## exercise 1

book(1,'The art of Prolog',400).

book(23,'The mistery of Strawberries',42).

person('Statler').

person('Waldorf').

author('Statler',1).

author('Waldorf',23).

hates('Statler',1).

owns('Waldorf',23).

1怎么把把fact写入KB

?- consult(user).

|book(1, 'The Art of Prolog', 400).

|: <Ctrl+D>

文本

中度可信度描述已自动生成

Ctrl+D退出；写入成功

知识点1，变量可以和其他变量或atom进行unify，当我们在query里使用变量时，实际上是在问：'is there a value for this variable(s) which evaluates the query to true'

2通过文件获得KB

前面的方法，需要每次打开软件后写入；

通过文件就比较方便，比如在记事本里写好，并给.pl后缀

?- emacs 会给出一个文本编辑器

注意打开的只是srcatch不会保存，需要新建file

3.如果有多条备选结果，使用分号查看；，如果enter，则退出query

4，加入rule到KB里，然后在query里使用

brochure(ISBN):-

book(ISBN,\_,Pages),

Pages < 100.

知识点2，在rule的名称后面标识/1,/2,/3，是表示有多少个参数

练习1 写一个query来获取ISBN numbers.

1. ?- book(ISBN,\_,\_).

想提取什么，就把

练习2 获取 book name hated by their author

2. ?- book(ISBN,Title,\_),hates(Person,ISBN),author(Person,ISBN).  
Notice that this shows the bindings to the variables Person and ISBN as well. If we want to 'hide' those, we can define a new predicate and query that predicate:  
上面这个，三个fact里都有ISBN，即同一个ISBN，后面两个是，同Person

会显示所有出现的变量

下面的写法，会只显示predicate里的变量

hated\_book(Title):- //这里写返回什么东西

book(ISBN,Title,\_), //下面写要满足什么条件的东西，

hates(Person,ISBN),

author(Person,ISBN).

3. 获取书的title和author, 并且print each result

?- author(Person,ISBN),book(ISBN,Title,\_),write('The book '),write(Title),write(' is written by '),write(Person).  
Notice that if you want to hide the variable-bindings here by introducing a new predicate (as in exercise 2)

print\_info:-

author(Person,ISBN),

book(ISBN,Title,\_),

write('The book '),

write(Title),

write(' is written by '),

write(Person).

prolog will only print 'Yes' when you ask the query ?- print\_info.. Why? Since there are no variables in the query, prolog does not have to show you all possible bindings. You ask prolog: 'can you print info?' and since it can do this once it answers 'Yes'. So it is not possible to hide the variable-bindings here.

练习4 写一个叫做proud\_author/1，变量只有一个的predicate，来获取person，拥有自己的书

proud\_author(Person):-

author(Person,Book),

owns(Person,Book).

练习5 建一个KB，翻译current book KB到自己的语言

For Portuguese it would look as follows:

livro(X, Y, Z) :- book(X, Y, Z).

pessoa(X) :- person(X).

autor(X, Y) :- author(X, Y).

odeia(X, Y) :- hates(X, Y).

possui(X, Y) :- owns(X, Y).

6.

The most important difference between 'books' and 'books in a library' is that often a library has more than one copy of a specific book. In other words, the difference between a book (the result of intellectual work, which is unique) and a copy/print of a book (which is usually not unique) becomes important. One possible representation is to extend the book/3 predicate with an extra argument: a copy-number. As discussed in the session there may be other solutions, that is - other representations which mainly depend on the interpretation of the task at hand. One of which is to introduce a fact for each library and book including the number of copies:

bookcopy(1, 'The Art of Prolog', 400, 'Library of KULeuven', 5).

bookcopy(1, 'The Art of Prolog', 400, 'Library of HUB', 3).

Once you are interested in distinguishing between the copies you need to write different facts for each copy. That is:

bookcopy(1,'The art of Prolog',400, 'Library of KULeuven', 1).

bookcopy(1,'The art of Prolog',400, 'Library of KULeuven', 2).

...

bookcopy(1,'The art of Prolog',400, 'Library of KULeuven', 5).

...

Each of the facts identifies a different copie - one with ID 1, the second copy with ID 2, etc.

As you already see from this, the danger is the introduction of redundancy, with all the problems it may cause (extra work to add data and correct data, chances of introducing inconsistency, extra storage space,...).

An alternative, cleaner representation is obtained by linking the new predicate bookcopy/3 with the information already encoded in book/3. We then represent the knowledge (given in the second interpretations) as follows:

book(1,'The art of Prolog',400).

bookcopy(1,'Library of KULeuven', 1).

bookcopy(1,'Library of KULeuven', 2).

...

The first argument of the bookcopy/3 predicate is the ISBN number of the book, the second argument is the library in which this book copy is stored, and the third distinguishes between different copies of the same book in the same library. This nicely seperates the concept of a book from the concept of a copy of the book. This eliminates the redundancy, so this is clearly the prefered representation.

在做数据库的时候，考虑冗余的情况，尽量链接到某一个已存在的fact类型上，再给予额外属性例如ID来识别

## exercise 2 Recursion

在其他语言里可以用循环语句来实现，但是PLPM里没有

在Prolog里是通过以下方式：

定义一个解决complete problem的solution

in terms of 解决一个同问题的小variant

以求和问题为例 26得8， 13758得24

这个求和问题，分解为两部分

第一部分，base case，即当只剩一位数的时候，即num<10，此时把该数当作temp sum ，加到之前的sum里

第二部分，recursive case，当大于一位数的时候，即num>10，通过模运算和整数除法，将该数字分解为个位digit和rest digits，把个位digit加到，temp sum里，并继续分解rest digits，直至符合base case.

练习1，

文本描述：

从left邻居得到纸片：如果没字，则告诉左边邻居，已完成

如果有字，取出第一个letter，传给右边邻居；当右边邻居传达已完成，你把字写到黑板上，并告诉左边邻居已完成；

hello，因为是从o开始写上去，所以最后得到reverse

从left邻居得到纸片：如果没字，则告诉左边邻居，已完成

如果有字，取出第一个letter，把字写到黑板，再传给右边邻居；

non-reverse

Solutions for Session 2

Note that usually there exist multiple correct solutions for an exercise. Below we give one possible solution for each exercise.

Here is the call tree for the query ?- count(437,Sum).

A call tree corresponds to a Prolog execution trace. For a better understanding of call trees, study Chapters 1 and 2 of the Bratko book, with a focus on Section 1.5 How Prolog answers questions, and all Sections of Chapter 2: Syntax and Meaning of Prolog Programs. If you have problems with understanding how Prolog finds answers to queries, give extra focus to Section 2.4 Procedural meaning, and Section 2.5 Order of clauses and goals.

Should you like some extra reading specifically on how to draw call trees, a manual on drawing Prolog call trees (a.k.a. search trees) can be found here.

In some of the solutions we will use the predicates even/1 and odd/1. Here are their definitions:

even(N):-

A is mod(N,2),

A = 0.

odd(N):-

A is mod(N,2),

A = 1.

In the case for Group A the steps result in writing 'olleh': the word is reversed. While for Group B the output is simply 'hello'. The prolog program (for Group A) would be this: (Note: if you try this on your computer, this will not work since we used the predicate split\_string/3 which is not a built-in, so you would have to define it yourself):

reverse(''). % the base case

reverse(Word):- % the recursive case

split\_string(Word,First,Rest),

reverse(Rest),

write(First).

Note that if we would swap the last two lines in the definition of the recursive case it is the solution for Group B: the program would write 'hello' (without reversing).

练习2

Roads: a)

reach(A, B):-

direct\_road(A, B).

reach(A, B):-

direct\_road(A, A1),

reach(A1, B).

b) One possible idea is:

add to after your facts a rule that reverses the 'direct\_road/2'. That is:

direct\_road(A, B):-

direct\_road(B, A).

with this rule the reach will fall into a loop. Why? Draw a call tree to make sure. In a later session we will see a solution to this problem.

potential solution: 需要记录已经visited的节点

练习3 判断一个数字各个位置上是不是都even

思路，把整个数字拆分成1位数， base case就是判断1位数的情况

all\_even(Number):- % the base case

Number<10,

even(Number).

all\_even(Number):- % the recursive case

Number>=10,

split\_number(Number,Digit,NewNumber),

even(Digit),

all\_even(NewNumber).

split\_number(Number,Digit,NewNumber) :-

Digit is mod(Number,10),

NewNumber is Number//10.

Here is a sketch of the call tree for the query ?- all\_even(2496)..

Here is a sketch of a call tree indicating the need for the Number>=10 check..

练习4，给出input里，有几位even

count\_even(Number,1):- % a base case

Number<10,

even(Number).

count\_even(Number,0):- % the other base case

Number<10,

odd(Number).

count\_even(Number,Count):- % the recursive case

Number>=10,

split\_number(Number,Digit,NewNumber),

DigCount is mod(Digit+1,2), % this is a small 'trick': DigCount will be 1 if Digit is even, 0 otherwise

count\_even(NewNumber,NumCount),

Count is DigCount + NumCount.

Here is a sketch of the call tree for the query ?- count\_even(24,A). Here is the call tree for ?- count\_even(25,A)..

练习5

?- funny(6).

6

3

10

5

16

8

4

2

I will quit

Yes

?- funny(7).

7

22

11

34

17

52

26

13

40

20

10

5

16

8

4

2

I will quit

Yes

When we move the two write(N), nl lines one line higher, the output is as follows.

?- funny(6).

6

3

3

10

5

5

16

8

4

2

I will quit

Yes

?- funny(7).

7

7

22

11

11

34

17

17

52

26

13

13

40

20

10

5

5

16

8

4

2

I will quit

Yes

It prints the odd numbers twice, the reason is that if funny(N) is called with N being odd, prolog jumps in the first recursive clause for funny/1 (the clause for even numbers), writes the odd number to the screen, fails on the condition even(N) and backtracks to the second recursive clause for funny/1 (the clause for odd numbers) where it writes the odd number to the screen again, ... .

主要是讲语句执行的顺序

Given your current knowledge of prolog, the following prolog program is probably the most straightforward way to compute fibonacci numbers:

fib(1,1).

fib(2,1).

fib(N,F):-

N>2,

N1 is N-1,

fib(N1,F1),

N2 is N-2,

fib(N2,F2),

F is F1+F2.

This is a rather slow way of computing fibonacci numbers. To see why it is slow, try to draw the call tree for fib(5,N). You will see that using the above program, prolog will amongst others compute fib(3,N) twice. Now if we would for instance try fib(10,N), there will be much more computations that are done multiple times. In one of the later sessions we will go more into detail about this and we will see a more efficient solution to this problem.

a)

double\_digit(N):- %输入一个数字

N>10, %判断是不是两位数以上

split\_number(N,Digit,NewN), %如果是，这一步，把该数字分为，最后一位Digit，和剩余位（NewN）

occurs\_in(Digit,NewN). %如果上面能分成功，则这一步判断最后一位，是否出现在剩余位里

double\_digit(N):-

N>10,

split\_number(N,\_,NewN),

double\_digit(NewN).

We have used an auxiliary predicate occurs\_in/2:

occurs\_in(Digit,Number):- %取上面得到的最后一位Digit，和剩余位(Number)

% the base case:

split\_number(Number,Digit,\_). %这一步若要分成功，则说明，剩余位(Number)的最后一位（Digit2）等于当前的最后一位Digit；若成功，说明一开始的最后一位Digit出现了两次，成功退出；

occurs\_in(Digit,Number):- %若上一步没成功，即执行下面的

% the recursive case

Number>=10, %首先判断剩余位是不是大于两位；如果不大于，那上面fail，这里也fail；第一个末位Digit的检索就结束了，fail；

split\_number(Number,\_,NewNumber), %若大于两位，则把剩余位的最后一位舍去(这里其实是用­\_表示无所谓这个值是多少)，然后得到新的剩余位（NewNumber）

occurs\_in(Digit,NewNumber). %判断该末位Digit，是否在剩余位中

b) This is is a little bit easier:

double\_digit\_nextto(N):-

N>10,

split\_number(N,Digit,NewN),

split\_number(NewN,Digit2,\_),

Digit2 = Digit.

double\_digit\_nextto(N):-

N>10,

split\_number(N,\_,NewN),

double\_digit\_nextto(NewN).

Solutions for Session 3

Not every pair that somewhere contains a list is a list itself. For example .([],1) is a pair which is not a list although it contains the empty list (it is not a list because the second element of the pair is not a list).

The first and third query will fail. The second query will succeed and results in the following bindings:

A = a

B = [b,c]

C = d

?- .(a, .(.(b,.(c,[])),.(d,[]))) = [A|B].

A = a

B = [[b,c],d]

Yes

?- .(a, .(.(b,.(c,[])),.(d,[]))) = [A,B|C].

A = a

B = [b,c]

C = [d]

Yes

?- [ a, b | [] ] = [ H | T].

H = a

T = [b]

Yes

?- [ a, b, c] = [ E1, E2 | T ].

E1 = a

E2 = b

T = [c]

Yes

?- .(a, .(b, .(c,.(d,[])))) = [A,B|C].

A = a

B = b

C = [c,d]

Yes

?- X = a, Y = [ 1, [2,3] ], Z = [X | Y].

X = a

Y = [1,[2,3]]

Z = [a,1,[2,3]]

Yes

?- X = a, Y = [ 1, [2,3] ], Z = [X,Y].

X = a

Y = [1,[2,3]]

Z = [a,[1,[2,3]]]

Yes

?- X = [a], Y = [ 1, [2,3] ], Z = [X,Y].

X = [a]

Y = [1,[2,3]]

Z = [[a],[1,[2,3]]]

Yes

?- listlength([1,[2,3]],N).

N = 2

Yes

?- listlength([1|[2,3]],N).

N = 3

Yes

If you do not see why these are the right answers, draw the tree representations of [1,[2,3]] and [1|[2,3]] (you will see that [1|[2,3]] is the same as [1,2,3]). Here are call trees for the listlength([1,[2,3]],N) query and the listlength([1|[2,3]],N) query (note that these trees are not complete, as they don't show the alternative branch for the last listlength goal).

?- listlength(L,4).

L = [\_61,\_63,\_65,\_67]

Yes

The exact numbers (\_61, \_63, ...) can differ.

The above is a way to generate a list with a given number (in this case 4) of uninstantiated elements. In some of the next sessions this will turn out to be very useful. Here is a calltree for the query.

listsum([],0).

listsum([A|Rest],Sum):-

listsum(Rest,Sum1),

Sum is Sum1 + A.

The most obvious solution for the average is the following:

listaverage([],0).

listaverage([H|T],Average) :-

listsum([H|T],Sum),

listlength([H|T],Length),

Average is Sum/Length.

In the above solution we traverse the list twice. We can also compute the average by traversing the list only once (which is more efficient):

listavg([],0).

listavg([H|T],Avg):-

listsumlength([H|T],Length,Sum),

Avg is Sum/Length.

listsumlength([],0,0).

listsumlength([A|Rest],Length,Sum):-

listsumlength(Rest,Length1,Sum1),

Length is Length1 + 1 ,

Sum is Sum1 + A.

countdisk([],0).

countdisk([w|T],C) :- countdisk(T,C1), C is C1+1.

countdisk([b|T],C) :- countdisk(T,C1), C is C1+1.

countdisk([n|T],C) :- countdisk(T,C).

The first queries are easy.

?- countdisk([w,b,n,n],Count).

Count = 2

Yes

?- countdisk([w,b,n,n],2).

Yes

?- countdisk([w,b,n,n],4).

No

The last query might seem a bit strange.

?- countdisk(List,Count).

List = [],

Count = 0

;

List = [w],

Count = 1

;

List = [w, w],

Count = 2

;

List = [w, w, w],

Count = 3

;

List = [w, w, w, w],

Count = 4

And so on (we get an infinite number of results).

The nb\_rounds/2 predicate:

nb\_rounds([],0).

nb\_rounds([\_],1).

nb\_rounds([\_,\_|T],Count):-

nb\_rounds(T,TmpCount),

Count is TmpCount+1.

The split/3 predicate:

split([],[],[]).

split([A],[A],[]).

split([A,B|T],[A|T1],[B|T2]):-

split(T,T1,T2).

You do not have to write a merge/3 predicate now that you already have the split/3 predicate! Depending on which arguments you instantiate, you can use split/3 to split a given list into two lists as well as to merge two given lists in another list !!

To split a given list:

?- split([move(b,1,b,3), move(c,10,c,8), move(b,3,b,5)],BlackMoves,WhiteMoves).

BlackMoves = [move(b,1,b,3),move(b,3,b,5)]

WhiteMoves = [move(c,10,c,8)]

Yes

To merge two given lists:

?- split(AllMoves,[move(b,1,b,3), move(b,3,b,5)],[move(c,10,c,8)]).

AllMoves = [move(b,1,b,3),move(c,10,c,8),move(b,3,b,5)]

Yes

If you really want to have a merge/3 predicate available, you can simply implement it as follows.

merge(L1,L2,L) :-

split(L,L1,L2).