viết lại chương trình như sau :

```

solution( [ ] ).
solution( [ X/Y | Others ] ) :-
    solution( Others ),
    member( Y, [ 1, 2, 3, 4, 5, 6, 7, 8 ] ),
    not( attack( X/Y, Others ) ).

attack( X/Y, Others ) :-
    member( X1/Y1, Others ),
    ( Y1 = Y,
      Y1 is Y + X1 - X;
      Y1 is Y - X1 + X ).

member( A, [ A | L ] ).
member( A, [ B | L ] ) :-
    member( A, L ).

% Mô hình lời giải

```

```

model( [ 1/Y1, 2/Y2, 3/Y3, 4/Y4, 5/Y5, 6/Y6, 7/Y7, 8/Y8 ] ).
?- model( S ), solution( S ).
S = [1/1, 2/1, 3/1, 4/1, 5/1, 6/1, 7/1, 8/1] ;
S = [1/1, 2/8, 3/1, 4/1, 5/1, 6/1, 7/1, 8/1] ;
S = [1/2, 2/8, 3/1, 4/1, 5/1, 6/1, 7/1, 8/1] ;
S = [1/1, 2/1, 3/7, 4/1, 5/1, 6/1, 7/1, 8/1] ;
S = [1/3, 2/1, 3/7, 4/1, 5/1, 6/1, 7/1, 8/1] ;
S = [1/1, 2/2, 3/7, 4/1, 5/1, 6/1, 7/1, 8/1]
Yes

```

## II.5.2. Sử dụng danh sách tọa độ theo cột

Trong chương trình thứ nhất, ta đã đưa ra lời giải biểu diễn bàn cờ có dạng :

[ 1/Y1, 2/Y2, 3/Y3, ... , 8/Y8 ]

do mỗi cột chỉ đặt đúng một quân hậu. Thực ra, ta không mất thông tin nếu bỏ đi các tọa độ X. Ta có thể biểu diễn bàn cờ chỉ với các tọa độ Y của các quân hậu :

[ Y1, Y2, Y3, ... , Y8 ]

Để không xảy ra các quân hậu nằm trên cùng cột, cần phải bố trí mỗi quân hậu một hàng. Từ đây ta đặt ra ràng buộc cho các tọa độ Y : mỗi hàng 1, 2, 3, ..., 8 của bàn cờ chỉ được phép đặt duy nhất một quân hậu. Ta nhận thấy rằng mỗi lời giải là một hoán vị của danh sách các số 1 .. 8 sao cho thứ tự của mỗi con số là khác nhau :

[ 1, 2, 3, 5, 6, 7, 8 ]

Mỗi hoán vị của danh sách là một lời giải S sao cho các quân hậu ở trạng thái an toàn (không ăn được lẫn nhau). Ta có :

```

solution( S ) :-
    permutation( [ 1, 2, 3, 5, 6, 7, 8 ], S ),
    insafety( S ).

```

Trong chương 1 trước đây, ta đã xây dựng quan hệ permutation, bây giờ ta cần định nghĩa quan hệ safety. Xảy ra hai trường hợp như sau :

1. Nếu danh sách S rỗng, khi đó S cũng là lời giải, vì không có quân hậu nào tấn công quân hậu nào.  
insafety( [ ] ).

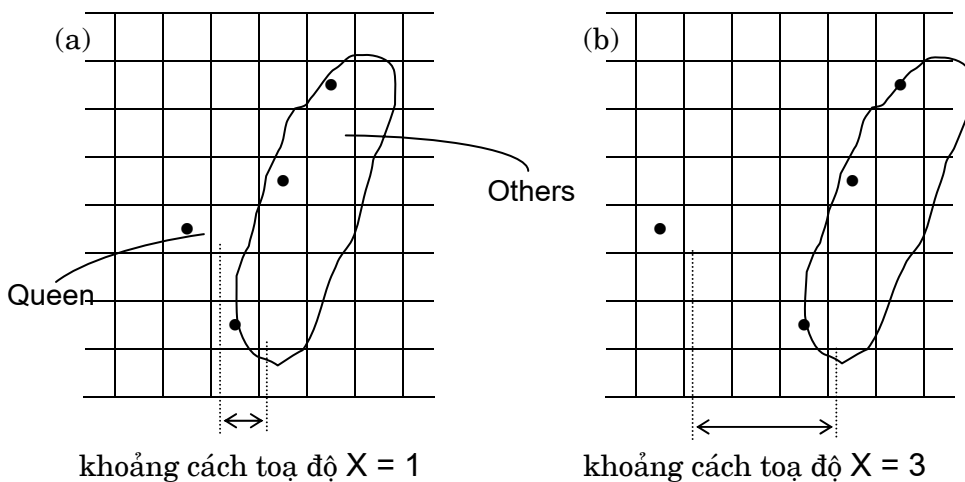
2. Nếu danh sách  $S$  khác rỗng, khi đó  $S$  có dạng  $[ \text{Queen} \mid \text{Others} ]$ . Ta thấy  $S$  là lời giải nếu các quân hậu trong  $\text{Others}$  là ở trạng thái an toàn và quân hậu  $\text{Queen}$  không thể tấn công quân hậu nào trong  $\text{Others}$ .

Từ đó ta có :

`insafety( [ ] ).`

`insafety( [ Queen | Others ] ) :-`  
     `insafety( Others ),`  
     `noattack( Queen, Others ).`

Trong định nghĩa `insafety`, quan hệ `noattack` tỏ ra tinh tế hơn so với cũng cùng quan hệ này trong chương trình 1 trên đây. Khó khăn nằm ở chỗ vị trí của một quân hậu chỉ được xác định bởi các tọa độ  $Y$ , mà vắng mặt tọa độ  $X$ . Để định nghĩa quan hệ `noattack`, ta tìm cách khái quát vấn đề như minh hoạ ở hình dưới đây.



Hình II.8. Khoảng cách giữa tọa độ  $X$  của  $\text{Queen}$  và tọa độ  $X$  của  $\text{Others}$  là 1.

(b) Khoảng cách giữa tọa độ  $X$  của  $\text{Queen}$  và tọa độ  $X$  của  $\text{Others}$  là 3.

Ta thấy rằng sử dụng đích :

`noattack( Queen, Others )`

là để minh chứng rằng quân hậu  $\text{Queen}$  chỉ có thể tấn công các quân hậu trong danh sách  $\text{Others}$  khi tọa độ  $X$  của  $\text{Queen}$  cách tọa độ  $X$  của  $\text{Others}$  ít nhất là 1.

Để thực hiện điều này, ta thêm một đối thứ ba là  $\text{XDist}$  (khoảng cách theo tọa độ  $X$  giữa  $\text{Queen}$  và  $\text{Others}$ ) vào `noattack` :

noattack( Queen, Others, XDist )

Vì vậy, ta phải thay đổi lại đích noattack trong insafety như sau :

```
insafety( [ Queen | Others ] ) :-
    insafety( Others ),
    noattack( Queen, Others, XDist ).
```

Để định nghĩa noattack, cần phân biệt hai trường hợp của danh sách Others :

1. Nếu Others rỗng, khi đó không có quân hậu nào tấn công quân hậu nào.  
noattack( \_ , [ ], \_ ).
2. Nếu danh sách Others khác rỗng, khi đó Queen không thể tấn công quân hậu là phần tử đầu của danh sách Others (khoảng cách giữa toạ độ X của Queen và toạ độ X của phần tử đầu này là 1), cũng như không thể tấn công một quân hậu nào trong phần danh sách còn lại của Others, với một khoảng cách là XDist + 1.

Từ đó ta có :

```
noattack( Y, [ Y1 | YList ], XDist ) :-
    Y1 - Y == XDist,
    Y - Y1 == XDist,
    Dist1 is XDist + 1,
    noattack( Y, YList, Dist1 ).
```

Tất cả những lập luận và quan hệ vừa định nghĩa trên đây cho ta chương trình lời giải thứ hai cho bài toán tám quân hậu như sau :

```
% chương trình thứ hai giải bài toán tám quân hậu
% The problem of the eight queens - Program 2
% -----
solution( Queens ) :-
    permutation( [ 1, 2, 3, 4, 5, 6, 7, 8 ], Queens ),
    insafety( Queens ).
permutation( [ ], [ ] ).
permutation( [ Head | Tail ], PermList ) :-
    permutation( Tail, PermTail ),
    remove( Head, PermList, PermTail ).
remove( X, [ X | L ], L ).
remove( X, [ Y | L ], [ Y | L1 ] ) :-
    remove( X, L, L1 ).
insafety( [ ] ).
```

```

insafety( [ Queen | Others ] ) :-
    insafety( Others ),
    noattack( Queen, Others, 1 ).
noattack( _ , [ ], _ ).
noattack( Y, [ Y1 | YList ], XDist ) :-
    Y1 - Y \= XDist,
    Y - Y1 \= XDist,
    Dist1 is XDist + 1,
    noattack( Y, YList, Dist1 ).

```

Sau khi yêu cầu, Prolog đưa ra các lời giải như sau :

```

?- solution( S ).
S = [5, 2, 6, 1, 7, 4, 8, 3] ;
S = [6, 3, 5, 7, 1, 4, 2, 8] ;
S = [6, 4, 7, 1, 3, 5, 2, 8] ;
S = [3, 6, 2, 7, 5, 1, 8, 4] ;
S = [6, 3, 1, 7, 5, 8, 2, 4] ;
S = [6, 2, 7, 1, 3, 5, 8, 4] ;
S = [6, 4, 7, 1, 8, 2, 5, 3] ;
...
Yes

```

### II.5.3. Sử dụng tọa độ theo hàng, cột và các đường chéo

Trong chương trình thứ ba, ta đưa ra lập luận như sau :

Cần phải đặt mỗi quân hậu lên một ô, nghĩa là trên một hàng, một cột, một đường chéo nghịch (từ dưới lên) và một đường chéo thuận (từ trên xuống). Để mọi quân hậu không thể ăn được lẫn nhau, chúng phải được đặt mỗi quân trên một hàng, một cột, một đường chéo nghịch và một đường chéo thuận phân biệt. Như vậy, ta dự kiến một hệ thống tọa độ biểu diễn các quân hậu như sau :

x    cột  
 y    hàng  
 u    đường chéo nghịch  
 v    đường chéo thuận

Các tọa độ không hoàn toàn độc lập với nhau : với x và y đã cho, ta có thể tính được u và v :

$u = x - y$   
 $v = x + y$



Sau đây là bốn miền giá trị tương ứng với bốn tọa độ  $X, Y, U, V$  :

$Dx = [ 1, 2, 3, 4, 5, 6, 7, 8 ]$

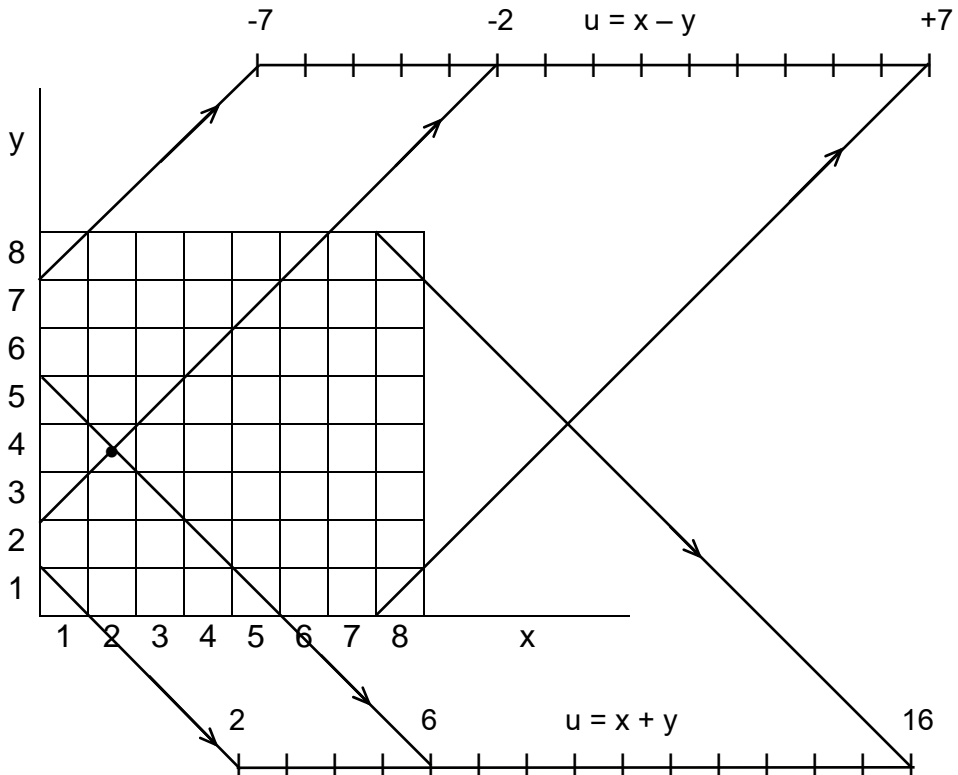
$Dy = [ 1, 2, 3, 4, 5, 6, 7, 8 ]$

$Du = [ -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7 ]$

$Dv = [ 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16 ]$

Bài toán tám quân hậu bây giờ được phát biểu lại như sau : hãy chọn ra tám bộ bốn  $(X, Y, U, V)$ , sao cho  $X \in Dx, Y \in Dy, U \in Du, V \in Dv$  và không bao giờ sử dụng hai lần cùng một phần tử trong mỗi miền giá trị  $Dx, Dy, Du$  và  $Dv$ .

Như vậy,  $U$  và  $V$  được sinh ra từ việc lựa chọn  $X$  và  $Y$ .



Hình II.9. Quan hệ giữa cột, hàng, đường chéo nghịch và đường chéo thuận  
Ô có đánh dấu ( ) trong hình có tọa độ  $x = 2, y = 4, u = 2 - 4 = -2, v = 2 + 4 = 6$ .

Lời giải đại khái có dạng như sau : cho trước bốn miền giá trị, hãy chọn một vị trí cho quân hậu đầu tiên, rồi xoá các tọa độ của chúng trong miền giá trị, sau đó sử dụng các miền giá trị mới này để đặt các quân hậu khác tiếp theo. Các vị trí trên bàn cờ cũng được biểu diễn bởi một danh sách các

toạ độ trên trục Y. Chương trình Prolog giải bài toán tám quân hậu sẽ sử dụng quan hệ :

`sol ( ListY, Dx, Dy, Du, Dv )`

cho phép ràng buộc các toạ độ của các quân hậu (trong ListY), xuất phát từ nguyên lý là các quân hậu nằm trên các cột liên tiếp nhau lấy từ Dx. Các toạ độ Y, U và V được lấy từ Dy, Du và Dv tương ứng.

Lời giải cuối cùng của bài toán tám quân hậu là mệnh đề :

?- `solution( S )`

cho phép gọi sol với danh sách các tham đối đầy đủ. Chương trình như sau :

```
% chương trình thứ ba giải bài toán tám quân hậu
% The problem of the eight queens - Program 3
% -----
solution( ListY ) :-
    sol( ListY ),
        [ 1, 2, 3, 4, 5, 6, 7, 8 ],
        [ 1, 2, 3, 4, 5, 6, 7, 8 ],
        [ -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7 ],
        [ 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16 ] ) .

sol( [ ], [ ], Dy, Du, Dv ).
sol( [ Y | ListY ], [ X | Dx1 ], Dy, Du, Dv ) :-
    del( Y, Dy, Dy1 ),
    U is X - Y,
    del( U, Du, Du1 ),
    V is X + Y,
    del( V, Dv, Dv1 ),
    sol ( ListY, Dx1, Dy1, Du1, Dv1 ).

del( A, [ A | List ], List ) .
del( A, [ B | List ], [ B | List1 ] ) :-
    del( A, List , List1 ).
```

Sau khi yêu cầu, Prolog đưa ra các lời giải như sau :

?- `solution( S )`.

`S = [ 1, 5, 8, 6, 3, 7, 2, 4 ] ;`

`S = [ 1, 6, 8, 3, 7, 4, 2, 5 ] ;`

`S = [ 1, 7, 4, 6, 8, 2, 5, 3 ] ;`

`S = [ 1, 7, 5, 8, 2, 4, 6, 3 ]`

...

Yes

Thủ tục sol vừa xây dựng trên đây có tính tổng quát vì có thể dùng để giải quyết bài toán cho N quân hậu bất kỳ (trên bàn cờ  $N \times N$ ). Sự khác nhau là ở chỗ miền giá trị Dx, Dy, ... thay đổi tùy theo N.

Để tạo sinh các miền giá trị này một cách tự động, ta định nghĩa thủ tục :

gen( N1, N2, List ).

để tạo ra một danh sách các số nguyên giữa N1 và N2 :

List = [ N1, N1 + 1, N1 + 2, ..., N2 - 1, N2 ]

Thân thủ tục như sau :

gen( N, N, [ N ] ).

gen( N1, N2, [ N1 | List ] ) :-

N1 < N2,

M is N1 + 1,

gen( M, N2, List ).

Bây giờ ta thay đổi quan hệ solution như sau :

solution( N, S ) :-

gen( 1, N, Dxy ),

Nu1 is 1 - N,

Nu2 is N - 1,

gen( Nu1, Nu2, Du ),

Nv2 is N + N,

gen( 2, Nv2, Dv ),

sol( S, Dxy, Dxy, Du, Dv ).

Giả sử cần giải bài toán với 12 quân hậu, ta có lời gọi như sau :

?- solution( 12, S ).

S = [ 1, 3, 5, 8, 10, 12, 6, 11, 2, 7, 9, 4]

...

Yes

#### II.5.4. Kết luận

Ví dụ bài toán tám quân hậu trên đây minh họa cách giải quyết một bài toán trong Prolog theo nhiều lời giải khác nhau, mỗi lời giải sử dụng một phương pháp biểu diễn cấu trúc dữ liệu. Mỗi cách biểu diễn dữ liệu đều có những đặc trưng riêng, như tiết kiệm bộ nhớ, hay biểu diễn tường minh, hay biểu diễn phức hợp các thành phần của đối tượng cần xử lý. Cách biểu diễn

dữ liệu nhằm tiết kiệm bộ nhớ có bất lợi ở chỗ là thường xuyên phải tính đi tính lại một số giá trị dữ liệu trước khi có thể sử dụng chúng.

Trong ba lời giải trên đây, lời giải thứ ba minh họa rõ nét hơn cả về cách xây dựng các cấu trúc dữ liệu xuất phát từ một tập hợp các phần tử đã cho có nhiều ràng buộc. Hai chương trình đầu xây dựng tất cả các hoán vị có thể rồi lần lượt kiểm tra có phải hoán vị đang xét là một lời giải không để loại bỏ những hoán vị không tốt trước khi xây dựng chúng một cách đầy đủ. Việc xác định các hoán vị gây ra tốn thời gian do phải thực hiện nhiều lần các phép tính số học. Chương trình thứ ba tránh được điều này nhờ cách biểu diễn bàn cờ hợp lý.

### II.5.5. Bộ diễn dịch Prolog

Xây dựng bộ diễn dịch Prolog bằng chính ngôn ngữ Prolog, được gọi là bộ siêu diễn dịch vani (vanilla meta-interpreter).

```
solve(true).
solve((A, B)) :-
    solve(A),
    solve(B).
solve(A) :-
    clause(A, B),
    solve(B).
```

Mệnh đề `clause(A, B)` Prolog cho phép kiểm tra nếu `A` là một sự kiện hay vế trái (LHS) của một luật nào đó trong cơ sở dữ liệu (chương trình Prolog), `B` là thân hay vế phải của luật đó (nếu `A` là một sự kiện thì `B = true`). Ví dụ :

```
?- clause(ins(X, [H|T], [ X,H|T ]), X @=< H).
```

```
X = _G420
```

```
H = _G417
```

```
T = _G418
```

```
Yes
```

Cách gọi bộ siêu diễn dịch vani :

```
?- solve(PrologGoal).
```

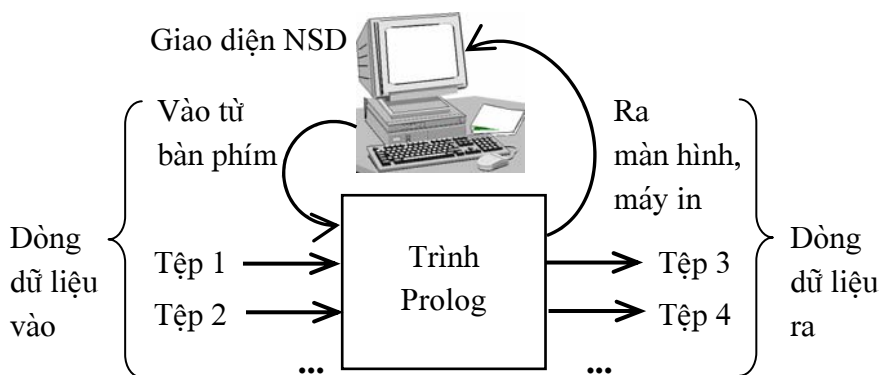
### III. Quá trình vào-ra và làm việc với tệp

#### III.1. Khái niệm

Cho đến lúc này, ta mới làm việc với Prolog qua chế độ tương tác : NSD đặt câu hỏi là dãy các đích dựa trên chương trình đã biên dịch (là một CSDL chứa luật và sự kiện), Prolog trả lời cho biết các đích được thoả mãn (Yes) hay không thoả mãn (No), đồng thời tùy theo yêu cầu mà đưa ra kết quả dưới dạng ràng buộc giá trị cho các biến ( $X = \dots$ ). Phương pháp này đơn giản, đủ để trao đổi thông tin, tuy nhiên người ta vẫn luôn luôn tìm cách mở rộng khả năng trao đổi này. Người ta cần giải quyết những vấn đề sau :

- Vào dữ liệu cho chương trình dưới các dạng khác câu hỏi, chẳng hạn các câu trong ngôn ngữ tự nhiên (tiếng Anh, tiếng Pháp...).
- Đưa ra thông tin dưới bất kỳ dạng thức nào mong muốn.
- Làm việc được với các tệp (file) không chỉ thuần túy màn hình, bàn phím.

Hầu hết các phiên bản Prolog đều có những vị từ thích hợp giải quyết được những vấn đề nêu trên. Giống như các ngôn ngữ lập trình khác, Prolog xem các thiết bị vào-ra chuẩn (bàn phím, màn hình) là các tệp đặc biệt. Quá trình vào-ra trên các thiết bị này và trên các thiết bị lưu trữ ngoài được xem là quá trình làm việc với các tệp. Hình dưới đây mô tả cách Prolog làm việc với các tệp.



Hình III.1. Liên lạc giữa một trình Prolog và nhiều tệp .

Trình Prolog có thể đọc dữ liệu vào từ nhiều tệp, được gọi là *dòng dữ liệu vào* (input streams), sau khi tính toán, có thể ghi lên nhiều tệp, được gọi là

*dòng dữ liệu ra* (output streams). Dữ liệu đến từ giao diện NSD (bàn phím), rồi kết quả gửi ra màn hình, cũng được xử lý như là những dòng dữ liệu vào ra khác. Đây là những *tệp giả* (pseudo-file) được đặt tên là user (người sử dụng). Các tệp chứa chương trình, hay dữ liệu Prolog được NSD lựa chọn đặt tên tự do (miễn là khác user) trong khuôn khổ của hệ điều hành.

Khi thực hiện một trình Prolog, tại mỗi thời điểm, chỉ có hai tệp hoạt động là tệp đang được đọc, được gọi là *dòng vào hiện hành* (active input streams), và tệp đang được ghi, được gọi là *dòng ra hiện hành* (active output streams).

Lúc mới chạy chương trình, dòng vào hiện hành là bàn phím và dòng ra hiện hành là màn hình (hoặc máy in) tương ứng với chế độ vào ra chuẩn user.

## III.2. Làm việc với các tệp

### III.2.1. Đọc và ghi lên tệp

Một số vị từ xử lý đọc và ghi lên tệp của Prolog như sau :

| <i>Tên vị từ</i> | <i>Ý nghĩa</i>                                                                                                                                                                                                     |
|------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| see(File)        | Mở tệp File để đọc dữ liệu và xác định File là dòng vào hiện hành. Tệp File phải có từ trước, nếu không, Prolog báo lỗi tệp File không tồn tại.                                                                    |
| see(user)        | Dòng vào hiện hành là bàn phím (chế độ chuẩn).                                                                                                                                                                     |
| seeing(File)     | Hợp nhất tệp File với tệp vào hiện hành.                                                                                                                                                                           |
| tell(File)       | Mở tệp File để ghi dữ liệu lên và xác định File là dòng ra hiện hành. Nếu tệp File chưa được tạo ra trước đó, thì tệp File sẽ được tạo ra. Nếu tệp File đã tồn tại, nội dung tệp File sẽ bị xóa để ghi lại từ đầu. |
| tell(user)       | Dòng ra hiện hành là màn hình (chế độ chuẩn).                                                                                                                                                                      |
| telling(File)    | Hợp nhất tệp File với tệp ra hiện hành.                                                                                                                                                                            |
| told             | Đóng tệp đang ghi lên hiện hành. Dòng vào trở lại chế độ vào chuẩn user.                                                                                                                                           |
| seen             | Đóng tệp đang đọc hiện hành. Dòng ra trở lại chế độ ra chuẩn user.                                                                                                                                                 |

|             |                                                                                                                                                                                                                                                                                                                                                                                             |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| read(Term)  | Đọc từ dòng vào hiện hành một giá trị để khớp với hạng Term. Nếu Term là biến thì được lấy giá trị này và vị từ thoả mãn. Nếu không thể số khớp, vị từ trả về thất bại mà không tiến hành quay lui. Mỗi hạng trong tệp phải kết thúc bởi một dấu chấm và một dấu cách (space) hoặc dấu Enter. Khi thực hiện read mà đang ở vị trí cuối tệp, Term sẽ nhận giá trị <code>end_of_file</code> . |
| write(Term) | Ghi lên tệp hiện hành giá trị của hạng Term. Nếu Term là biến thì giá trị này được đưa ra theo kiểu của Prolog. Các kiểu giá trị khác nhau đều có thể đưa ra bởi write.                                                                                                                                                                                                                     |

Ví dụ III.1 :

NSD định hướng dòng vào là tệp `myexp1.pl` :

?- `see('myexp1.pl').` % Bắt đầu đọc tệp `myexp1.pl`.

Yes

Hoặc :

?- `see('C:/My Documents/Gt-Prolog/Example/myexp1.pl').`

Yes

Đích `see(F)` luôn luôn được thoả mãn, trừ trường hợp xảy ra sai sót đối với các tệp dữ liệu. Chú ý tên thư mục và đường dẫn được viết theo kiểu Unix và được đặt trong cặp dấu nháy đơn. Sau khi làm việc trên tệp `myexp1.pl`, lệnh `seen` cho phép trở về chế độ chuẩn.

?- `seen.`

Yes

Ví dụ III.2 :

Dùng `read` để đọc dữ liệu vào bất kỳ từ bàn phím :

?- `read(N).`

| 100.

N = 100

Yes

?- `read('Your name ?').`

| asimo.

No

?- `read('Your name ?').`

```

| 'Your name ?'.
Yes
?- read(asimo).
| Your_name.
Yes
% Đọc và ghi các hạng
?- read(X).
| father(tom, mary).
X = father(tom, mary)
Yes
T = father(tom, mary), write(T).
father(tom, mary)
T = father(tom, mary)
Yes

```

### Ví dụ III.3

Đọc nội dung trong tệp 'myex1.pl', sau đó quay lại chế độ vào ra chuẩn.

```

?- see('myex1.pl'), read(T), see(user).
T = del(_G467, [_G467|_G468], _G468)
Yes

```

Trong dãy đích trên, đích `read(T)` đọc được sự kiện  $(X, [X | L], L)$ . là nội dung dòng đầu tiên của tệp có nghĩa, sau khi bỏ qua các dòng chú thích (nếu có).

Ta cũng có thể hướng dòng ra lên tệp bằng cách sử dụng đích :

```

?- tell('myex2.pl').

```

Dãy đích sau đây gửi thông tin là sự kiện `parent(tom, bob)`. lên tệp `myex2.pl`, sau đó quay lại chế độ vào ra chuẩn :

```

tell(myex2.txt'), write('parent(tom, bob).'), tell(user).

```

Các tệp chỉ có thể truy cập tuần tự. Prolog ghi nhớ vị trí hiện hành của dòng vào để đọc dữ liệu. Mỗi lần đọc hết một đối tượng (luật, hay sự kiện), Prolog dời đầu đọc đến vị trí đầu đối tượng tiếp theo. Khi đọc đến hết tệp, Prolog đưa ra thông báo hết tệp :

```

?- see('exp.txt'), read(T), see(user).
T = end_of_file
Yes

```



Ví dụ III.4 :

Dùng write để đưa dữ liệu bất kỳ ra màn hình :

?- write(asimo).

asimo

Yes

Cách ghi lên tệp cũng theo cơ chế tương tự, dữ liệu được ghi liên tiếp bắt đầu từ vị trí cuối cùng của đối tượng. Prolog không thể quay lui hay ghi đè lên phần đã ghi trước đó.

Prolog chỉ làm việc với các tệp dạng văn bản (text files), nghĩa là chỉ vào ra với các chữ cái chữ số và ký tự điều khiển ASCII.

### III.2.2. Một số ví dụ đọc và ghi lên tệp

Một số ví dụ từ đọc và ghi khác của Prolog như sau :

| Tên vị từ               | Ý nghĩa                                                                |
|-------------------------|------------------------------------------------------------------------|
| write(File, Term)       | Ghi lên tệp File giá trị hạng Term.                                    |
| writeln(Term)           | Ghi lên dòng ra hiện hành giá trị hạng Term kèm dấu nhảy đơn (quotes). |
| writeln(File, Term)     | Ghi lên tệp File giá trị hạng Term kèm dấu nhảy đơn (quotes).          |
| print(Term)             | In ra dòng ra hiện hành giá trị hạng Term.                             |
| print(File, Term)       | In ra tệp File giá trị hạng Term.                                      |
| read(File, Term)        | Đọc từ tệp File hiện hành cho Term.                                    |
| read_clause(Term)       | Tương tự to read/1. Đọc một mệnh đề từ dòng vào hiện hành.             |
| read_clause(File, Term) | Đọc một mệnh đề từ tệp File.                                           |
| nl                      | Nhảy qua dòng mới (newline).                                           |
| tab(N)                  | In ra N dấu khoảng trống (space)                                       |
| tab(File, N)            | In ra N dấu khoảng trống trên tệp File                                 |

Ví dụ III.5 :

?- nl.        % Qua dòng mới

Yes

?- tab(5), write(\*), nl.

\*

Yes

đưa ra màn hình 5 dấu cách rồi đến một dấu \* và qua dòng.

Ví dụ III.6 :

Viết thủ tục tính lũy thừa 3 của một số :

cube( N, C) :-

C is N \* N\* N.

Giả sử ta muốn tính nhiều lần cube, khi đó ta phải viết nhiều lần đích :

?- cube( 2, X ).

X=8

Yes

?- cube( 5, Y ).

Y 125

?- cube( 12, Z ).

Z = 1728

Yes

Để chỉ cần sử dụng một đích mà có thể tính nhiều lần cube, ta cần sửa lại chương trình như sau :

cube :-

read( X ),

compute( X ).

compute( stop ) :- !.

compute( N ) :-

C is N \*N\* N,

write( C ),

cube.

Nghĩa thủ tục của chương trình cube như sau : để tìm lũy thừa 3, trước tiên đọc X, sau đó thực hiện tính toán với X và in ra kết quả. Nếu X có giá trị là stop, ngừng ngay, nếu không, thực hiện tính toán một cách đệ quy. Chú ý khi nhập dữ liệu cho vị từ read, cần kết thúc bởi một dấu chấm :

?- cube.

|: 3.

27

|: 10.

1000

|: 18.

5832

|: stop.

Yes

Ta có thể tiếp tục thay đổi chương trình. Một cách trực giác, nếu viết lại cube mà không sử dụng compute như sau là sai :

```
cube :-
    read( stop), !.
cube :-
    read( N),
    C is N *N * N,
    write( C),
    cube.
```

bởi vì, giả sử NSD gõ vào 3, đích read( stop) thất bại, nhát cắt bỏ qua dữ liệu này và do vậy, cube(3) không được tính. Lệnh read( N) tiếp theo sẽ yêu cầu NSD vào tiếp dữ liệu cho N. Nếu N là số, việc tính toán thành công, ngược lại, nếu N là stop, Prolog sẽ thực hiện tính toán trên các dữ liệu phi số stop :

```
?- cube1.
|: 3.                % Prolog bỏ qua, không tính
|: 9.
729                  % Prolog tính ra kết quả cho N = 9
|: 4.                % Prolog bỏ qua, không tính
|: stop.             % Prolog báo lỗi
ERROR: Arithmetic: `stop/0' is not a function
^ Exception: (9) _L143 is stop*stop*stop ? creep
```

Thông thường các chương trình khi thực hiện cần sự tương tác giữa NSD và hệ thống. NSD cần được biết kiểu và giá trị dữ liệu chương trình yêu cầu nhập vào. Muốn vậy, chương trình cần đưa ra dòng yêu cầu hay lời nhắc (prompt). Hàm cube được viết lại như sau :

```
cube :-
    write('Please enter a number: '),
    read( X ),
    compute( X ).
compute( stop ) :- !.
compute( N ) :-
    C is N *N* N,
    write('The cube of '), write(N),
    write(' is '), write( C), nl,
    cube.
cube.
Please enter a number: 3.
The cube of 3 is 27
Please enter a number: stop.
```

Yes

### Ví dụ III.7

Ta xây dựng thủ tục `displaylist` sau đây để in ra các phần tử của danh sách :

```
displaylist( [ ] ).
displaylist( [X | L ] ) :-
    write( X ), nl,
    displaylist( L ).
?- displaylist( [[a, b, c], [d, e, f], [g, h, i]]).
[a, b, c]
[d, e, f]
[g, h, i]
Yes
```

Ta thấy trong trường hợp các phần tử của một danh sách lại là những danh sách như trên thì tốt hơn cả là in chúng ra trên cùng hàng :

```
displaylist( [ ] ).
displaylist( [X | L ] ) :-
    write( X ), tab( 1),
    displaylist( L ), nl.
displaylist( [[a, b, c], [d, e, f], [g, h, i]]).
[a, b, c] [d, e, f] [g, h, i]
Yes
```

Thủ tục dưới đây in ra các phần tử kiểu danh sách phẳng trên cùng hàng :

```
displaylist2( [ ] ).
displaylist2( [ L | L1 ] ) :-
    inline( u),
    displaylist2( L1 ), nl.
inline( [ ] ).
inline( [ X | L ] ) :-
    write( X ), tab( 1),
    inline( L ).
?- displaylist2( [[a, b, c], [d, e, f], [g, h, i]]).
a b c d e f g h i
Yes
```

Ví dụ dưới đây in ra danh sách các số nguyên dưới dạng một đồ thị gồm các dòng kẻ là các dấu sao (hoa thị) \* :

```

barres( [ N | L] ) :-
    asterisk(N), nl,
    barres(L).
asterisk( N ) :-
    N > 0,
    write( *),
    N1 is N - 1,
    asterisk( N1).
asterisk( N ) :-
    N =< 0.
?- barres([3, 4, 6, 5, 9]).
* * *
* * * *
* * * * *
* * * * *
* * * * * *
* * * * * * *
No

```

Ví dụ III.8 :

Đọc nội dung một tệp vào danh sách các số nguyên :

```

readmyfile( File, List ) :-
    see( File),
    readlist( List),
    seen,
    !.
readlist( [X | L ] ) :-
    get0(X),
    X =\= -1,
    !,
    read_list( L).
readlist( [ ] ).

```

### III.2.3. Nạp chương trình Prolog vào bộ nhớ

Các chương trình Prolog thường được lưu cất trong các tệp có tên hậu tố (hay phần mở rộng của tên) là « .pl ». Để nạp chương trình (load) vào bộ nhớ và biên dịch (compile, Prolog sử dụng vị từ :

```

?- consult(file_name).

```

trong đó, file\_name là một nguyên tử.

Ví dụ III.9 :

Đích sau đây nạp và biên dịch chương trình nằm trong tệp myexp.pl :

```

?- consult('myexp.pl').

```

Yes

Prolog cho phép viết gọn trong một danh sách như sau :

?- ['myexp.pl' ].

Để nạp và biên dịch đồng thời nhiều tệp chương trình khác nhau, có thể liệt kê trong một danh sách như sau :

?- ['file1.pl', 'file2.pl'].

Sau khi các chương trình đã được nạp vào bộ nhớ, NSD bắt đầu thực hiện chương trình. NSD có thể xem nội dung toàn bộ chương trình nhờ vị từ :

?- listing.

hoặc xem một mệnh đề nào đó :

?- listing(displaylist).

displaylist( [ ] ).

displaylist( [X | L ] ) :-

    write( X ),

    tab( 1 ),

    displaylist( L ), nl.

Yes

### III.3. Ứng dụng chế độ làm việc với các tệp

#### III.3.1. Định dạng các hạng

Giả sử một bản ghi cơ sở dữ liệu, là một sự kiện có dạng cấu trúc hàm tử của Prolog, có nội dung như sau :

```
family(individual(tom, smith, date(7, may, 1960), work(microsoft, 30000)),
       individual( ann, smith, date(9, avril, 1962), inactive),
       [ individual( roza, smith, date(16, june, 1991), inactive),
         individual( eric, smith, date(23, march, 1993), inactive) ] ).
```

Ta cần in ra nội dung bản ghi sử dụng vị từ write(F) theo quy cách như sau :

parents

    tom smith, birth day may 7,1960, work microsoft, salary 30000

    ann smith, birth day avril 9, 1962, out of work

children

    roza smith, birth day june 16, 1991, out of work

    eric smith, birth day march 23, 1993, out of work

Ta xây dựng thủ tục writefamily( F) như sau :

```

writefamily(family(Husband, Wife, Children)) :-
    nl, write(parents),nl, nl,
    writeindividual(Husband) ,nl,
    writeindividual(Wife), nl, nl,
    write(children), nl, nl,
    writeindividual(Children).

writeindividual(individual(Firstname, Name, date(D, M, Y), Work)) :-
    tab(4), write(Firstname),
    tab(1), write(Name),
    write(' , birth day '), write(M), tab(1),
    write(D), tab(1), write(' , '), write(Y), write(' , '),
    writework(Work).

writeindividual([ ]).

writeindividual([ P | L ]):-
    writeindividual( P), nl,
    writeindividual( L).

writework(inactive):-
    write('out of work').

writework(work(Soc, Sal)):-
    write(' work '), write(Soc),
    write(' , salaire '), write(Sal).

```

Thực hiện đích  $X = \dots$ , `writefamily(X)`, ta nhận được kết quả như sau

```

?- X = family(individual( tom, smith, date(7, may, 1960), work(microsoft,
30000) ),individual( ann, smith, date(9, avril, 1962), inactive),[individual(
roza, smith, date(16, june, 1991), inactive),individual( eric, smith, date(23,
march, 1993), inactive) ] ), writefamily(X).

```

parents

```

tom smith, birth day may 7 , 1960, work microsoft, salaire 30000
ann smith, birth day avril 9 , 1962, out of work

```

children

```

roza smith, birth day june 16 , 1991, out of work
eric smith, birth day march 23 , 1993, out of work

```

```

X = family(individual(tom, smith, date(7, may, 1960), work(microsoft,
30000)), individual(ann, smith, date(9, avril, 1962), inactive), [individual(roza,
smith, date(16, june, 1991), inactive), individual(eric, smith, date(23, march,
1993), inactive)])

```

Yes

### III.3.2. Sử dụng tệp xử lý các hạng

Để đọc dữ liệu trên tệp, người ta sử dụng dãy đích sau :

$\dots$ , `see( F)`, `fileprocess`, `see( user)`,  $\dots$

Thủ tục `fileprocess` đọc và xử lý lần lượt từng hạng của `F` cho đến khi đọc hết tệp. Mô hình thủ tục như sau :

```
filetreat :-
    read( Term),
    treat( Term).
treat( end_of_file ) :- !.      % Kết thúc tệp
    treat( Term ) :-
        treatment( Term),      % Xử lý hạng hiện hành
        filetreat.             % Xử lý phần còn lại của tệp
```

Trong thủ tục trên, `treatment( Terme)` thể hiện mọi thao tác có thể tác động lên hạng. Chẳng hạn thủ tục dưới đây liệt kê từng hạng của tệp kể từ dòng thứ `N` trở đi cho đến hết tệp, kèm theo thứ tự có mặt của hạng đó trong tệp :

```
viewfile( N ) :-
    read( Term),
    viewterm( Term, N).
viewterm( end_of_file, _ ) :- !.
viewterm( Term, N ) :-
    write( N), tab( 2),
    write( Term), nl,
    N1 is N + 1,
    viewfile( N1).
?- see('exp.txt'), viewfile(1), see( user), seen.
1 parent(pam, bob)
2 parent(tom, bob)
3 parent(tom, liz)
4 parent(bob, ann)
5 parent(bob, pat)
...
Yes
```

Sau đây là một mô hình khác để xử lý tệp. Giả sử `file1` là tệp dữ liệu nguồn chứa các hạng có dạng :

```
object( NoObject, Description, Price, FurnisherName).
```

Mỗi hạng mô tả một phần tử của danh sách các đối tượng. Giả sử rằng tệp cần xây dựng `file2` chứa các đối tượng do cùng một nhà cung cấp cấp hàng. Trong tệp này, tên nhà cung cấp được viết một lần ở đầu tệp, mà



không xuất hiện trong các đối tượng, có dạng object( No, Desc, Price). Thủ tục tạo tệp như sau :

```
createfile(Furnisher) :-
    write(Furnisher), write( '.'), nl,
    creatremaining(Furnisher).
creatremaining( Fournisseur) :-
    read( Objet),
    treat( Objet, Furnisher).
treat( end_of_file) :- !.
treat(object( No, Desc, Price, Furn), Furn) :-
    write( object( No, Desc, Price) ),
    write( '.'), nl,
    creatremaining(Furn).
treat( _ , Furnisher) :-
    creatremaining(Furnisher).
```

Giả sử file1 là tệp

```
see(' file1.txt'),tell(' file2.txt'), createfile(suzuki), seen, see(user), told,
tell(user).
```

*Ví dụ III.10 :*

Sao chép nội dung một tệp lên một tệp khác :

```
copie :-
    repeat,
    read(X),
    mywrite(X),
    X == end_of_file, !.
mywrite( end_of_file).
mywrite( X) :-
    write( X), write( '.'), nl.
```

Đích sau cho phép copy từ tệp nguồn f1.txt vào tệp đích f2.txt :

```
?- tell('f2.txt'), see('f1.txt'), copie, seen, told.
Yes
```

Trong thủ tục copie có sử dụng vị từ repeat. Vị từ repeat luôn luôn thành công, tạo ra một vòng lặp vô hạn. Vị từ repeat được định nghĩa như sau :

```
repeat.
repeat :- repeat.
```

### III.3.3. Thao tác trên các ký tự

Một số vị từ xử lý ký tự của Prolog như sau :

| Tên vị từ            | Ý nghĩa                                                                                            |
|----------------------|----------------------------------------------------------------------------------------------------|
| put(Char)            | Đưa Char ra dòng ra hiện hành, Char hoặc là một giá trị nguyên trong khoảng 0..255, hoặc một ký tự |
| put(File, Char)      | Đưa Char ra tệp File                                                                               |
| get_char(Char)       | Đọc từ tệp File và hợp nhất Char với ký tự tiếp theo.                                              |
| get_char(File, Char) | Hợp nhất Char với ký tự tiếp theo trong tệp File.                                                  |
| get0(Char)           | Đọc ký tự tiếp theo                                                                                |
| get0(File, Char)     | Đọc ký tự tiếp theo trong tệp File.                                                                |
| get(-Char)           | Đọc ký tự khác khoảng trống từ dòng vào và hợp nhất với Char.                                      |
| get(File, Char)      | Đọc ký tự khác khoảng trống tiếp theo trong tệp File.                                              |
| skip(Char)           | Đọc vào và bỏ qua các ký tự đọc được cho đến khi gặp đúng ký tự khớp được với Char.                |
| skip(File, Char)     | Đọc vào từ tệp File và bỏ qua các ký tự đọc được cho đến khi gặp đúng ký tự khớp được với Char.    |

Ví dụ III.11 :

% Đưa ra liên tiếp các ký tự A, B và C có mã ASCII lần lượt là 65, 66, 67

?- put( 65), put( 66), put( 67).

ABC

yes

% Đọc và ghi các ký tự

?- get0(X).

|: a        % Gõ vào một ký tự rồi Enter (↵), không gõ dấu chấm

X = 97

Yes.

?- get0(X).

^D

X = -1.

Yes.

Ví dụ III.12 :

Sau đây ta xây dựng thủ tục del\_space đọc vào một câu gồm nhiều từ cách nhau bởi các khoảng trống và trả về đúng câu đó sau khi đã loại bỏ các khoảng trống thừa, chỉ giữ lại một khoảng trống giữa các từ mà thôi.

Thủ tục hoạt động tương tự các thủ tục xử lý tệp, bằng cách đọc lần lượt từng ký tự rồi đưa ra màn hình. Thủ tục sử dụng kỹ thuật nhát cắt để xử lý

tình huống ký tự đọc vào hoặc là một khoảng trống, hoặc là một chữ cái, hoặc là một dấu chấm kết thúc. Sau đây là thủ tục `del_space` :

```
del_space :-
    get0( C),
    put( C),
    follow( C).

follow( 46) :- !.    % 46 là mã ASCII của dấu chấm
follow( 32) :- !,    % 32 là mã ASCII của dấu khoảng trắng
    get( C),          % Bỏ qua các dấu khoảng trắng tiếp theo
    put( C),
    follow( C).

follow( Letter) :-
    del_space.
```

Chạy thử như sau :

```
?- del_space.
|: The robot try to cast the balls to the basket.
The robot try to cast the balls to the basket.
Yes
```

### III.3.4. Thao tác trên các nguyên tử

Prolog có vị từ `name/2` cho phép đặt tương ứng các nguyên tử với các mã ASCII :

```
name( A, L)

Vị từ thoả mãn khi L là danh sách các của các ký tự của A. Ví dụ :
?- name(mic29, [109, 105, 99, 50, 57 ]).
Yes

?- name( aikieutido, L).
L = [ 97, 105, 107, 105, 101, 117, 116, 117, 105 |... ]
Yes

?- name(X, [ 97, 105, 107, 105, 101, 117, 116, 117, 105, 100, 111 ]).
X = aikieutido
Yes
```

Hai chức năng chính của vị từ `name` như sau :

1. Chuyển một nguyên tử thành một danh sách các ký tự (mã ASCII).
2. Tạo một nguyên tử từ một danh sách các ký tự.

*Ví dụ III.13 :*

Xây dựng thủ tục quản lý các cuộc gọi dịch vụ xe taxi chở hành khách nhờ các nguyên tử sau :

Tên các cuộc gọi    call1, call2, ...

Tên các lái xe        chauffeur1, chauffeur2, ...

Tên các xe taxi      taxi1, taxi2, ...

Vị từ taxi( X ) kiểm tra một nguyên tử có biểu diễn đúng một taxi theo cách biểu diễn như trên không :

taxi( T ) :-

    name( T, Tlist),

    name( taxi, L),

    append( L, \_ , Tlist).

Một cách tương tự, ta có thể xây dựng các vị từ chauffer và taxi.

*Ví dụ III.14 :*

Sau đây ta xây dựng thủ tục cho phép tạo ra một nguyên tử bằng cách tổ hợp các ký tự. Thủ tục readsentence( Wordlist) sẽ đọc một câu thuộc ngôn ngữ tự nhiên rồi gán cho Wordlist danh sách các giá trị mã biểu diễn trong của các ký tự trong câu. Tiếp theo, mỗi câu được xem là một danh sách các từ, mỗi từ được chuyển thành một nguyên tử.

readsentence( WordList) :-

    get0( Char),

    readchain( Char, WordList).

readchain( 46,[ ] ) :- !.

% dấu chấm kết thúc câu

readchain( 32, WordList) :-

    readsentence(WordList).

% Bỏ qua các dấu khoảng trống

readchain( L, [ W | WordList ] ) :-

    readletter( L, Letters, Nextchar ), % Đọc các ký tự của từ tiếp theo

    name( W, Letters),

    readchain( Nextchar, WordList).

readletter( 46, [ ], 46) :- !.

% kết thúc từ là một dấu chấm

readletter( 32, [ ], 32) :- !.

% kết thúc từ là một dấu khoảng

trống

readletter( C, [ C | Letters] , Nextchar) :-

    get0( Char),

    readletter( Char, Letters, Nextchar).

Chạy chương trình, ta có các kết quả như sau :

?- readsentence( WordList).

|: The robot ASIMO try to cast the balls to the basket.

WordList = ['The', robot, 'ASIMO', try, to, cast, the, balls, to|...]

Yes

?- readsentence( WordList).

|: " Ai đi trăm tuổi ngàn rừng "       % dấu Enter ↵ sau dấu nhảy kếp

|: .                                       % dấu chấm kết thúc câu

WordList = [ "" Ai', đi, trăm, tuổi, ngàn, 'rừng "\n' ]

Yes

Trong thủ tục, ta đã giả thiết rằng kết thúc câu vào là một dấu chấm và nếu có dấu chấm câu trong câu, thì tùy theo cách xuất hiện mà nó được xem như là một từ hoặc dính vào với từ.

Thủ tục đọc ký tự đầu tiên là Char, rồi chuyển cho thủ tục readchain.

Thủ tục readchain xử lý 3 trường hợp như sau :

- (1) Nếu Char là một dấu chấm, thì quá trình đọc câu vào kết thúc.
- (2) Nếu Char là một khoảng trống, áp dụng thủ tục readsentence cho phần còn lại của câu.
- (3) Nếu Char là một ký tự : trước tiên đọc từ W được bắt đầu bởi ký tự Char, sau đó sử dụng readsentence để đọc phần còn lại của câu và tạo ra danh sách WordList. Kết quả được tích lũy trong [ W | WordList ].

Thủ tục readletter( L, Letters, Nextchar ) đọc các ký tự của một từ, trong đó :

- (1) L là chữ cái hiện hành (đã được đọc) của từ đang đọc.
- (2) Letters là danh sách các chữ cái, bắt đầu bởi L cho đến hết từ.
- (3) Nextchar là ký tự theo sau từ đang đọc, có thể không phải là một chữ cái.

Nhờ cách biểu diễn các từ của câu trong một danh sách, người ta có thể sử dụng Prolog để xử lý ngôn ngữ tự nhiên, như tìm hiểu nghĩa của câu theo một quy ước nào đó, v.v.. thuộc lĩnh vực trí tuệ nhân tạo.

### III.3.5. Một số vị từ xử lý cơ sở dữ liệu

Sau đây là một số vị từ chuẩn cho phép xử lý trên các luật và sự kiện của một cơ sở dữ liệu Prolog.

#### **assert(P)**

Thêm P vào cơ sở dữ liệu. Ví dụ cho cơ sở dữ liệu lúc ban đầu :

personal(tom).

personal(ann).

Sau khi thực hiện đích :

?- assert(personal(bob)).

cơ sở dữ liệu lúc này trở thành :

personal(tom).

personal(ann).

personal(bob).

Do NSD không biết assert đã thêm P vào đầu hay cuối của cơ sở dữ liệu, Prolog cho phép sử dụng hai dạng khác là :

**asserta(P)**      Thêm P vào đầu cơ sở dữ liệu.

**assertz(P)**      Thêm P vào cuối cơ sở dữ liệu.

Sử dụng vị từ :

assert((P :- B, C, D)).

có thể làm thay đổi nội dung một mệnh đề trong chương trình. Tuy nhiên, người ta khuyên không nên sử dụng lệnh này.

#### **retract(P)**

Loại bỏ P khỏi cơ sở dữ liệu. Ví dụ cho cơ sở dữ liệu lúc ban đầu :

personal(tom).

personal(ann).

personal(bob).

Sau khi thực hiện đích :

?- retract(personal(ann)).

cơ sở dữ liệu lúc này chỉ còn :

personal(tom).

personal(bob).

Có thể sử dụng biến trong retract như sau :

?- retract(personal(X)).

X = tom ;

X = bob ;

No

Lúc này cơ sở dữ liệu đã rỗng.

### **abolish(*Term*, *Arity*)**

Loại bỏ tất cả các hạng Term có cấp Arity khỏi cơ sở dữ liệu. Ví dụ :

?- abolish(personal, 2).

Loại bỏ tất cả các hạng Term có cấp Arity=2.

### *Ví dụ III.15*

Xây dựng bộ siêu diễn dịch Prolog trong Prolog, việc xoá một đích được viết lại như sau :

prove(Goal) :- call(Goal).

hoặc :

prove(Goal) :- Goal.

hoặc viết các mệnh đề :

prove(true).

prove((Goal1, Goal2)) :-

    prove(Goal1),

    prove(Goal2).

prove(Goal) :-

    clause(Goal, Body),

    prove(Body).

## Tóm tắt chương 5 :

### Kỹ thuật nhất cắt và phủ định

- Nhất cắt ngăn cản sự quay lui, không những làm tăng hiệu quả chạy chương trình mà còn làm tối ưu tính biểu hiện của ngôn ngữ.
- Để tăng hiệu quả chạy chương trình, người lập trình sử dụng nhất cắt để chỉ ra cho Prolog biết những con đường dẫn đến thất bại.
- Nhất cắt cho phép tạo ra các kết luận loại trừ nhau dạng :  
*If Condition Thì Conclusion\_1 nếu Conclusion\_2*
- Nhất cắt cho phép định nghĩa phép phủ định : not Goal thoả mãn nếu Goal thất bại.
- Prolog có hai đích đặc biệt : true luôn luôn đúng và fail luôn luôn sai.
- Cần thận trọng khi sử dụng kỹ thuật nhất cắt, nhất cắt có thể làm sai lệch sự tương ứng giữa nghĩa khai báo và nghĩa thủ tục của một chương trình.
- Phép phủ định not trong Prolog không hoàn toàn mang ý nghĩa logic, cần chú ý khi sử dụng not.

### Sử dụng các cấu trúc

Các ví dụ đã trình bày trong chương này minh hoạ những đặc trưng rất tiêu biểu của kỹ thuật lập trình Prolog :

- Trong Prolog, tập hợp các sự kiện đủ để biểu diễn một cơ sở dữ liệu.
- Kỹ thuật đặt câu hỏi và so khớp của Prolog là những phương tiện mềm dẻo cho phép truy cập từ cơ sở dữ liệu những thông tin có cấu trúc.
- Cần sử dụng phương pháp trừu tượng hoá dữ liệu như là một kỹ thuật lập trình cho phép sử dụng các cấu trúc dữ liệu phức tạp một cách đơn giản, làm chương trình trở nên dễ hiểu. Trong Prolog, phương pháp trừu tượng hoá dữ liệu rất dễ triển khai.
- Những cấu trúc toán học trừu tượng như ôôtômat cũng rất dễ cài đặt trong Prolog.
- Người ta có thể tiếp cận đến nhiều lời giải khác nhau cho một bài toán nhờ sử dụng nhiều cách biểu diễn dữ liệu khác nhau, như trường hợp



bài toán tám quân hậu. Cách biểu diễn dữ liệu sử dụng nhiều thông tin tiết kiệm được tính toán, mặc dù làm cho chương trình trở nên rườm rà, khó cô đọng.

- Kỹ thuật tổng quát hoá một bài toán, tuy trừu tượng, nhưng lại làm tăng khả năng hướng đến lời giải, làm đơn giản hoá phát biểu bài toán.

### **Làm việc với tệp**

Cùng với chế độ tương tác *câu hỏi-trả lời*, quá trình vào ra và chế độ làm việc với tệp đã làm phong phú môi trường làm việc của Prolog.

- Các tệp trong Prolog đều hoạt động theo kiểu tuần tự. Prolog phân biệt dòng vào hiện hành và dòng ra hiện hành.
- Thiết bị cuối (terminal) của NSD gồm màn hình và bàn phím được xem như một tệp giả có tên là user.
- Prolog có nhiều vị từ có sẵn để xử lý các dòng vào-ra.
- Khi làm việc với tệp, chế độ đọc ghi là xử lý từng ký tự hoặc từng hạng.

## Bài tập chương 5

### 1. Cho chương trình :

p( 1 ).  
p( 2 ) :- !.  
p( 3 ).

Cho biết các câu trả lời của Prolog từ các câu hỏi sau :

- (a) ?- p( X ).
- (b) ?- p( X ), p( Y ).
- (c) ?- p( X ), !, p( Y ).

### 2. Quan hệ sau đây cho biết một số có thể là dương, bằng không, hoặc âm :

sign( Number, positive) :-  
    Number > 0.

sign( 0, null).

sign( Number, negative) :-  
    Number < 0.

Hãy sử dụng kỹ thuật nhất cắt để viết lại chương trình trên hiệu quả hơn.

### 3. Thủ tục separate(Number, Positive, Negative) xếp các phần tử trong danh sách Number lần lượt thành hai danh sách, danh sách Positive chỉ chứa các số dương, hoặc bằng không, danh sách Negative chỉ chứa các số âm.

Ví dụ :

separate( [ 3, -1, 0, 5, -2 ], [ 3, 0, 5 ], [ -1, -2 ] )

Hãy định nghĩa thủ tục trên theo hai cách, một cách không sử dụng kỹ thuật nhất cắt, một cách có sử dụng kỹ thuật nhất cắt.

### 4. Cho hai danh sách, Accept và Reject, hãy viết danh sách các đích sử dụng kỹ thuật quay lui và các quan hệ member và not để tìm các phần tử có mặt trong Accept nhưng không có mặt trong Reject.

### 5. Định nghĩa thủ tục difference( Set1, Set2, SetDiff) tìm hiệu hai tập hợp Set1 và Set2 với quy ước các tập hợp được biểu diễn bởi các danh sách.

6. Chẳng hạn :

`difference( [ a, b, c, d ], [ b, d, e, f ], [ a, c ] )`

7. Hãy định nghĩa vị từ `unifiable( List1, Term, List2)` để kiểm tra so khớp, trong đó `List2` là danh sách tất cả các phần tử của `List1` có thể so khớp với `Term` nhưng không thực hiện phép thế trên các biến đã được so khớp. Ví dụ :

`?- unifiable( [ X, bibo, t( Y ) ], t(a), List ).`

`List = [X, t( Y )]`

Chú ý rằng `X` và `Y` vẫn là các biến tự do không thực hiện phép thế `t(a)` cho `X`, hay phép thế `a` cho `Y`. Muốn vậy, thực hiện hướng dẫn sau :

Sử dụng phép phủ định `not( Term1 = Term2)`. Nếu quan hệ `Term1 = Term2` được thoả mãn, khi đó, `not( Term1 = Term2)` sẽ thất bại, và phép thế biến không xảy ra.

8. Bài toán mã đi tuần. Giả sử các ô của bàn cờ vua  $8 \times 8$  được biểu diễn bởi các cặp toạ độ có dạng `X/Y`, với `X` và `Y` nằm trong khoảng 1 và 8.

(a) Định nghĩa quan hệ `jump( case1, case2 )`, bằng cách sử dụng luật đi của quân mã, và giả sử rằng `case1` luôn luôn bị ràng buộc. Ví dụ :

`?- jump( 1/1, C ).`

`C = 3/2;`

`C = 2/3;`

`No`

(b) Định nghĩa quan hệ `mvt_knight( path )`, với `path` là một danh sách gồm các ô biểu diễn lộ trình các bước nhảy hợp lý của quân mã trên bàn cờ rỗng.

(c) Sử dụng quan hệ `mvt_knight`, viết một câu hỏi để tìm tất cả các lộ trình bốn bước nhảy hợp lý của quân mã, xuất phát từ ô có toạ độ `2/1`, để đến biên bên phải của bàn cờ (`Y = 8`) và để đến ô `5/4` sau hai bước nhảy.

9. Cho `f` một tệp chứa các hạng, hãy định nghĩa thủ tục `findterm(Term)` để đưa ra màn hình hạng đầu tiên của `f` khớp được với `Term` ?

10. Cho `f` một tệp chứa các hạng, hãy định nghĩa thủ tục `findallterm(Term)` để đưa ra màn hình tất cả các hạng của `f` khớp được với `Term` ? Kiểm tra tính chất biến `Term` không thể được gán giá trị khi thực hiện tìm kiếm.

11. Hãy mở rộng thủ tục `del_space` đã được trình bày trong phần lý thuyết để có thể xử lý loại bỏ các dấu cách thừa nằm trước dấu phẩy (comma) và chỉ giữ lại một dấu cách nằm ngay sau dấu phẩy.
12. Tương tự bài 3 cho các dấu chấm câu khác như dấu chấm (period), dấu chấm phẩy (semicolon), dấu chấm hỏi (question mark), v.v...
13. Định nghĩa quan hệ `firstchar(Atom, Char)` cho phép kiểm tra Char có phải là ký tự đầu tiên của Atom không (Atom bắt đầu bởi Char) ?
14. Định nghĩa thủ tục cho phép đổi một danh từ tiếng Anh từ số ít (singular) sang số nhiều (plural) được thực hiện như sau :  
?- plural ( table, X ).  
X = tables  
Yes
15. Áp dụng thủ tục `readsentence` đã được trình bày trong phần lý thuyết để xây dựng thủ tục :  
?- find( Keyword, Sentence ).  
cho phép tìm trong tệp đang đọc một câu có chứa từ khoá Keyword. Câu Sentence phải ở dạng mới được đọc vào chưa xử lý, nghĩa là được biểu diễn bởi một chuỗi ký tự, hoặc bởi một nguyên tử.