Logic for CS

黃瀚萱
Department of Computer Science
National Chengchi University
2020 Spring

Schedule, Part I

Date	Topic
3/6	Introduction to this course
3/13	Thinking as computation
3/20	Propositional Logic
3/27	Logic Inference
4/3	Off
4/10	First Order Logic
4/17	Interpretation of FOL
4/24	Inference in FOL (Online)

Schedule, Part II

Date	Topic
5/1	Prolog Basics & KR (Online)
5/8	Midterm Exam
5/15	Prolog Programming
5/22	Prolog Programming II
5/29	Logic Programming (Non-remote)
6/5	Discussion of the Final Project (Non-remote)
6/12	Final Project Presentation (Non-remote)
6/19	Term Exam (Non-remote)

Logic Programming with Prolog II

Writing Correct Prolog Programs

- In last week, we talked about the syntax of Prolog.
- We have known how to write *grammatically correct* Prolog programs.
- Grammatically correct programs are not correct programs.
  - Given a garbage program to Prolog, Prolog will output a garbage results, even the program is grammatically correct.
- How to right a correct program in Prolog?

The Truth in Prolog

- How do we know whether a grammatically correct program will do what it is supposed to do?
  - Perhaps clearer than other programming language.
- Elements of a Prolog program
  - Atomic sentences
  - Conditional sentences
- We can exam the truth of these sentences.

Truth in Mind

- We define an atomic sentence "child(john, sue)" with a predicate **child**.
  - Prolog does not concern the meaning of "child"; it is just a symbol.
  - But programmers have a definite idea in mind by "John is a child of Sue".
  - The programmers believe or are willing to assume the English sentence is true.

```
child(john,sue).
```

Truth in Mind

- The following clause is rooted from the English sentence "If x is a child of y, then y is the parent of x" to be true.

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

- If we wrongly define the clause as follows, Prolog will never complain; it is grammatically correct.

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

- Garbage in garbage out.

- It is better to add a comment to help anyone understanding.

Part of the Truth is Missing

- However, comments cannot ensure all the clauses are true.

- Prolog may output an incorrect answer if some facts are missing.

```
child(john, sue)

?- child(john, sam)
```

Part of the Truth is Missing

- "Two people are of opposite sex if one of them is male and the other is female."

- "x is of opposite sex from y if x is male and y is female".

```
opp_sex(X,Y) :- male(X), female(Y)

?- opp_sex(jane, sam)?
```

- The converse is missing.

```
opp_sex(Y,X) :- male(X), female(Y)
```

The Whole Truth

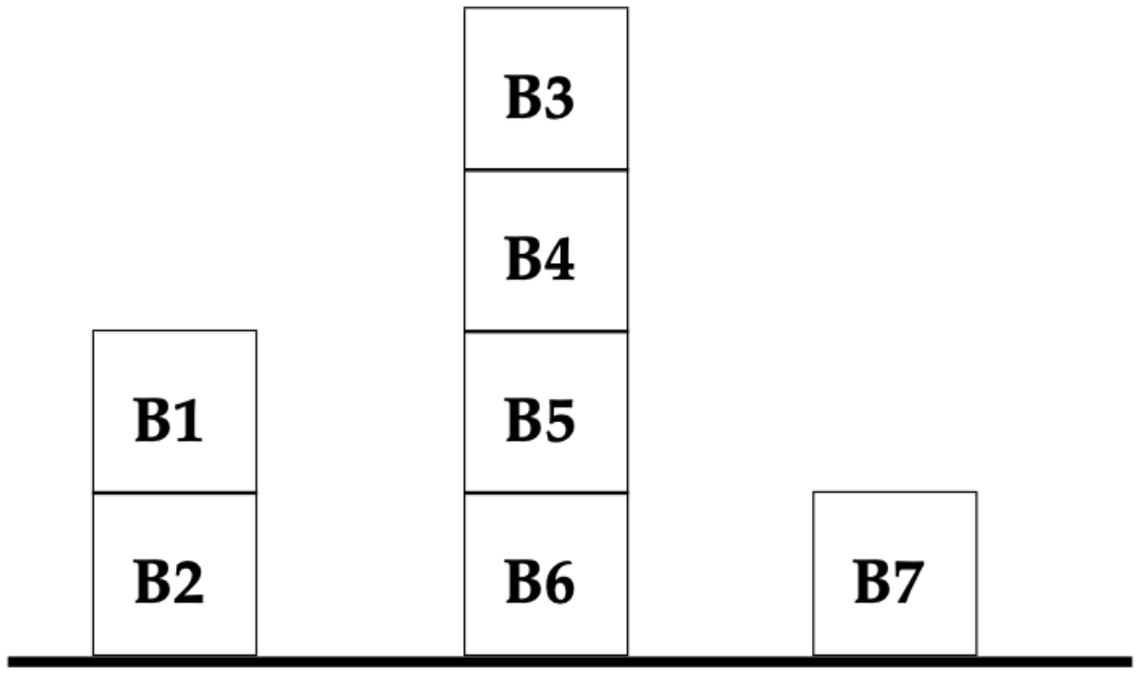
- A Prolog program must include enough clauses to spell out the whole truth.
- A program might not contain all possible truths.
  - The family program does not include the parent of George.
  - It is acceptable if some truths might be missing.
- But a program must include all relevant truths.
  - Explicit: Directly given in the clauses.
  - Implicit: Can be inferred from explicit truths.

Rules of Writing Prolog Programs

- Grammatically correct
- All clauses must be true.
  - **The truth, and nothing but**
  - **The whole (relevant) truth**
- All the necessary clauses must be represented in a way that Prolog can find them using back-chaining.
  - **In a form suitable for back-chaining**
- To make the program logically correct.

Block Problem

- How describe the scene?
- Ask following questions
  - Is B3 above B5?
  - Is B1 to the left of B7?
  - Is B4 to the right of B2?



```
on(b1, b2).
on(b3, b4).
on(b4, b5).
on(b5, b6).

just_left(b2, b6).
just_left(b6, b7).

above(X, Y) :- on(X, Y).
above(X, Y) :- on(X, Z), above(Z, Y).

left(X, Y) :- just_left(X, Y).
left(X, Y) :- just_left(X, Z), left(Z, Y).
left(X, Y) :- on(X, Z), left(Z, Y).
left(X, Y) :- on(Y, Z), left(X, Z).
```

Exam the Truth

- If x is directly on y, then x is above y.

```
above(X, Y) :- on(X, Y).
```

- If x is directly on z, and z is above y, then x is above y.

```
above(X, Y) :- on(X, Z), above(Z, Y).
```

- Both sentences are clearly true. (Soundness)
- Is this all the necessary information? (Completeness)

Exam the Truth

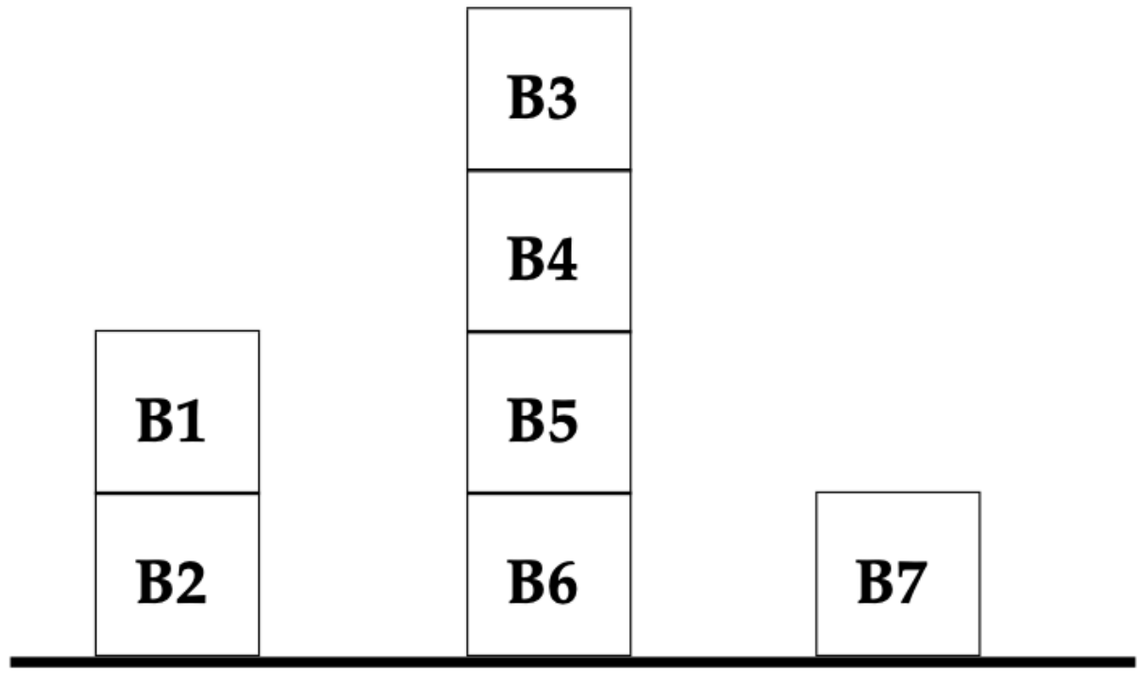
- For the query "is b1 above b2", Prolog determines it is true because B1 is directly on B2.
  - We need only

```
above(X, Y) :- on(X, Y).
```

```
[trace]  ?- above(b1, b2).
   Call: (8) above(b1, b2) ? creep
   Call: (9) on(b1, b2) ? creep
   Exit: (9) on(b1, b2) ? creep
   Exit: (8) above(b1, b2) ? creep
true .
```

Exam the Truth

- Recursive predicate




```
[trace]  ?- above(b3, b5).
   Call: (8) above(b3, b5) ? creep
   Call: (9) on(b3, b5) ? creep
   Fail: (9) on(b3, b5) ? creep
   Redo: (8) above(b3, b5) ? creep
   Call: (9) on(b3, _4484) ? creep
   Exit: (9) on(b3, b4) ? creep
   Call: (9) above(b4, b5) ? creep
   Call: (10) on(b4, b5) ? creep
   Exit: (10) on(b4, b5) ? creep
   Exit: (9) above(b4, b5) ? creep
   Exit: (8) above(b3, b5) ? creep
true .
```

Recursion in Prolog

- Most modern programming languages enable recursion.
- Recursion is really simple and lies at the heart of Prolog programming.
- A predicate recursive when the predicate appears in both of the head and the body sides.

```
above(X, Y) :- on(X, Y).
above(X, Y) :- on(X, Z), above(Z, Y).

left(X, Y) :- just_left(X, Y).
left(X, Y) :- just_left(X, Z), left(Z, Y).
left(X, Y) :- on(X, Z), left(Z, Y).
left(X, Y) :- on(Y, Z), left(X, Z).
```

When We Need Recursion in Prolog

- There is a predicate that involves using another predicate some number of times.
  - A block x is above a block y when there are some number *n* of intermediate blocks such that on(x,z_1), on(x,z_2), on(x,z_3), …, on(x,z_{n-2}), on(x,z_{n-1}), on(x,z_n), and on(z_n, y).
- *n* can be any number.
- *n* = 0: on(x, y)

Define a Recursive Predicate in Prolog

- Write the basic case (e.g., $n = 0$)
  - above(X, Y) :- on(X, Y).
- Write a clause that handles the $n + 1$ case based on the n case.
- above(X, Y) :- on(X, Z), above(Z, Y).
- Additional variable Z is introduced for the case *n.*

Mathematical Induction

- Math induction is not just for arithmetic.
  - The predicate above(x, y) will work correctly when x is above y with n intermediate blocks between them.
- If we can prove *S*(*n*) is true for all *n*, then the Prolog program for above will work no matter how many blocks.
  - Ensure *S*(0) is true.
  - Ensure *S*(*n*+1) is true if *S*(*n*) is true.
- So, the above predicate will work correctly no matter how many blocks between x and y.

Non-terminating Programs

- Infinite recursion
  - above(X) :- above(X).
- This clause is logically true.
- But, it violates the third rule:
  - **In a form suitable for back-chaining**
- The back-chaining algorithm will get stuck in the infinite loop.

```
[trace]  ?- loop(jane).
   Call: (8) loop(jane) ? creep
   Call: (9) loop(jane) ? creep
   Call: (10) loop(jane) ? creep
```

Infinite Recursion

- A smart clause may also result in infinite recursion.

```
male(john).
female(jane).
opp_sex(X, Y) :- male(X), female(Y).
opp_sex(X, Y) :- opp_sex(Y, X).
```

```
[trace]  ?- opp_sex(john,jane).
   Call: (8) opp_sex(john, jane) ? creep
   Call: (9) male(john) ? creep
   Exit: (9) male(john) ? creep
   Call: (9) female(jane) ? creep
   Exit: (9) female(jane) ? creep
   Exit: (8) opp_sex(john, jane) ? creep
true
```

```
male(john).
female(jane).
opp_sex(X, Y) :- male(X), female(Y).
opp_sex(X, Y) :- opp_sex(Y, X).
```

```
[trace]  ?- opp_sex(jane,john).
   Call: (8) opp_sex(jane, john) ? creep
   Call: (9) male(jane) ? creep
   Fail: (9) male(jane) ? creep
   Redo: (8) opp_sex(jane, john) ? creep
   Call: (9) opp_sex(john, jane) ? creep
   Call: (10) male(john) ? creep
   Exit: (10) male(john) ? creep
   Call: (10) female(jane) ? creep
   Exit: (10) female(jane) ? creep
   Exit: (9) opp_sex(john, jane) ? creep
   Exit: (8) opp_sex(jane, john) ? creep
true .
```

```
male(john).
male(tom).
female(jane).
opp_sex(X, Y) :- male(X), female(Y).
opp_sex(X, Y) :- opp_sex(Y, X).
```

```
[trace]  ?- opp_sex(john, tom).
   Call: (8) opp_sex(john, tom) ? creep
   Call: (9) male(john) ? creep
   Exit: (9) male(john) ? creep
   Call: (9) female(tom) ? creep
   Fail: (9) female(tom) ? creep
   Redo: (8) opp_sex(john, tom) ? creep
   Call: (9) opp_sex(tom, john) ? creep
   Call: (10) male(tom) ? creep
   Exit: (10) male(tom) ? creep
   Call: (10) female(john) ? creep
   Fail: (10) female(john) ? creep
   Redo: (9) opp_sex(tom, john) ? creep
   Call: (10) opp_sex(john, tom) ? creep
...(Infinite ...)
```

```
male(john).
male(tom).
female(jane).
male_female(X, Y) :- male(X), female(Y).
opp_sex(X,Y) :- male_female(X,Y).
opp_sex(X,Y) :- male_female(Y,X).
```

```
[trace]  ?- opp_sex(john, tom).
   Call: (8) opp_sex(john, tom) ? creep
   Call: (9) male_female(john, tom) ? creep
   Call: (10) male(john) ? creep
   Exit: (10) male(john) ? creep
   Call: (10) female(tom) ? creep
   Fail: (10) female(tom) ? creep
   Fail: (9) male_female(john, tom) ? creep
   Redo: (8) opp_sex(john, tom) ? creep
   Call: (9) male_female(tom, john) ? creep
   Call: (10) male(tom) ? creep
   Exit: (10) male(tom) ? creep
   Call: (10) female(john) ? creep
   Fail: (10) female(john) ? creep
   Fail: (9) male_female(tom, john) ? creep
   Fail: (8) opp_sex(john, tom) ? creep
false.
```

Evaluation Order

- The following clause

```
above(X, Y) :- on(X, Y).
above(X, Y) :- on(X, Z), above(Z, Y).
```

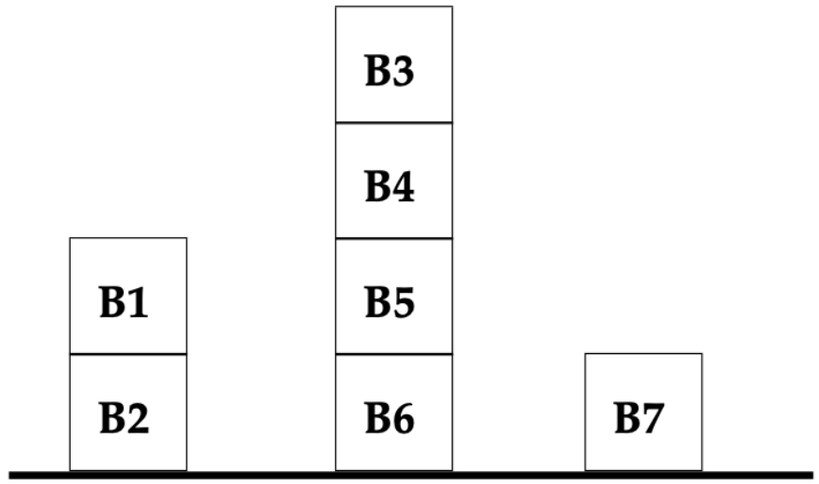
- and the following one are logically equivalent.

```
above(X, Y) :- on(X, Y).
above(X, Y) :- above(Z, Y), on(X, Z).
```

- But the second one may lead to infinite loop.
  - It does not work well for back-chaining even it is logically correct.

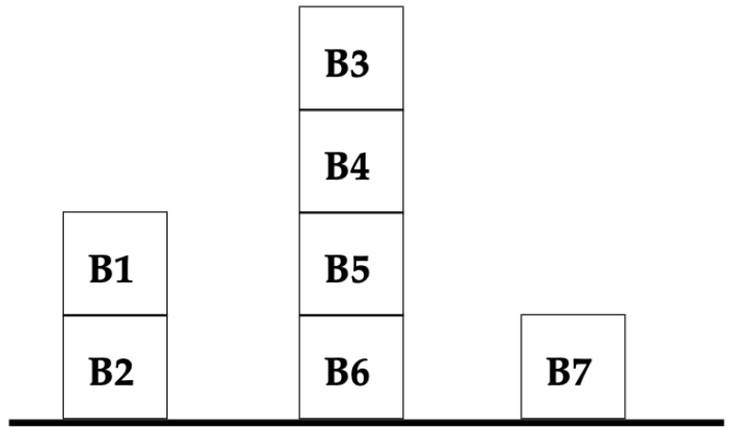
```
above(X, Y) :- on(X, Y).
above(X, Y) :- on(X, Z), above(Z, Y).
```

```
[trace]  ?- above(b2,b3).
   Call: (8) above(b2, b3) ? creep
   Call: (9) on(b2, b3) ? creep
   Fail: (9) on(b2, b3) ? creep
   Redo: (8) above(b2, b3) ? creep
   Call: (9) on(b2, _3126) ? creep
   Fail: (9) on(b2, _3126) ? creep
   Fail: (8) above(b2, b3) ? creep
false.
```

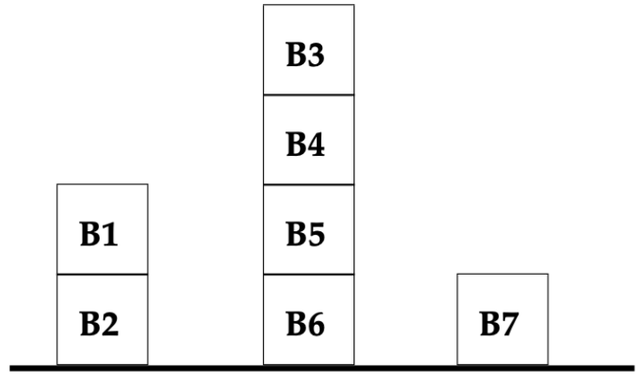


```
above(X, Y) :- on(X, Y).
above(X, Y) :- above(Z, Y), on(X, Z).
```

```
[trace]  ?- above(b2, b3).
   Call: (8) above(b2, b3) ? creep
   Call: (9) on(b2, b3) ? creep
   Fail: (9) on(b2, b3) ? creep
   Redo: (8) above(b2, b3) ? creep
   Call: (9) above(_3128, b3) ? creep
   Call: (10) on(_3128, b3) ? creep
   Fail: (10) on(_3128, b3) ? creep
   Redo: (9) above(_3128, b3) ? creep
   Call: (10) above(_3128, b3) ? creep
   Call: (11) on(_3128, b3) ? creep
   Fail: (11) on(_3128, b3) ? creep
   Redo: (10) above(_3128, b3) ? creep
   Call: (11) above(_3128, b3) ? creep
   Call: (12) on(_3128, b3) ? creep
..(Infinite ...)
```



Evaluation Order

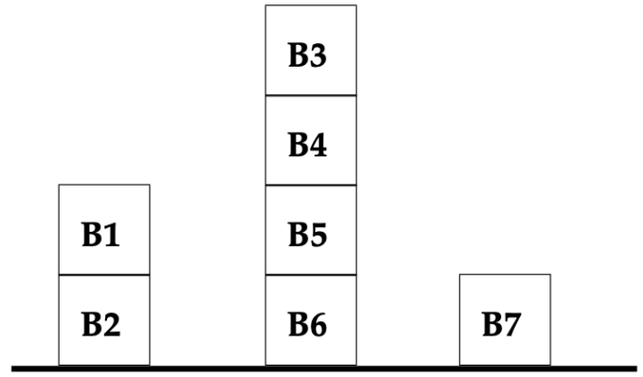


- How Prolog does perform a conjunctive query?

```
above(X, Y) :- above(Z, Y), on(X, Z).
above(b1, b1).
above(Z1, b1), on(b1, Z1).
above(Z2, b1), on(Z1, Z2), on(b1, Z1).
above(Z3, b1), on(Z2, Z3), on(Z1, Z2), on(b1, Z1).

...
```

Evaluation Order



- How Prolog does perform a conjunctive query?

```
above(X, Y) :- on(X, Z), above(Z, Y).
above(b1, b1).
on(b1, Z1), above(Z1, b1).   % Unify Z1 with b2
on(b1, b2), above(b2, b1).
on(b1, b2), on(b2, Z2), above(Z2, b1).
on(b1, b2), on(b2, Z2), above(Z2, b1).  % Fail to unify Z2

false.
```

Prevent from Non-terminating

- There is no simple to guarantee a program will terminate.
- A good rule of thumb:
  - The recursive predicate should appear toward the end of the clause.
  - Allowing new variables to be instantiated first.
- When the body contains a recursive predicate, make sure that its new variables are instantiated by earlier atoms in the body.
  - above(X, Y) :- on(X, **Z**), above(**Z**, Y).

Z will be instantiated first.

More Complex Predicate

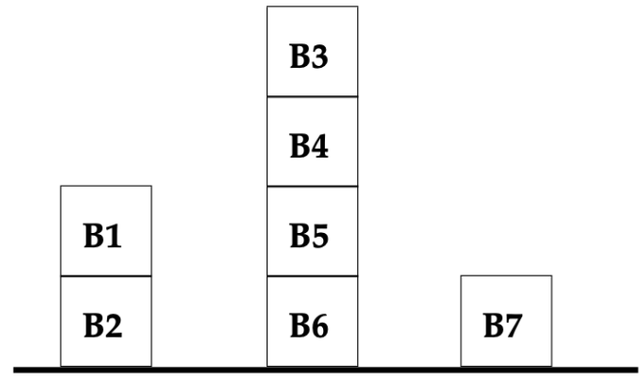
- The first two clauses are identical to **above**.

- The third clause is for the case where x is not directly on the table and there is an intermediate block z such that x is on z, and z is to the left of y.

- The last clause is for the case where y is not directly on the table and there is an intermediate block z such that y is on z, and z is to the left of x.

```
left(X, Y) :- just_left(X, Y).
left(X, Y) :- just_left(X, Z), left(Z, Y).
left(X, Y) :- on(X, Z), left(Z, Y).
left(X, Y) :- on(Y, Z), left(X, Z).
```

```
[trace]  ?- left(b1, b7).
   Call: (8) left(b1, b7) ? creep
   Call: (9) just_left(b1, b7) ? creep
   Fail: (9) just_left(b1, b7) ? creep
   Redo: (8) left(b1, b7) ? creep
   Call: (9) just_left(b1, _3132) ? creep
   Fail: (9) just_left(b1, _3132) ? creep
   Redo: (8) left(b1, b7) ? creep
   Call: (9) on(b1, _3132) ? creep
   Exit: (9) on(b1, b2) ? creep
   Call: (9) left(b2, b7) ? creep
   Call: (10) just_left(b2, b7) ? creep
   Fail: (10) just_left(b2, b7) ? creep
   Redo: (9) left(b2, b7) ? creep
   Call: (10) just_left(b2, _3132) ? creep
   Exit: (10) just_left(b2, b6) ? creep
   Call: (10) left(b6, b7) ? creep
   Call: (11) just_left(b6, b7) ? creep
   Exit: (11) just_left(b6, b7) ? creep
   Exit: (10) left(b6, b7) ? creep
   Exit: (9) left(b2, b7) ? creep
   Exit: (8) left(b1, b7) ? creep
true .
```



Find b2 is on the table

Check if b2 is left to b7

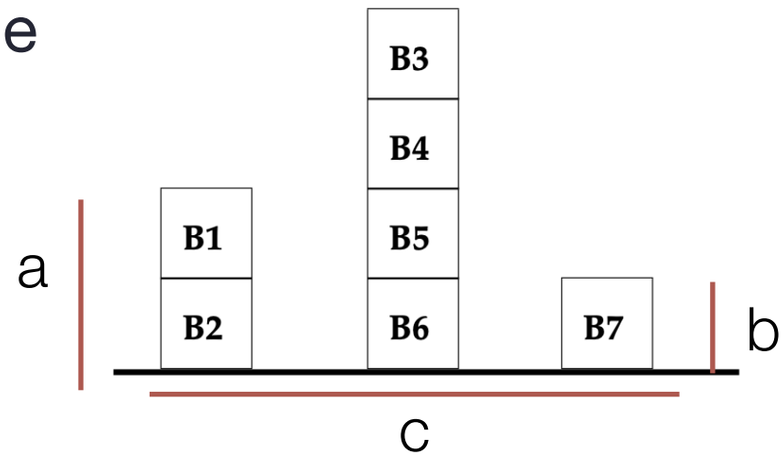
Find intermediate blocks which are on the table and between b2 and b7

b6 is on the table and b2 is left to b6

b6 is left to b7

Reconsideration of Recursion

- For the predicate **above**, we consider a number *n*, the number of intermediate blocks between a top and a bottom block.
  - The clause has to work for any $n > 0$.
- Measuring the depth of recursion
  - The depth of left(x, y) = a + b + c
    - a = Number of blocks between x and the table
    - b = Number of blocks between y and the table
    - c = Number of stacks between x and y.

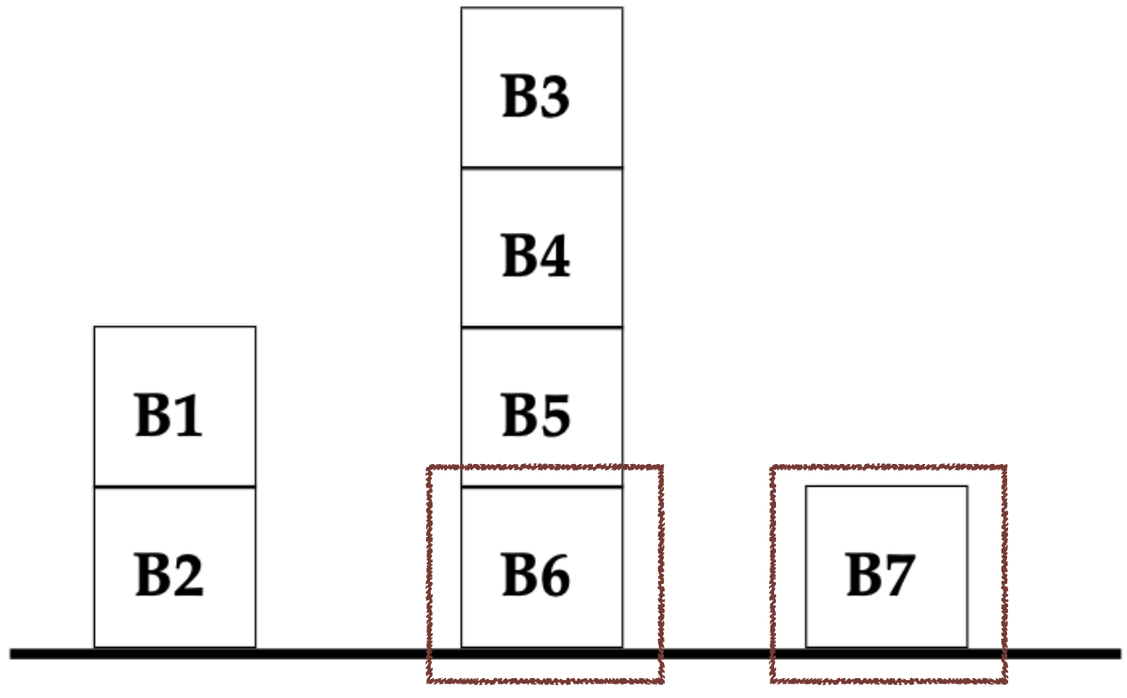


Handle the Cases in Recursion

- Write clauses to handle the case $n = 0$
  - `left(X, Y) :- just_left(X, Y).`
- Write clauses to handle the case $n + 1$
  - Both x and y are directly on the table
    - `left(X, Y) :- just_left(X, Z), left(Z, Y).`
  - x is on a block z
    - `left(X, Y) :- on(X, Z), left(Z, Y).`
  - y is on a block z
    - `left(X, Y) :- on(Y, Z), left(X, Z).`

Handle the Cases in Recursion

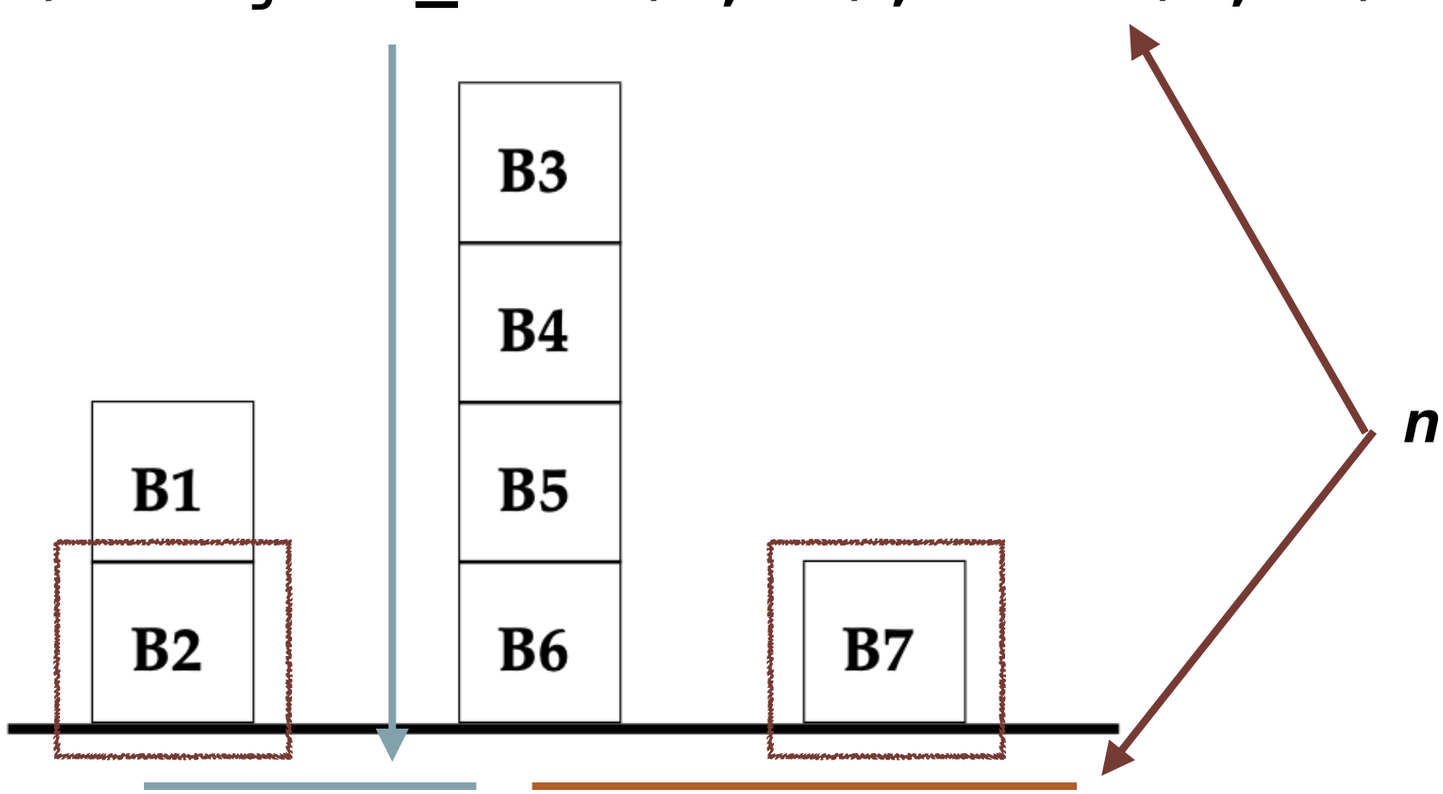
- Write clauses to handle the case $n = 0$
  - `left(X, Y) :- just_left(X, Y).`




```
?- left(b6, b7).
```

Handle the Cases in Recursion

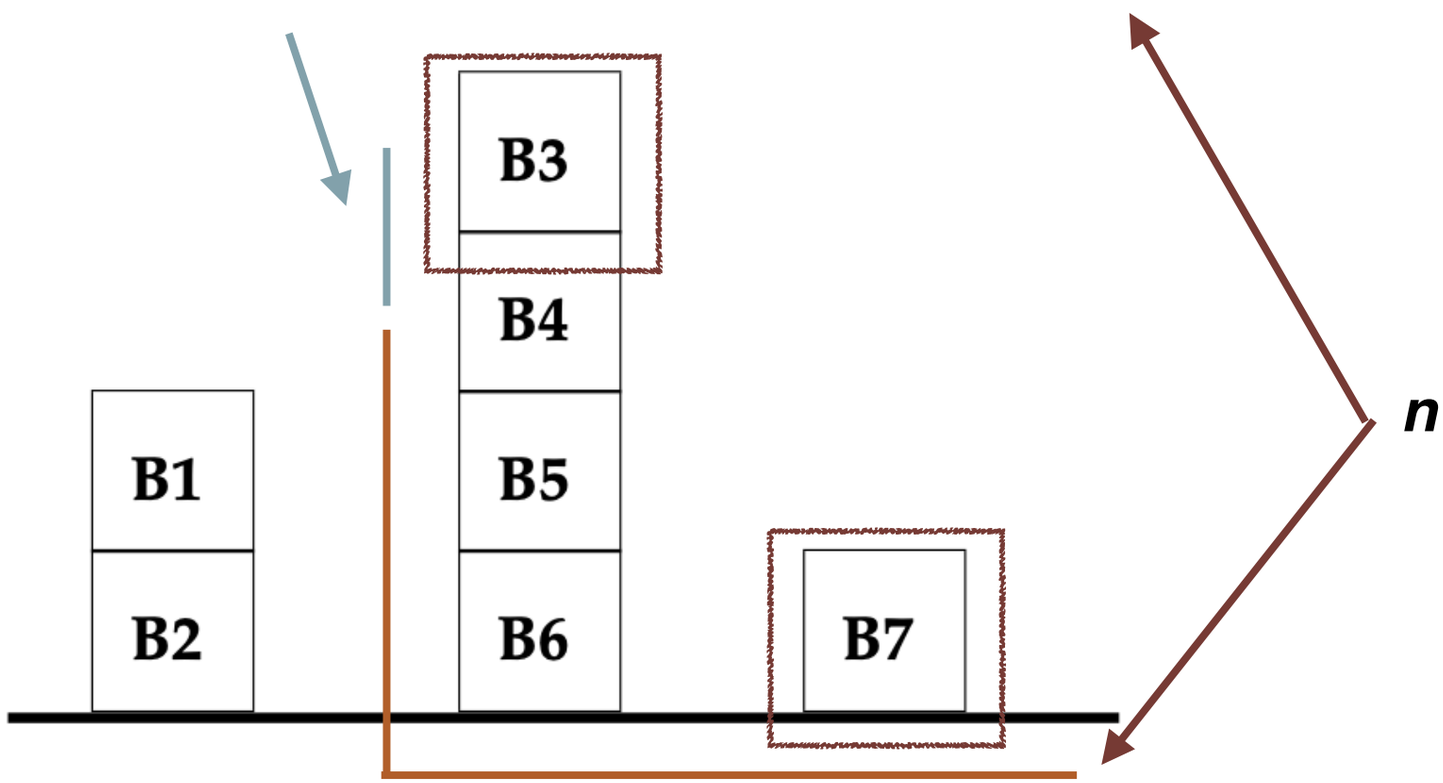
- Write clauses to handle the case *n* + 1
  - Both x and y are directly on the table
    - `left(X, Y) :- just_left(X, Z), left(Z, Y).`




```
?- left(b2, b7).
```

Handle the Cases in Recursion

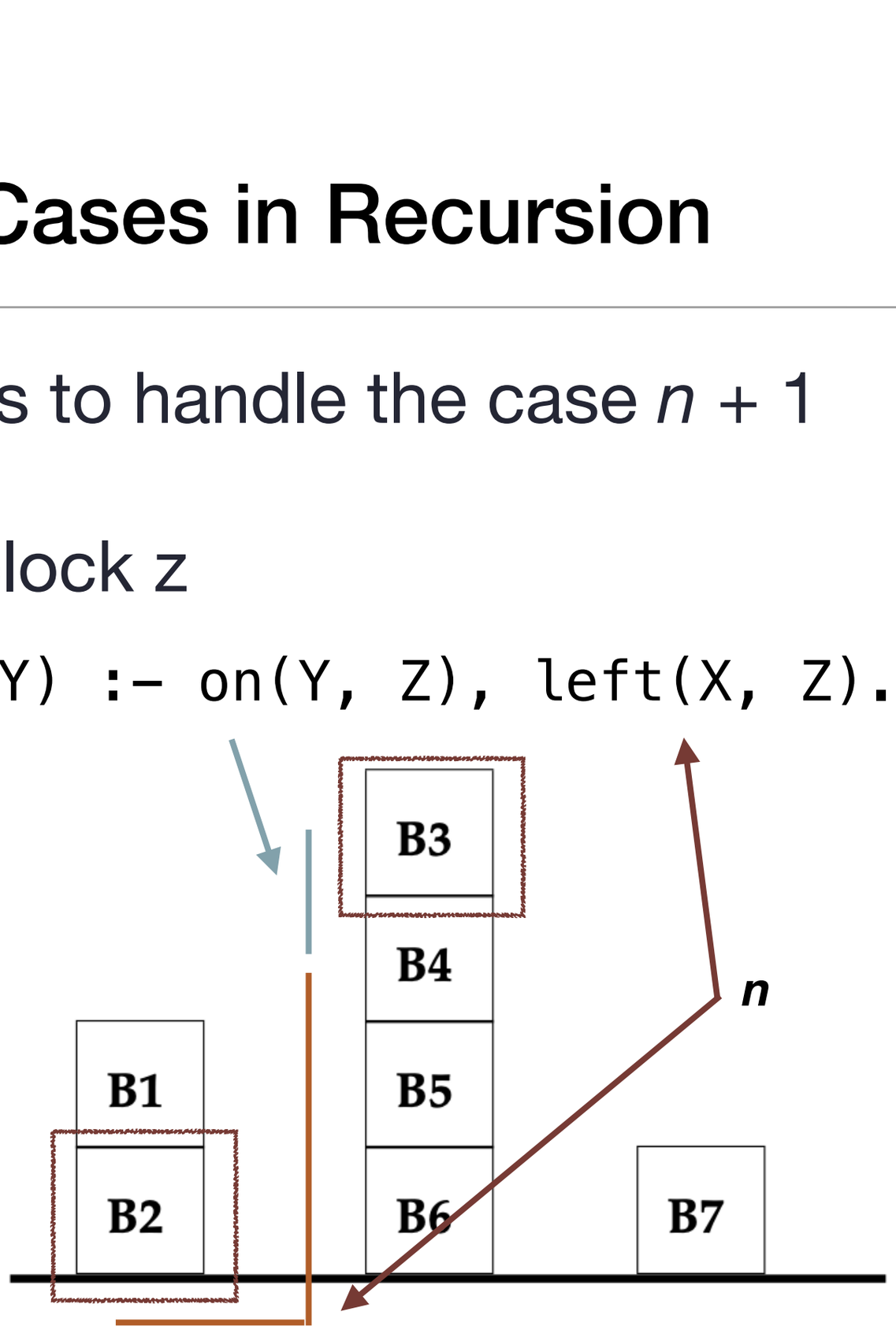
- Write clauses to handle the case *n* + 1
  - x is on a block z
    - `left(X, Y) :- on(X, Z), left(Z, Y).`




```
?- left(b3, b7).
```

Handle the Cases in Recursion

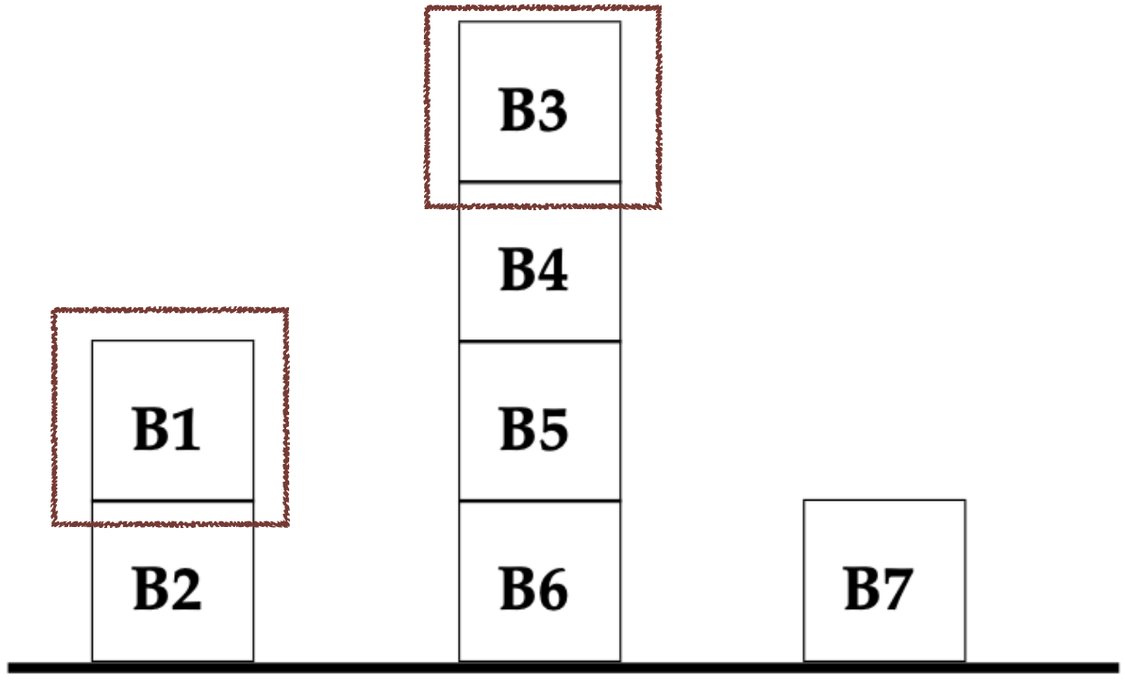
- Write clauses to handle the case *n* + 1
  - y is on a block z
    - `left(X, Y) :- on(Y, Z), left(X, Z).`




```
?- left(b2, b3).
```

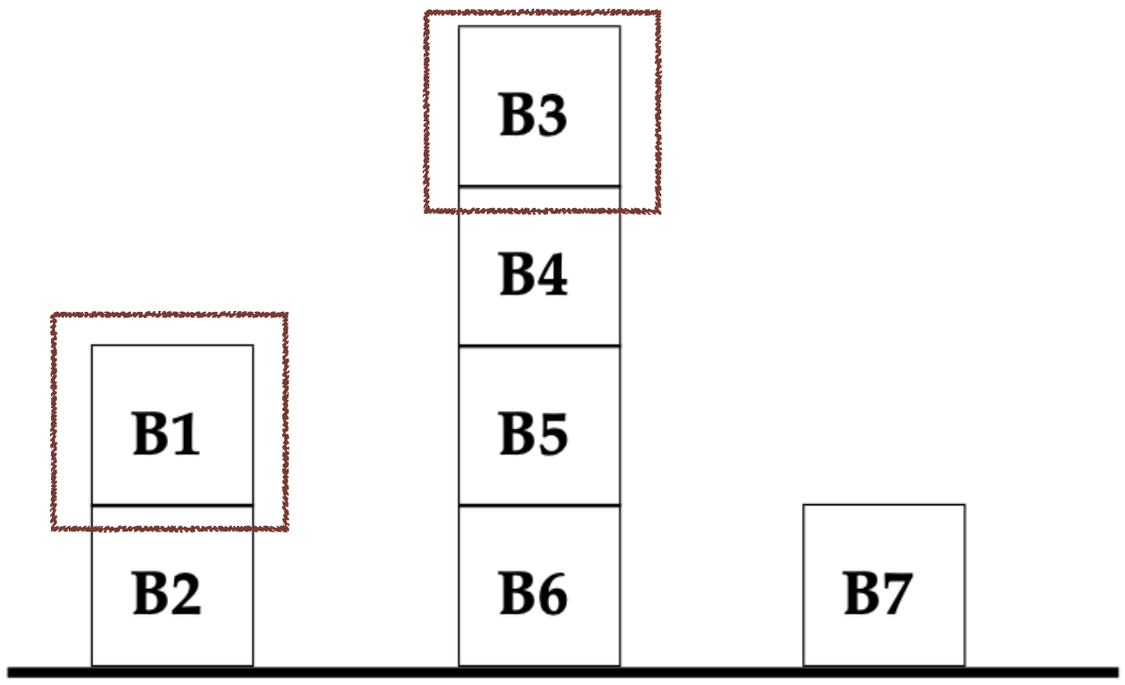
Handle the Cases in Recursion

```
left(X, Y) :- just_left(X, Y).
left(X, Y) :- just_left(X, Z), left(Z, Y).
left(X, Y) :- on(X, Z), left(Z, Y).
left(X, Y) :- on(Y, Z), left(X, Z).
```




```
?- left(b1, b3).
```

```
1.left(X, Y) :- just_left(X, Y).
2.left(X, Y) :- just_left(X, Z), left(Z, Y).
3.left(X, Y) :- on(X, Z), left(Z, Y).
4.left(X, Y) :- on(Y, Z), left(X, Z).
```




```
left(b1, b3) % Clause 3
left(b2, b3) % Clause 4
left(b2, b4) % Clause 4
left(b2, b5) % Clause 4
left(b2, b6) % Clause 1
true.
```

Non-terminating Cases

- The non-terminating cases occur when the size of *n* cannot be decreasing.
  - `above(X, Y) :- above(X, Y).`
- The correct version makes sure that Z is lower than X.
- The depth of above(Z, Y) is smaller than above(X, Y) is guaranteed!

```
above(X, Y) :- on(X, Y).
above(X, Y) :- on(X, Z), above(Z, Y).
```

```
above(X, Y) :- on(X, Y).
above(X, Y) :- above(Z, Y), on(X, Z).
```

Performance of Prolog Programs

- Similar to programming in general purpose languages, different implementation may lead to different complexities.

- The current implementation of left(X, Y) is relatively slow.

```
?- time(left(b3, b3)).
% 627 inferences, 0.000 CPU in 0.000 seconds (98% CPU, 28645833 Lips)
false.
```

```
left(X, Y) :- just_left(X, Y).
left(X, Y) :- just_left(X, Z), left(Z, Y).
left(X, Y) :- on(X, Z), left(Z, Y).
left(X, Y) :- on(Y, Z), left(X, Z).
```

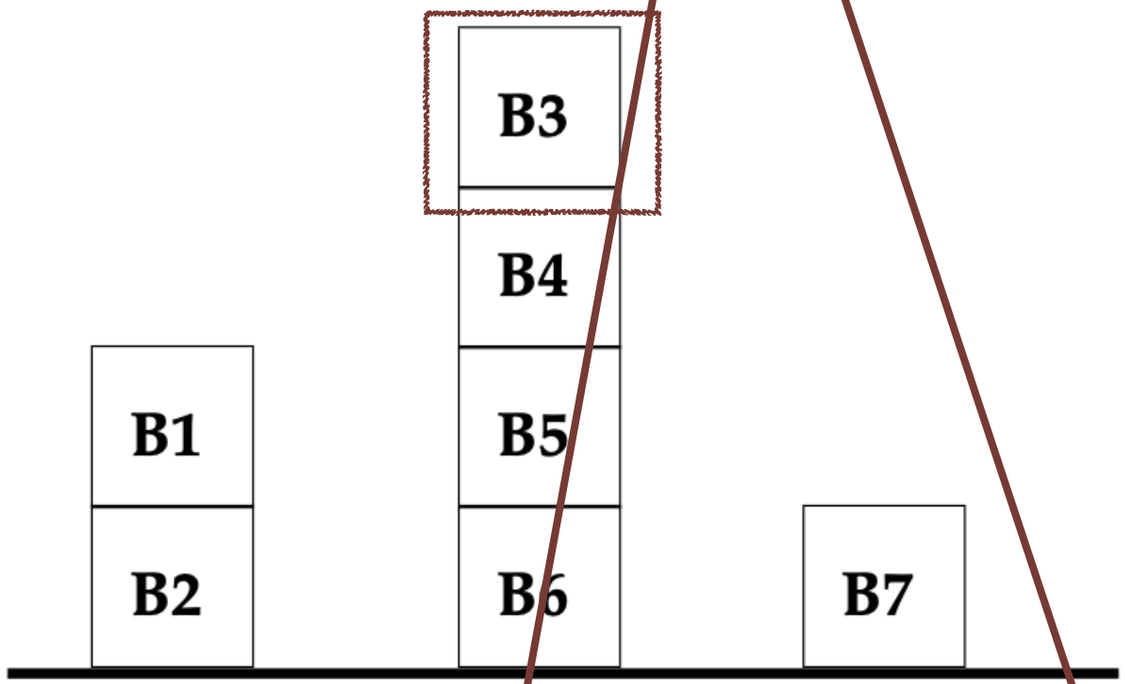
Inefficient Inference

- Why left(b3, b3) is so slow?
- The actually procedure is performed as follows.

```
def left(X, Y):
    if X and Y are both on the table then
        return just_left(X, Y)
    else if X is on another block Z then
        return left(Z, Y)
    else if Y is on another block Z then
        return left(X, Z)
```

- The problem is the same queries will be re-computed multiple times.

```
1.left(X, Y) :- just_left(X, Y).
2.left(X, Y) :- just_left(X, Z), left(Z, Y).
3.left(X, Y) :- on(X, Z), left(Z, Y).
4.left(X, Y) :- on(Y, Z), left(X, Z).
```




```
left(b3, b3) considers left(b4, b3) and left(b3, b4)
left(b4, b3) considers left(b5, b3) and left(b4, b4)
left(b3, b4) considers left(b4, b4) and left(b3, b5)
```

The Performance Issue

- The problem of the current implementation:
  - Both clauses 3 and 4 will be explored when neither X nor Y is directly on the table.
- 2 branches will be made.
- Requiring $O(2^k)$ for exploring the search space.

```
1.left(X, Y) :- just_left(X, Y).
2.left(X, Y) :- just_left(X, Z), left(Z, Y).
3.left(X, Y) :- on(X, Z), left(Z, Y).
4.left(X, Y) :- on(Y, Z), left(X, Z).
```

Recap of the Fibonacci Number Algorithm

```
int fibonacci(int n) {
    if (n == 0 || n == 1)
        return 1;
    return fibonacci(n-2) + fibonacci(n-1);
}
```

$$
\begin{aligned}
\text{Fibonacci}(5) &= \text{Fibonacci}(3) + \text{Fibonacci}(4) \\
&= (\text{Fibonacci}(1) + \text{Fibonacci}(2)) + (\text{Fibonacci}(2) + \text{Fibonacci}(3)) \\
&= (\text{Fibonacci}(1) + (\text{Fibonacci}(0) + \text{Fibonacci}(1))) + ((\text{Fibonacci}(0) + \text{Fibonacci}(1)) + (\text{Fibonacci}(1) + \text{Fibonacci}(2))) \\
&= (\text{Fibonacci}(1) + (\text{Fibonacci}(0) + \text{Fibonacci}(1))) + ((\text{Fibonacci}(0) + \text{Fibonacci}(1)) + (\text{Fibonacci}(1) + (\text{Fibonacci}(0) + \text{Fibonacci}(1))))
\end{aligned}
$$

Performance Issues in Prolog

- The growth of complexity is exponential $O(2^k)$
- Could be very slow for k more than 50 or even larger.
- More important point is that the complexity is unnecessary.
  - No reason to compute the same query more than once.
- In other words, the Prolog program has not been given how to answer the query *efficiently* but only *correctly*.

Enhancing the Performance of Back-chaining

- We can also design an algorithm for better performance in Prolog.

- For the case that neither X nor Y is directly on the table, we can find their bases separately.

- Reducing the interaction between independent subproblems.

- Avoiding the branch at all.

Enhancing the Performance of Back-chaining

- Explicitly force Prolog find XB and YB, the bottom blocks of both X and Y, respectively.

- Check if XB is left to YB.

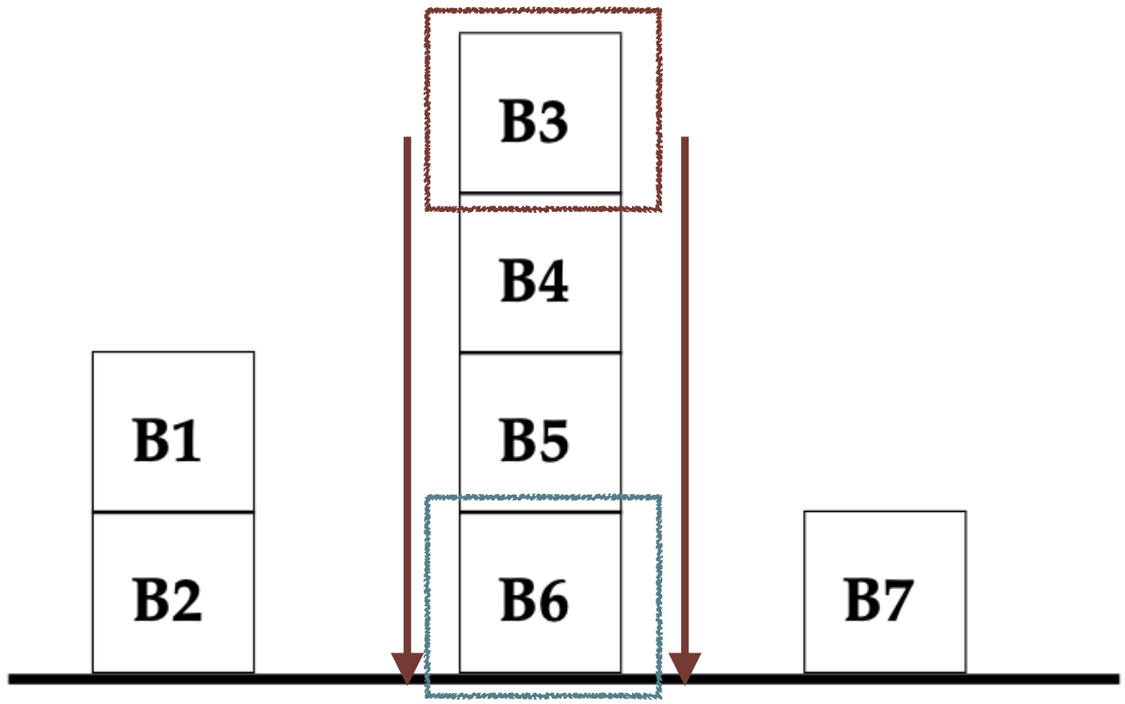
```
?- time(left(b3, b3)).
% 35 inferences, 0.000 CPU in 0.000 seconds (97% CPU, 6280280 Lips)
false.
```

```
left(X, Y) :- base(XB, X), base(YB, Y), table_left(XB, YB).

base(X, X) :- \+ on(X, Y).
base(Z, X) :- on(X, Y), base(Z, Y).

table_left(X, Y) :- just_left(X, Y).
table_left(X, Y) :- just_left(X, Z), table_left(Z, Y).
```

Enhancing the Performance of Back-chaining

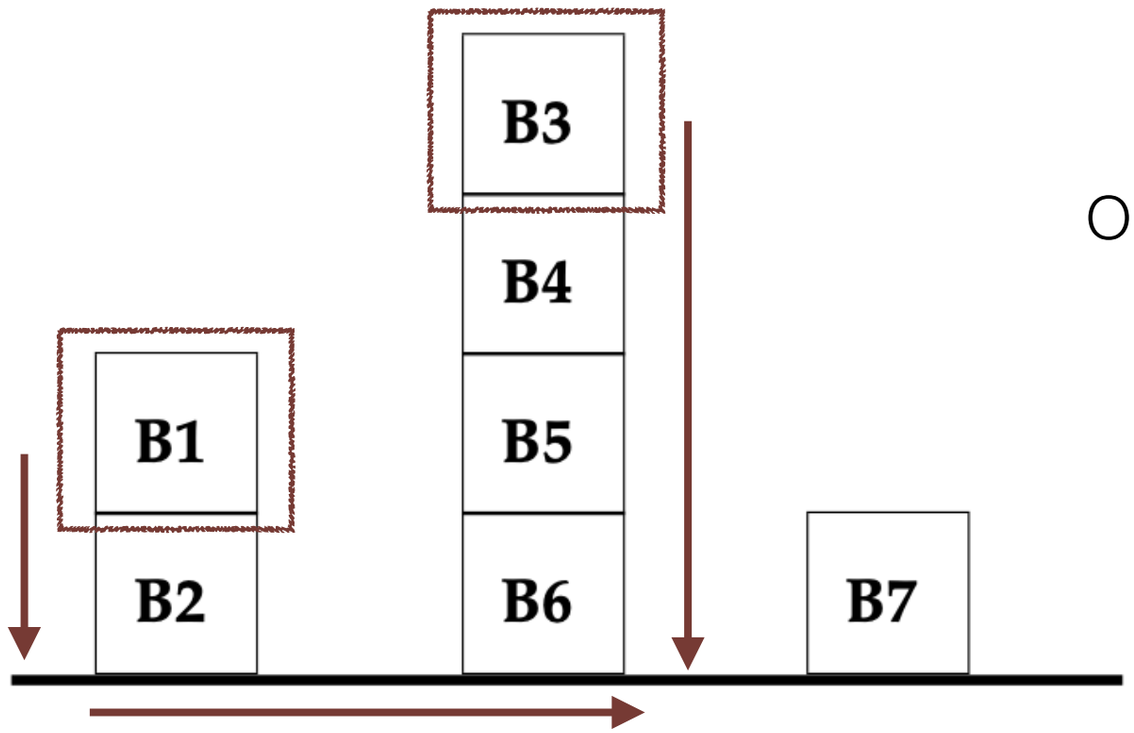



```
left(X, Y) :- base(XB, X), base(YB, Y), table_left(XB, YB).

base(X, X) :- \+ on(X, Y).
base(Z, X) :- on(X, Y), base(Z, Y).

table_left(X, Y) :- just_left(X, Y).
table_left(X, Y) :- just_left(X, Z), table_left(Z, Y).
```

Enhancing the Performance of Back-chaining



O(*k*) for k blocks

```
left(X, Y) :- base(XB, X), base(YB, Y), table_left(XB, YB).

base(X, X) :- \+ on(X, Y).
base(Z, X) :- on(X, Y), base(Z, Y).

table_left(X, Y) :- just_left(X, Y).
table_left(X, Y) :- just_left(X, Z), table_left(Z, Y).
```

Arithmetic in Prolog

- A Prolog term is either a constant, variable, or number.
- A number term
  - A sequence of one or more digits
  - Optionally
    - A minus sign
    - A decimal point

Numbers in Clauses

- Numbers can appear in programs and queries anywhere a constant or variable can appear.

```
age(tom, 21).
age(jane, 45).
current_temperature(28.5).

?- age(tom, N).
N = 21

?- age(X, N), \+ N=21.
X = jane, N = 45
```

Arithmetic Relations in Prolog

- Arithmetic expression
  - Formula built out numbers and variables using parentheses and operations +, -, *, /, **
- Comparison operators are defined in relations.

Relation	Symbol	Example
Less than	<	(X-Y)**2 < 3*(X/Y)
Greater than	>	5 > X+Y
Less than or equal	=<	3.14 =< X*Y
Greater than or equal	>=	8 >= X+Y+Z
Equal (two numbers)	=:=	0.5 =:= (3/6)
Equal (Variable)	is	X is 6/5

Arithmetic Expressions in Prolog

- Arithmetic expressions are not Prolog terms or literal.
  - Arithmetic expressions can only be used within Prolog programs and queries in the arithmetic relations.

Relation	Symbol	Example
Less than	<	(X-Y)**2 < 3*(X/Y)
Greater than	>	5 > X+Y
Less than or equal	=<	3.14 =< X*Y
Greater than or equal	>=	8 >= X+Y+Z
Equal (two numbers)	=:=	0.5 =:= (3/6)
Equal (Variable)	is	X is 6/5

Arithmetic Relations

- We can use arithmetic expressions in queries.

```
?- X = 3, (X+4) > 2 * X.
X = 3.

?- X = 3, (X+2) >= 2*X + 1.
false.
```

- Note that variables should be instantiated at the first.

```
?- X=4, X > 2.
X = 4.

?- X > 2, X = 4.
ERROR: Arguments are not sufficiently instantiated
```

The "is" Relation

- All variables should be instantiated at the first, except the the **is** relation.

- The relation will evaluate the arithmetic expression and get the value.

```
?- Y = 2, X = Y + 4.
Y = 2,
X = 2+4.

?- Y = 2, X is Y + 4.
Y = 2,
X = 6.
```

The "is" Relation

- Arithmetic expressions can be used as arguments of predicates.

- We can use the "is" relation to assign the value of an expression to a variable, and pass the variable to a predicate.

```
?- q(Y), p(Y+4, Z).
```

```
?- q(Y), X is Y + 4, p(X, Z).
```

Numeric Calculations as Arithmetic Inferences

```
birth_year(tom, 2001).
birth_year(jane, 1995).
current_year(2020).

age(P, X) :- birth_year(P, Y1),
current_year(Y2), X is Y2 - Y1.
```

```
?- age(tom, Y).
Y = 19.

?- age(jane, Y).
Y = 25.

?- age(john, Y).
false
```

Calculating Factorial

```
factorial(1, 1).
factorial(N, M) :- N > 1, N1 is N - 1, factorial(N1, T), M is T * N.
```

```
?- factorial(9, M).
M = 362880

?- factorial(20, M).
M = 2432902008176640000 .

?- factorial(200, M).
M =
788657867364790503552363213932185062295135977687173263294742533244
359449963403342920304284011984623904177212138919638830257642790242
637105061926624952829931113462857270763317237396988943922445621451
664240254033291864131227428294853277524242407573903240321257405579
568660226031904170324062351700858796178922227896237038973747200000
000000000000000000000000000000000000000000 .
```

Fibonacci Numbers

```
fibo(0, 1).
fibo(1, 1).
fibo(N, F) :- N > 1,
              N1 is N - 1,
              N2 is N - 2,
              fibo(N1, F1),
              fibo(N2, F2),
              F is F1 + F2.
```

```
?- fibo(1, X).
X = 1 .

?- fibo(2, X).
X = 2 .

?- fibo(3, X).
X = 3 .

?- fibo(4, X).
X = 5 .

?- fibo(5, X).
X = 8 .

?- fibo(6, X).
X = 13 .

?- fibo(20, X).
X = 10946 .
```

Fibonacci Numbers

- The recursive program can be very slow for a little larger value of *n*

```
?- time(fibo(20, X)).
% 43,780 inferences, 0.008 CPU in 0.008 seconds (100% CPU, 5705817 Lips)
X = 10946 .

?- time(fibo(50, X)).
...
```

$$
\begin{aligned}
\text{Fibonacci}(5) &= \text{Fibonacci}(3) + \text{Fibonacci}(4) \\
&= (\text{Fibonacci}(1) + \text{Fibonacci}(2)) + (\text{Fibonacci}(2) + \text{Fibonacci}(3)) \\
&= (\text{Fibonacci}(1) + (\text{Fibonacci}(0) + \text{Fibonacci}(1))) + ((\text{Fibonacci}(0) + \text{Fibonacci}(1)) + (\text{Fibonacci}(1) + \text{Fibonacci}(2))) \\
&= (\text{Fibonacci}(1) + (\text{Fibonacci}(0) + \text{Fibonacci}(1))) + ((\text{Fibonacci}(0) + \text{Fibonacci}(1)) + (\text{Fibonacci}(1) + (\text{Fibonacci}(0) + \text{Fibonacci}(1))))
\end{aligned}
$$

Dynamic Programming

```
int fibonacci(int n) {
    int table[n+1];
    table[0] = 1;
    table[1] = 1;
    for (i = 2; i <= n; ++i)
        table[i] = table[i - 2] + table[i - 1];
    return table[n];
}
```

$$\begin{aligned}\text{Fibonacci}(5) &= \text{Fibonacci}(3) + \text{Fibonacci}(4) \\ &= (\text{Fibonacci}(1) + \text{Fibonacci}(2)) + (\text{Fibonacci}(2) + \text{Fibonacci}(3)) \\ &= (\text{Fibonacci}(1) + (\text{Fibonacci}(0) + \text{Fibonacci}(1))) + (\text{Fibonacci}(2) + \text{Fibonacci}(3))\end{aligned}$$

Memoization

Memoization is dynamic programming in the top-down direction.

```
int fibonacci(int n) {
    if (table[n])
        return table[n];
    if (n == 0 || n == 1)
        table[n] = 1;
    else
        table[n] = fibonacci(n-2) + fibonacci(n-1);
    return table[n];
}
```

$$\begin{aligned}\text{Fibonacci}(5) &= \text{Fibonacci}(3) + \text{Fibonacci}(4) \\ &= (\text{Fibonacci}(1) + \text{Fibonacci}(2)) + (\text{Fibonacci}(2) + \text{Fibonacci}(3)) \\ &= (\text{Fibonacci}(1) + (\text{Fibonacci}(0) + \text{Fibonacci}(1))) + (\text{Fibonacci}(2) + \text{Fibonacci}(3))\end{aligned}$$

Table: Memoization in Prolog

```
:- table fibo/2.

fibo(0, 1).
fibo(1, 1).
fibo(N, F) :- N > 1,
              N1 is N - 1,
              N2 is N - 2,
              fibo(N1, F1),
              fibo(N2, F2),
              F is F1 + F2.
```

```
?- fibo(1, X).
X = 1 .

?- fibo(2, X).
X = 2 .

?- fibo(3, X).
X = 3 .

?- fibo(4, X).
X = 5 .

?- fibo(5, X).
X = 8 .

?- fibo(6, X).
X = 13 .

?- fibo(20, X).
X = 10946 .
```

From O(2^n) to O(n)

```
?- time(fibo(20, X)).
% 43,780 inferences, 0.008 CPU in 0.008 seconds (100% CPU, 5705817 Lips)
X = 10946 .

?- time(fibo(50, X)).
...
```

```
?- time(fibo(20, X)).
% 1,059 inferences, 0.000 CPU in 0.000 seconds (100% CPU, 7480663 Lips)
X = 10946.

?- time(fibo(50, X)).
% 1,554 inferences, 0.001 CPU in 0.001 seconds (100% CPU, 1401241 Lips)
X = 20365011074.
```

Table Execution

- Tabling a predicate provides two properties:
  - Re-evaluation of a tabled predicate is avoided by memoizing the answers.
  - Avoiding Infinite loop
    - a goal calling a variant of itself recursively

```
male(john).
male(tom).
female(jane).
opp_sex(X, Y) :- male(X), female(Y).
opp_sex(X, Y) :- opp_sex(Y, X).
```

```
[trace]  ?- opp_sex(john, tom).
   Call: (8) opp_sex(john, tom) ? creep
   Call: (9) male(john) ? creep
   Exit: (9) male(john) ? creep
   Call: (9) female(tom) ? creep
   Fail: (9) female(tom) ? creep
   Redo: (8) opp_sex(john, tom) ? creep
   Call: (9) opp_sex(tom, john) ? creep
   Call: (10) male(tom) ? creep
   Exit: (10) male(tom) ? creep
   Call: (10) female(john) ? creep
   Fail: (10) female(john) ? creep
   Redo: (9) opp_sex(tom, john) ? creep
   Call: (10) opp_sex(john, tom) ? creep
...(Infinite ...)
```

Prevent from Infinite Loop with Tabling

- A sm

```
:- table opp_sex/2.

male(john).
male(tom).
female(jane).
opp_sex(X, Y) :- male(X), female(Y).
opp_sex(X, Y) :- opp_sex(Y, X).
```

```
?- time(opp_sex(john, tom)).
% 6 inferences, 0.000 CPU in 0.000 seconds (90% CPU, 1326847 Lips)
false.

?- time(opp_sex(tom, john)).
% 5 inferences, 0.000 CPU in 0.000 seconds (87% CPU, 219500 Lips)
false.
```

Drawback of Table Execution

- Why does Prolog not always perform tabling?
- Tabling comes with two downsides
  - The memoized answers are not automatically **updated** or **invalidated** if the world changes and the answer tables must be stored (in memory).

Lists in Prolog

- Manipulate multiple variables in a consistent way.

```
uniq_person3(X, Y, Z) :- person(X), person(Y), person(Z),
                         \+ X = Y, \+ X = Z, \+ Y = Z.
```

- Can be very annoying for the cases with a larger number.
  - uniq_persons5(X, Y, Z, U, V) :-
  - ...
  - **uniq_persons100(X, Y, Z, ...) :-**
- We want to have a predicate for arbitrary numbers
  - uniq_people(L) :- ...

Lists in Prolog

- In Prolog, a list is a sequence of objects (elements)
  - [tom, mary, jane, john]
  - [a, b, c, d, e]
  - [1, 2, 4, 8, 16]
  - []
  - [[tom, 95], [jane, 96], [mary, 90], [john, 91]]
- The first element of a list is an object, **head**
- **T**he rest is a list, **tail.**

Lists as Prolog Terms

- A Prolog term is a constant, variable, number, or list.
- Define a list in Prolog focused on elements
  - `[1, 3, a, john, "hello world", [k, i, j], 55, []]`
- Define a list in Prolog focused on the structure
  - In the form $[item_1, item_2, ... | tail]$ ← **a list**
  - `[1, X, 3] = [1|[X, 3]]  = [1,X|[3]] = [1,X,3|[]]`

Unification with Lists

- Like a single variable, unification will also performed on lists.
    - Two lists without variables considered to unify when they are identical.
    - Two lists with distinct variables are considered to unify when the variables can be valuation that makes the two lists identical.

Unification with Lists

```
[] and []
[a,b,c] and [a,b,c]
[X] and [a] with X=a
[a,b,X] and [Y,b,Y] with X=a, Y=a
[X,X] and [[Y,a,c],[b,a,c]] with X=[b,a,c], Y=b
[[X]] and [Y] with X=_G237, Y=[_G237]

[a,b,c] and [a|[b,c]]
[a,b,c] and [a|[b|[c]]]
[X|Y] and [a] with X=a, Y=[]
[a|[b,c]] and [a,X,Y] with X=b, Y=c
[a,b,c] and [X|Y] with X=a, Y=[b,c]
[X,Y|Y] and [a,[b,c],b,c] with X=a, Y=[b,c]
```

The Cases that Cannot Be Unified

```
[a] and []
[a,b,c] and [a,a,c]
[] and [[]]
[X,Y] and [U,V,W]
[a,b,c] and [X,b,X]
[X|Y] and []
```

Uses of Lists

- To check all elements are person

```
?- person_list([john,sue,george,harry]).
true .
```

```
?- person_list([john,5,harry]).
false .
```

```
person_list([H|T]) :- person(H), person_list(T).
```

Uses of Lists

- To check an object is in a list

```
?- elem(john, [john,sue,george,harry]).
true .
```

```
?- elem(tom, [john,sue,george,harry]).
false .
```

```
elem(X, [X|_]).
elem(X, [_|T]) :- elem(X, T).
```

No need to deal with elem(X, []) since it is always false.

Uses of Lists

- Now we can define a predicate for checking uniq_people

```
?- uniq_people([john,sue,george,harry]).
true .

?- uniq_people([harry,sue,george,harry]).
false .
```

```
uniq_people([]).
uniq_people([H|T]) :- uniq_people(T),
                      person(H),
                      \+ elem(H, T).
```

Now we have to define the empty case because it is always true.