Chapter 15

Logic Programming Languages

15.1 Introduction
15.2 A Brief Introduction to Predicate Calculus
15.3 Predicate Calculus and Proving Theorem
15.4 An Overview of Logic Programming
15.5 The Origins of Prolog
15.6 The Basic Elements of Prolog
15.7 Deficiencies of Prolog
15.8 Applications of Logic Programming
15.9 Conclusion

Express the program in a form of symbolic logic and use a logical inferencing process to produce the results.

Logic programs are declarative rather than procedural, which means that only the specifications of the desired results are stated rather than detailed procedures for producing them

http://swish.swi-prolog.org/

15.1 Introduction

- Logic Programming
  - Express the program in a form of symbolic logic and use a logical inferencing process to produce the results.
  - Logic programs are *declarative* rather than *procedural*, which means that only the specifications of the desired results are stated rather than detailed procedures for producing them

procedural sorting program (how?)

```
for (i = n-1 ; i <= 1 ; i--) {
     for (j = 1; j < i ; j++) {
         if (A[j]<A[j+1]) swap(A[j],A[j+1])}}
```

```
:- sort([5, 6, 3, 100], New_list)
New_list = [3, 5, 6, 100]
```

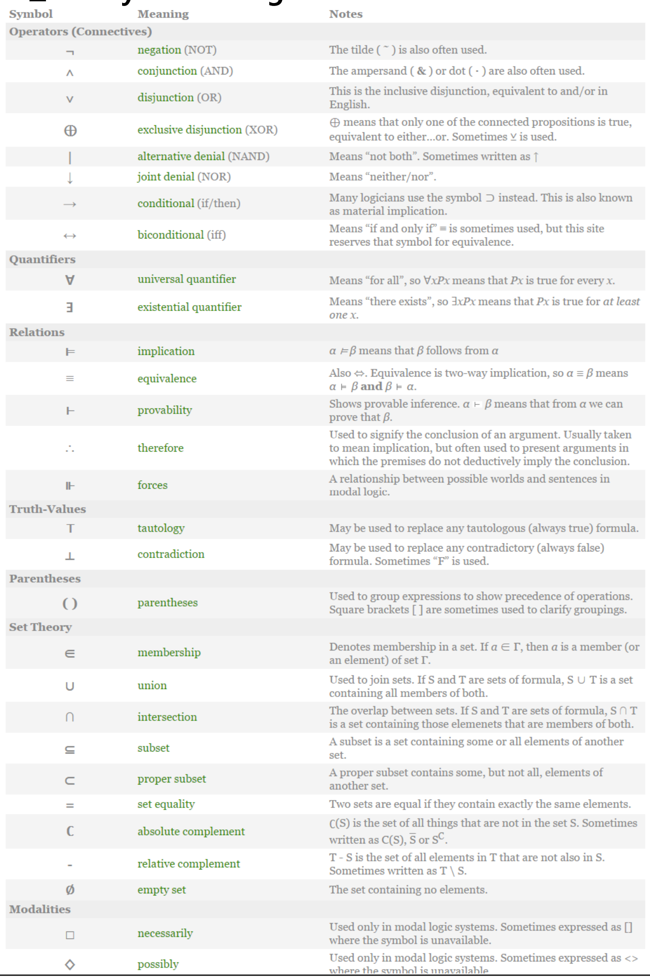
declarative sorting program (what?)

```
sort(old_list, new_list) ⊂ permute(old_list, new_list) ∩ sorted(new_list)
sorted(list) ⊂ ∀j such that 1 ≤ j < n, list(j) ≤ list(j+1)
```

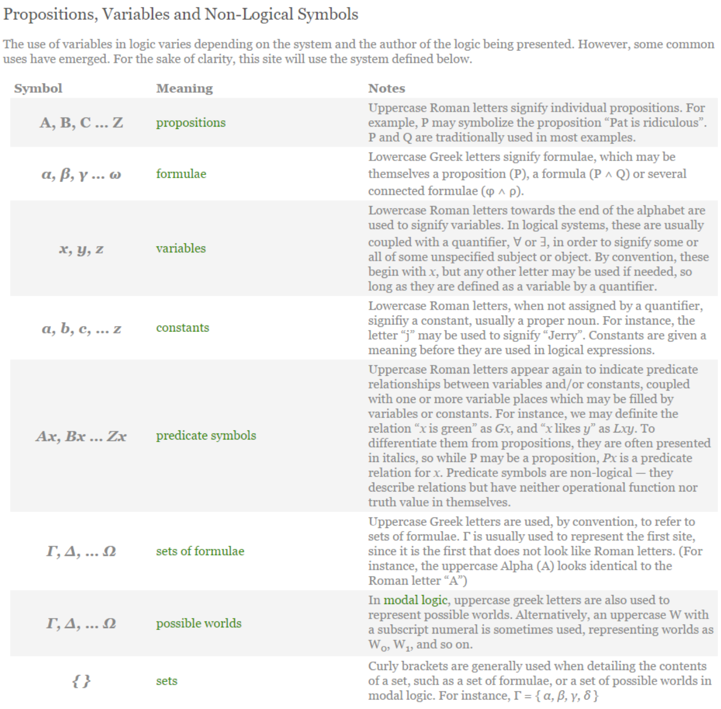
  - Programming that uses a form of symbolic logic as a programming language is often called ***logic programming***
  - Languages based on symbolic logic are called logic programming languages or declarative languages

Symbolic logic is ***the method of representing logical expressions*** through the use of symbols and variables, rather than in ordinary language. ➔ This has the benefit of removing the ambiguity that normally accompanies ordinary languages, such as English, and allows easier operation. (http://www.philosophy-index.com/logic/symbolic/)

- <참고> symbolic Logic



Symbol	Meaning	Notes
Operators (Connectives)		
¬	negation (NOT)	The tilde (˜) is also often used.
∧	conjunction (AND)	The ampersand (&) or dot (·) are also often used.
∨	disjunction (OR)	This is the inclusive disjunction, equivalent to and/or in English.
⊕	exclusive disjunction (XOR)	⊕ means that only one of the connected propositions is true, equivalent to either…or. Sometimes ⊻ is used.
|	alternative denial (NAND)	Means "not both". Sometimes written as ↑
↓	joint denial (NOR)	Means "neither/nor".
→	conditional (if/then)	Many logicians use the symbol ⊃ instead. This is also known as material implication.
↔	biconditional (iff)	Means "if and only if" ≡ is sometimes used, but this site reserves that symbol for equivalence.
Quantifiers		
∀	universal quantifier	Means "for all", so $\forall xPx$ means that Px is true for every x.
∃	existential quantifier	Means "there exists", so $\exists xPx$ means that Px is true for *at least one* x.
Relations		
⊨	implication	$\alpha \models \beta$ means that β follows from α
≡	equivalence	Also ⇔. Equivalence is two-way implication, so $\alpha \equiv \beta$ means $\alpha \models \beta$ **and** $\beta \models \alpha$.
⊢	provability	Shows provable inference. $\alpha \vdash \beta$ means that from α we can prove that β.
∴	therefore	Used to signify the conclusion of an argument. Usually taken to mean implication, but often used to present arguments in which the premises do not deductively imply the conclusion.
⊩	forces	A relationship between possible worlds and sentences in modal logic.
Truth-Values		
⊤	tautology	May be used to replace any tautologous (always true) formula.
⊥	contradiction	May be used to replace any contradictory (always false) formula. Sometimes "F" is used.
Parentheses		
( )	parentheses	Used to group expressions to show precedence of operations. Square brackets [ ] are sometimes used to clarify groupings.
Set Theory		
∈	membership	Denotes membership in a set. If $\alpha \in \Gamma$, then α is a member (or an element) of set Γ.
∪	union	Used to join sets. If S and T are sets of formula, S ∪ T is a set containing all members of both.
∩	intersection	The overlap between sets. If S and T are sets of formula, S ∩ T is a set containing those elemenets that are members of both.
⊆	subset	A subset is a set containing some or all elements of another set.
⊂	proper subset	A proper subset contains some, but not all, elements of another set.
=	set equality	Two sets are equal if they contain exactly the same elements.
∁	absolute complement	∁(S) is the set of all things that are not in the set S. Sometimes written as C(S), $\overline{S}$ or S^C.
-	relative complement	T - S is the set of all elements in T that are not also in S. Sometimes written as T \ S.
∅	empty set	The set containing no elements.
Modalities		
□	necessarily	Used only in modal logic systems. Sometimes expressed as [] where the symbol is unavailable.
◇	possibly	Used only in modal logic systems. Sometimes expressed as <> where the symbol is unavailable.



Propositions, Variables and Non-Logical Symbols

The use of variables in logic varies depending on the system and the author of the logic being presented. However, some common uses have emerged. For the sake of clarity, this site will use the system defined below.

Symbol	Meaning	Notes
A, B, C … Z	propositions	Uppercase Roman letters signify individual propositions. For example, P may symbolize the proposition "Pat is ridiculous". P and Q are traditionally used in most examples.
α, β, γ … ω	formulae	Lowercase Greek letters signify formulae, which may be themselves a proposition (P), a formula (P ∧ Q) or several connected formulae (φ ∧ ρ).
x, y, z	variables	Lowercase Roman letters towards the end of the alphabet are used to signify variables. In logical systems, these are usually coupled with a quantifier, ∀ or ∃, in order to signify some or all of some unspecified subject or object. By convention, these begin with x, but any other letter may be used if needed, so long as they are defined as a variable by a quantifier.
a, b, c, … z	constants	Lowercase Roman letters, when not assigned by a quantifier, signifiy a constant, usually a proper noun. For instance, the letter "j" may be used to signify "Jerry". Constants are given a meaning before they are used in logical expressions.
Ax, Bx … Zx	predicate symbols	Uppercase Roman letters appear again to indicate predicate relationships between variables and/or constants, coupled with one or more variable places which may be filled by variables or constants. For instance, we may definite the relation "x is green" as Gx, and "x likes y" as Lxy. To differentiate them from propositions, they are often presented in italics, so while P may be a proposition, Px is a predicate relation for x. Predicate symbols are non-logical — they describe relations but have neither operational function nor truth value in themselves.
Γ, Δ, … Ω	sets of formulae	Uppercase Greek letters are used, by convention, to refer to sets of formulae. Γ is usually used to represent the first site, since it is the first that does not look like Roman letters. (For instance, the uppercase Alpha (A) looks identical to the Roman letter "A")
Γ, Δ, … Ω	possible worlds	In modal logic, uppercase greek letters are also used to represent possible worlds. Alternatively, an uppercase W with a subscript numeral is sometimes used, representing worlds as W_0, W_1, and so on.
{ }	sets	Curly brackets are generally used when detailing the contents of a set, such as a set of formulae, or a set of possible worlds in modal logic. For instance, Γ = { α, β, γ, δ }

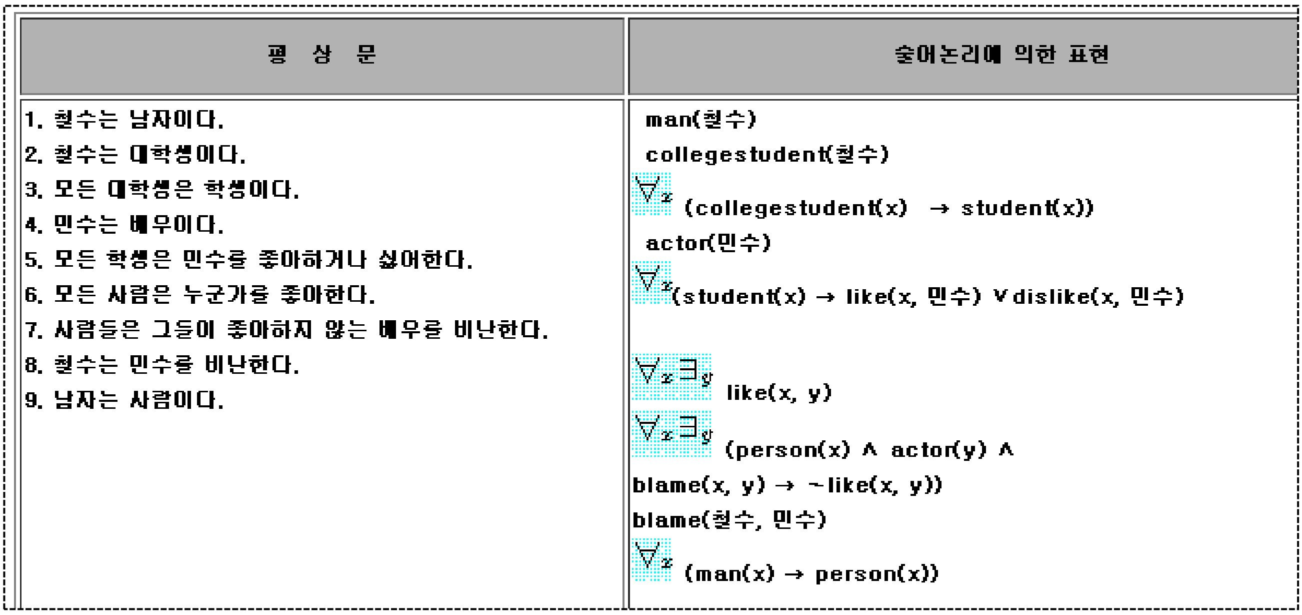
15.2 A Brief Introduction to Predicate Calculus

- **Terms**
  - ***proposition*** (명제)

    어떤 문제에 대한 하나의 **논리적 판단 내용과 주장**을 **언어 또는 기호로 표시**한 것.

    - ⇔ a logical statement that may or may not be true
    - ⇔ made up of objects and their relationships to each other
    - ⇔ 예) `father(bob, jake)`
  - ***formal logic***
    - ⇔ it was developed to provide a method for describing propositions, with the goal of allowing those formally stated propositions to be checked for validity
  - ***symbolic logic*** can be used for the three basic needs of formal logic
    - ⇔ to express propositions
    - ⇔ to express the relationships between propositions
    - ⇔ to describe how new propositions can be inferred from other propositions that are assumed to be true
  - the particular form of symbolic logic that is used for logic programming is called ***predicate calculus***

- <참고> Predicate Calculus (Predicate logic, 술어 논리)



평상문	술어논리에 의한 표현
1. 철수는 남자이다.	man(철수)
2. 철수는 대학생이다.	collegestudent(철수)
3. 모든 대학생은 학생이다.	$\forall_x$ (collegestudent(x) → student(x))
4. 민수는 배우이다.	actor(민수)
5. 모든 학생은 민수를 좋아하거나 싫어한다.	$\forall_x$(student(x) → like(x, 민수) ∨ dislike(x, 민수))
6. 모든 사람은 누군가를 좋아한다.	$\forall_x \exists_y$ like(x, y)
7. 사람들은 그들이 좋아하지 않는 배우를 비난한다.	$\forall_x \exists_y$ (person(x) ∧ actor(y) ∧ blame(x, y) → ~like(x, y))
8. 철수는 민수를 비난한다.	blame(철수, 민수)
9. 남자는 사람이다.	$\forall_x$ (man(x) → person(x))

- "철수는 민수를 좋아하는가?" : ~like(철수, 민수) ??

(1) Proposition

– the **objects** in logic programming propositions are represented by **simple terms**, which are either ***constants*** or ***variables***

⇔ **constant** : a symbol that represents an object

⇔ **variable** : a symbol that can represent different object at different times

⇒ **unknown value** (logic language) *vs.* **memory cell** (imperative language)

- **Atomic Proposition** (Relation, Predicate)

– the simplest proposition

– represents the relationship between objects

```
man(jake)
man(fred)
like(bob,redheads)
like(fred,X)
```

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

append		
X	Y	Z
[ ]	[ ]	[ ]
[a]	[ ]	[a]
.....	.....	.....
[a,b]	[c,d]	[a,b,c,d]
....	......	......

- **Compound Proposition**
  - have two or more atomic propositions, which are connected by logical connectors

Name	Symbol	Example	Meaning
negation	~	~a	not a
conjunction(논리곱)	$\cap$	a $\cap$ b	a and b
disjunction (논리합)	$\cup$	a $\cup$ b	a or b
equivalence (동가)	$\equiv$	a $\equiv$ b	a is equivalent to b
implication	$\supset$	a $\supset$ b	a implies b
	$\subset$	a $\subset$ b	b implies a

  - variables can appear in propositions but only when introduced by special symbols called quantifiers
    - ⇔ example
      - ⇒ $\forall$X. (woman(X) $\supset$ human(X))
        - → for any value of X, if X is a woman, then X is a human
      - ⇒ $\exists$X. (mother(mary,X) $\cap$ male(X))
        - → there exists a value of X such that mary is the mother of X and X is male (*i.e.* mary has a son)

(2) Clausal Form

- **Clausal Form**
  - a relatively simple form of propositions (standard)
    - ⇔ *disjunction* on the left side and conjunction on the right side
  - all propositions can be expressed in clausal form
  - syntax (A, B : compound term)

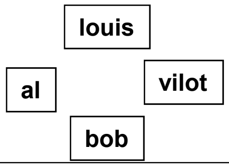
$$B_1 \cup B_2 \cup \ldots\ldots \cup B_n \subset A_1 \cap A_2 \cap \ldots\ldots \cap A_n$$

  - ⇔ meaning : *if all of the As are true, then at least one B is true*
  - ⇔ universal quantifiers are implicit in the use of variables in the atomic propositions

  - example of clausal form proposition

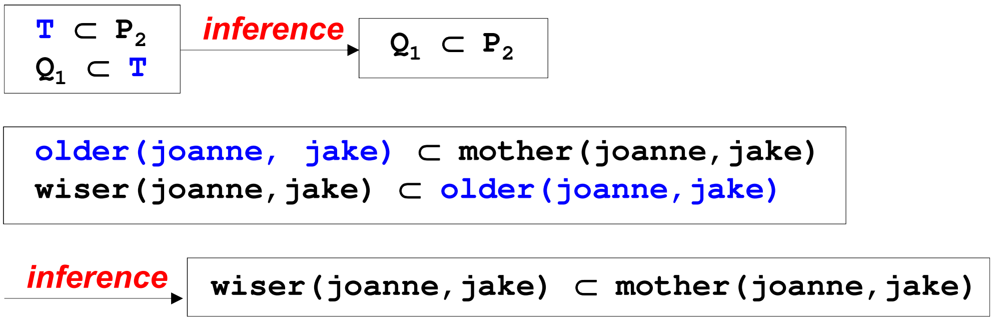
```
likes(bob, mary) ⊂ likes(bob, redhead) ∩ redhead(mary).
```

```
father(louis,al) ∪ father(louis,violet) ⊂
         father(al,bob) ∩ mother(violet,bob) ∩ grandfather(louis, bob)
```



15.3 Predicate Calculus and Proving Theorem

- Resolution
  - automatic theorem proving method proposed by Alan Robinson in 1965.
  - an inference rule that allows inferred propositions to be computed from given propositions in clausal form
  - Rule : ***If we have two clausal forms, and we can match the head of the first clause with one of the statements in the body of the second clause, then first clause can be used to replace its head in the second clause by its body***
  - a critically important property of resolution is its ability to detect any inconsistency in a given set of propositions
    - ⇔ the theorem is negated so that resolution can be used to prove the theorem by finding a inconsistency : *proof by contradiction*
  - examples



```
T ⊂ P2
Q1 ⊂ T
```

inference

```
Q1 ⊂ P2
```

```
older(joanne, jake) ⊂ mother(joanne,jake)
wiser(joanne,jake) ⊂ older(joanne,jake)
```

inference

```
wiser(joanne,jake) ⊂ mother(joanne,jake)
```

- the presence of variables in propositions requires resolution to find values for those variables that allow the matching process to succeed
  - ⇔ this process of determining useful values for variable is called, *unification*

- Horn Clause
  - a restricted kind of clausal form to simplify the resolution process
  - could be only two forms
    - ⇔ a single atomic proposition on the left side

```
likes(bob, mary) ⊂ likes(bob, redhead) ∩ redhead(mary).
```

    - ⇔ an empty left side

```
man(jake).
man(fred).
like(bob,redheads).
like(fred,X).
```

15.4 An Overview of Logic Programming

- ***Procedural Language*** **(imperative languages)**
  - **the programmer knows *what* is to be accomplished by a program and instructs the computer on exactly *how* the computation is to be done**
  - **the computer is treated as a simple device that obeys orders**
  - **everything that is computed must have *every detail of that computation spelled out***

- ***Declarative Language*** **(logic languages)**
  - **programs consist of *declarations* rather than assignments and control flow statements**
  - ***do not state exactly how a result is to be computed*, but rather describe the form of result**
  - **we assume that the computer system can somehow determine how a result is to be gotten**
  - **algorithm = logic + control**

- **Example : Sorting**
  - **Procedural Languages**
    - ⇔ **sorting is done by explaining in a Pascal program *all the details of some sorting algorithm* to a computer that has a Pascal compiler**
    - ⇔ **the computer, after translating the Pascal program into machine code or some interpretive intermediate code, follows the instructions and produces the sorted list**

procedural sorting program (how?)

```
for (i = n-1 ; i <= 1 ; i--) {
    for (j = 1; j < i ; j++) {
        if (A[j]<A[j+1]) swap(A[j],A[j+1]);
    }
 }
```

  - **Declarative Language**
    - ⇔ **It is necessary only *to describe the characteristics of the sorted list***
    - ⇔ **software requirements specifications**
      - ⇒ **It is some permutation of the given list such that for each pair of adjacent list, a given relationship holds between the two elements**

declarative sorting program (what?)

```
sort(old_list, new_list) ⊂ permute(old_list, new_list) ∩ sorted(new_list)
sorted(list) ⊂ ∀j such that 1 ≤ j < n, list(j) ≤ list(j+1)
```

15.5 The Origins of Prolog

- Prolog
  - Originally developed by Alain Colmerauer, Phillippe Roussel, Robert Kowalski
  - Japanese Fifth Generation Computing System (FGCS) Project in 1981
    - ⇔ tried to develop an intelligent computer system, and Prolog was chosen as the basis for this efforts
  - a logic programming language whose syntax is modified version of predicate calculus
  - its inferencing method is a restricted form of resolution
  - Warren Abstract Machine

15.6 The Basic Elements of Prolog

- Term
  - a constant
    - ⇔ atom : begins with ***lower case letter***
    - ⇔ integer
  - a variable
    - ⇔ begins with ***upper case letter***
    - ⇔ instantiation : the binding of a value to variable
  - a structure
    - ⇔ `functor(parameter_list)`

- Fact
  - unconditional assertion or fact
  - example

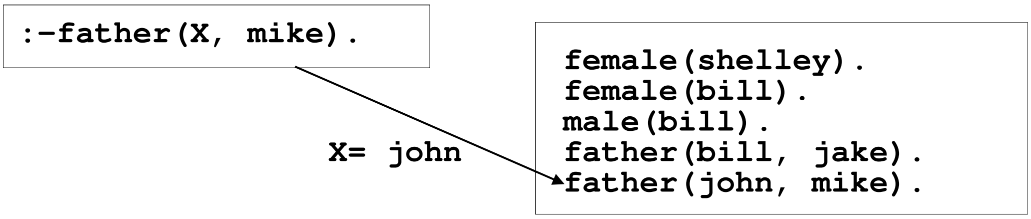
```
female(shelley).
female(bill).
male(bill).
father(bill, jake).
father(bill, shelley).
```

- **Rule**
  - Headed Horn Clause
  - describes the logical relationships among facts
  - in Prolog, AND operation is implied in clause body
  - syntax
    - ⇔ `structure_1` can be concluded if the `antecedent expression` is true or can be made to be true by some instantiation of its variables

```
structure_1  :- antecedent_expression
```

$A :- B_1 \cap B_2 \cap B_3 \cap \ldots B_n$

- **Goal**
  - *a proposition that we want the system to either prove or disprove*
  - when *variables are present*, the system not only asserts the validity of the goal but also identifies *the instantiations of variables* that make the goal true



```
:-father(X, mike).
```

X= john

```
female(shelley).
female(bill).
male(bill).
father(bill, jake).
father(john, mike).
```

- **Unification** (a two-way parameter passing)
  - a unifier of two terms is a substitution making the terms identical
  - a *mgu* (*most general unifier*) if two terms is a unifier such that the associated common instance is most general
    - ⇔ a term *s* is more general than a term *t* if t is an instance of *s*, but *s* is not an instance of *t*

```
aa :- …, P(XXX), …. .

P(YYY) :- …….
```

XXX	YYY	mgu
X	a	X=a
f(Y,9)	f(a,X)	Y=a, X=9
[H|T]	[1,2,3,4]	H=1,T=[2,3,4]
f(X,Y,Z)	f(a,A,B)	X=a, Y=A, Z=B
[1|X]	[2,3]	fail
X	s(X)	X=S(S(S..(X))

- **Proving a Goal** (matching)
  - to prove that a goal is true, the inferencing process must ***find a chain of inference rules and/or facts in the database*** that connect the goal to one or more facts in the database
    - ⇔ ***forward chaining*** : ***begins with the facts and rules of the database*** and attempt to find a sequence of matches that lead to the goal
    - ⇔ ***backward chaining*** : ***begins with the goal*** and attempt to find a sequence of matching propositions that lead to some set of original facts in the database
    - ⇔ Example

```
father(bob).
man(X) :- father(X).

:- man(bob).
```

  - whether the solution search is done depth first search or breadth first search
    - ⇔ ***depth first search*** : finds a complete sequence of proposition - a proof - for the first subgoal before working on other
    - ⇔ ***breadth first search*** : works on all subgoals of given goal in parallel
    - ⇔ Example

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

- **backtracking**
  - ⇔ when a goal with multiple subgoals is being processed and the system fails to show the truth of one of the subgoals, ***the system reconsiders the previous subgoals***
  - ⇔ a new solution is found by beginning the search where the previous search for that subgoal stopped
  - ⇔ Example

```
parent(X,Y) :- mother(X,Y).
parent(X,Y) :- father(X,Y).
grandparent(X,Y) :- parent(X,Z),parent(Z,Y).
sibling(X,Y) :- mother(M,X), mother(M,Y),
                father(F,X), father(F,Y).
mother(soonja, dongsoo).
mother(soonja, soonsoo).
mother(heeja, soonja).
father(doowhan, dongsoo).
father(doowhat, soonsoo).
```

Database of Facts and Ruls

```
:- parent(soonja, dongsoo).
yes

:- parent(soonja, X).
X = dongsoo ;
X = soonsoo ;
no
```

Query to DB and Results

left-to-right, depth-first order of evaluation !

- **Simple Arithmetic**
  - Prolog supports integer variables and integer arithmetic
  - Example 1

```
:- X is Y / 17 + Z.
:- Sum is Sum + Number.   (never useful !)
```

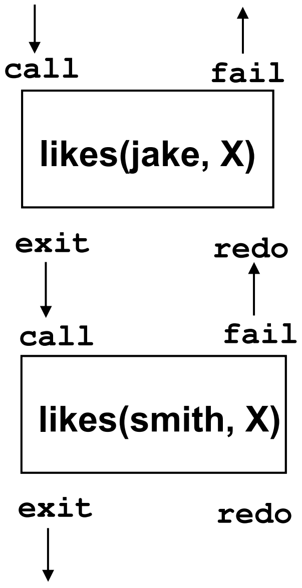
```
speed(ford, 100).
speed(pony, 105).
speed(pride, 95).
time(ford, 20).
time(pony, 21).
time(pride, 24).
distance(X,Y) :- speed(X, Speed),
                 time(X, Time),
                 Y is Speed * Time.
```

```
:- distance(pony, X).
X = 2205
yes
```

```
:- trace.
:- distance(pony, X).
(1)  1 Call  : distance(pony, _0) ?
(2)  2 Call  : speed(pony, _5) ?
(2)  2 Exit  : speed(pony, 105)
(3)  2 Call  : time(pony, _6) ?
(3)  2 Exit  : time(pony, 21)
(4)  2 Call  : _0 is 105*21 ?
(4)  Exit  : 2205 is 105*21
(1)  Exit  : distance(pony, 2205)
X = 2205
```

- Example 2

```
likes(jake, chocolate).
likes(jake, orange).
likes(smith, apple).
likes(smith, orange).
```



```
:- trace.
:- likes(jakes, X), likes(smith, X).
(1)  1 Call  : likes(jake, _0) ?
(1)  1 Exit  : likes(jake, chocolate)
(2)  1 Call  : likes(smith, chocolate) ?
(2)  1 Fail  : likes(smith, chocolate)
(1)  1 Back to : likes(jake, _0) ?
(1)  1 Exit  : likes(jake, orange)
(3)  1 Call  : likes(smith, orange) ?
(3)  1 Exit  : likes(smith, orange)
X = orange
yes
```

- **Example 2**

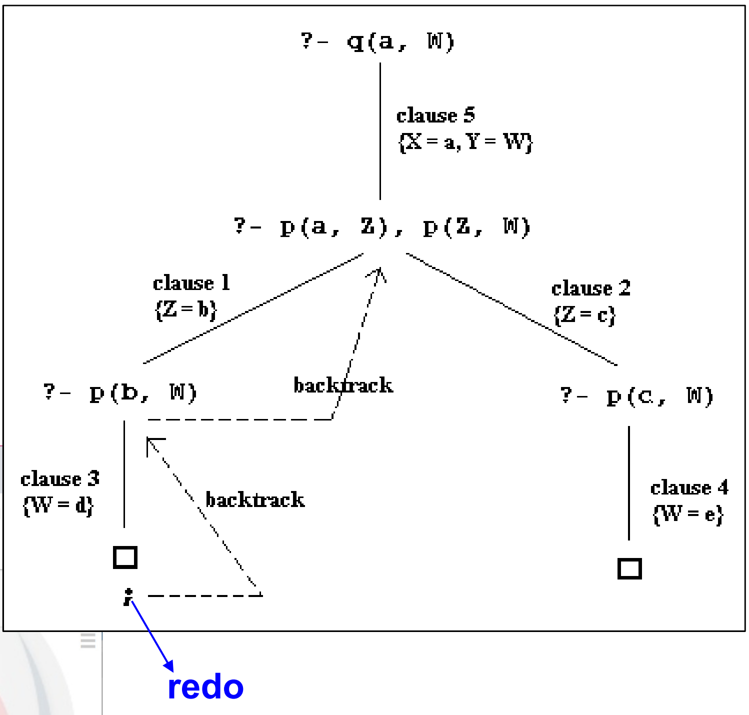
```
append(A, A, [ ]).
append([A|L1], L2, [A|L3]) :- append(L1, L2, L3).

reverse([ ], [ ]).
reverse([H|T], L) :-  reverse(L, L1), append(L1, [H], L).

member(Element, [Element | _]).
member(Element, [ _ | List) :- member(Element, List).
```

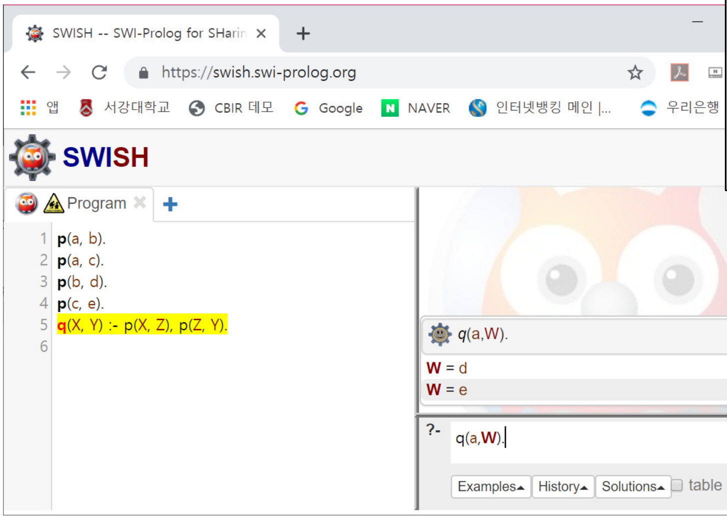
```
:- trace.
:- member(a, [b, c, d]).
(1) 1 Call : member(a, [b, c, d]) ?
(2) 2 Call : member(a, [c, d]) ?
(3) 3 Call : member(a, [d]) ?
(4) 4 Call : member(a, [ ]) ?
(4) 4 Fail : member(a, [ ])
(3) 3 Fail : member(a, [d])
(2) 2 Fail : member(a, [c, d])
(1) 1 Fail : member(a, [b, c, d])
no

:- member(a, [b, a, c]).
(1) 1 Call : member(a, [b, a, c]) ?
(2) 2 Call : member(a, [a, c]) ?
(2) 2 Exit : member(a, [a, c])
(1) 1 Exit : member(a, [b, a, c])
yes
```

- **List Structures**
  - **Syntax**
    - ⇔ `[apple, orange, grape]`
    - ⇔ `[]` : empty list
  - **Example 1**

```
[apple,orange,graph | [] ]
[apple,orange | [graph]]
[apple | [orange, graph]]
```

```
C1: p(a, b).
C2: p(a, c).
C3: p(b, d).
C4: p(c, e).
C5: q(X, Y) :- p(X, Z), p(Z, Y).
```

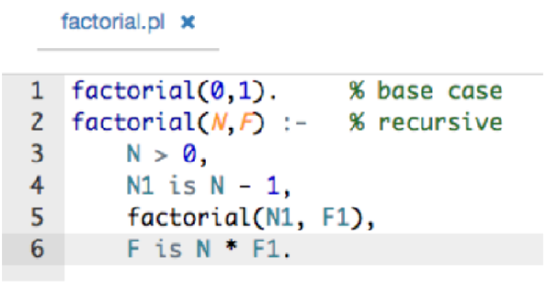




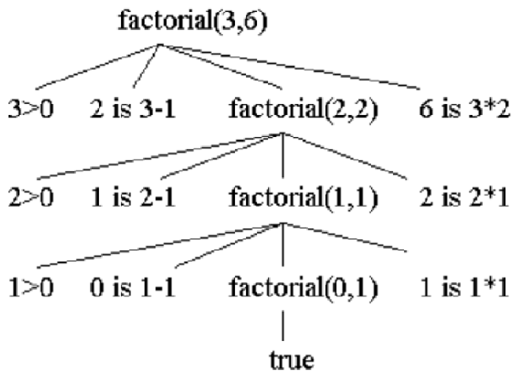
```
p(a, b).
p(a, c).
p(b, d).
p(c, e).
q(X, Y) :- p(X, Z), p(Z, Y).
```

```
q(a,W).
W = d
W = e
```

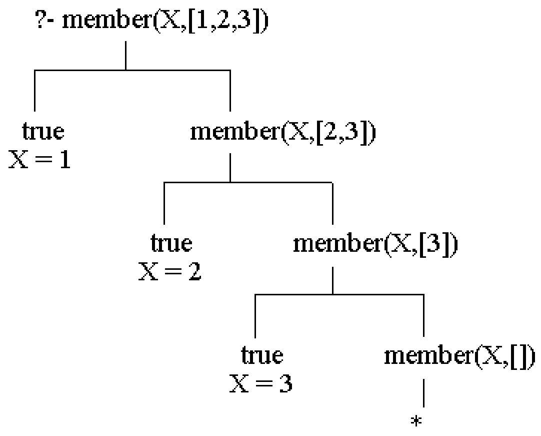
```
?- q(a,W).
```



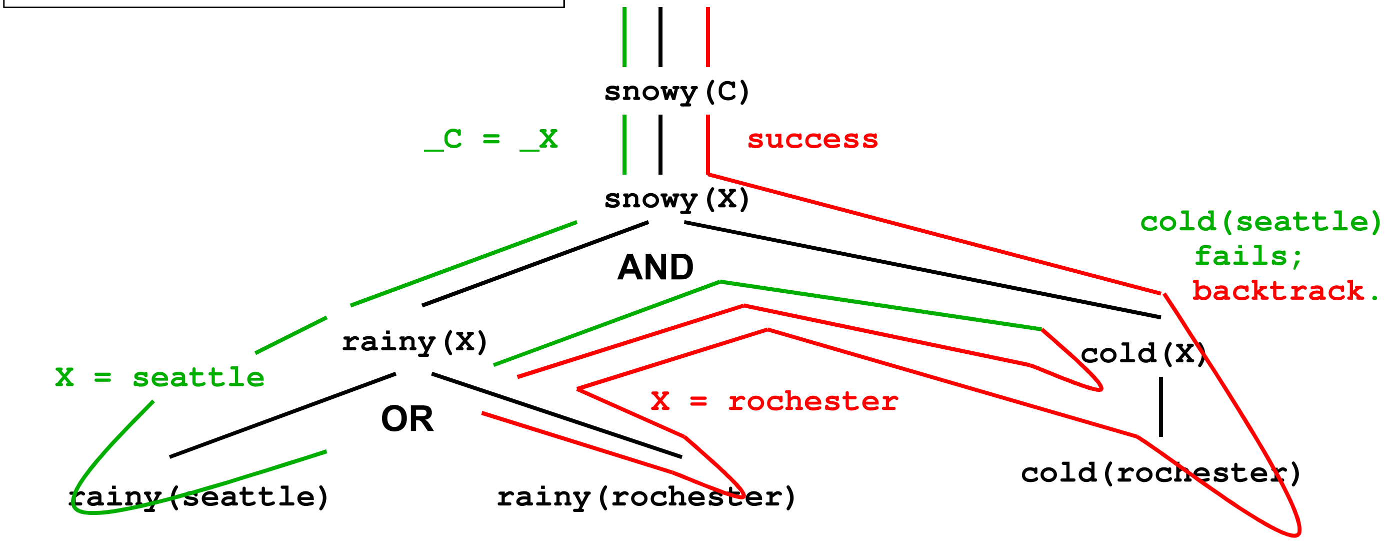
```
factorial(0,1).      % base case
factorial(N,F) :-    % recursive
    N > 0,
    N1 is N - 1,
    factorial(N1, F1),
    F is N * F1.
```



member(X, [X|_]).
member(X, [_|Y]) :- **member**(X,Y).

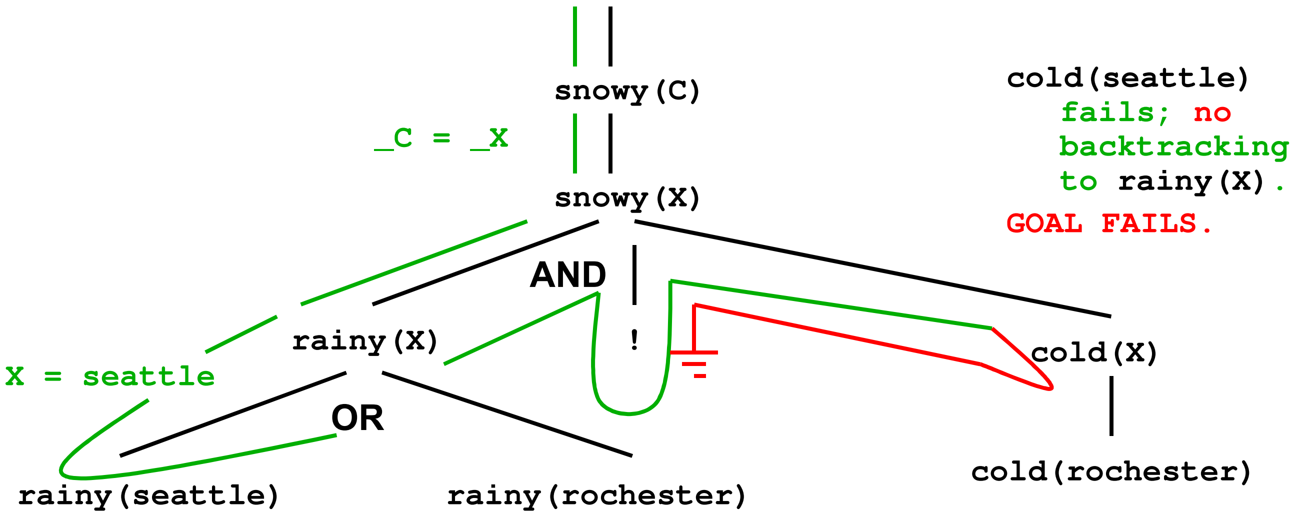


```
rainy(seattle).
rainy(rochester).
cold(rochester).

snowy(X) :- rainy(X), cold(X).
snowy(A).
:- snowy(C).
```



Cut

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), !, cold(X).

:- snowy(C).
```



cold(seattle) fails; no backtracking to rainy(X).

GOAL FAILS.

15.7 Deficiencies of Prolog

(1) Resolution Order Control

- Ordering of fact and rules
  - the user can profoundly affect efficiency by ordering the database statements to optimize a particular application

    ⇔ by placing particular rules first in database, if those rules are much more likely to succeed than others

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

- Explicit control of backtracking : cut operator (!)
  - not an operator, but a goal
  - it always succeeds immediately, but it cannot be resatisfied through backtracking
  - a side effect of the cut is that subgoals to its left in a compound goal also can not be satisfied through backtracking
  - although it is sometimes needed, it is possible to abuse it. Indeed, it is sometimes used to make logic program have a control flow that is inspired by imperative programming style

    ⇔ the programs does specify how solutions are to be found (bad logic programming style !)

```
aa:- a, b, !, c, d
member[Element, [Element | _] :- !
```

(2) The Closed World Assumption and Negation Problem

- the nature of Prolog's resolution sometimes creates misleading results

• Closed World Assumption

- Prolog has no knowledge of the world other than its database
  - ⇔ the only truths, as far as Prolog is concerned, are those that can be proved using its database
- Prolog can prove that a given goal is true, but *it can not prove that a given goal is false.*
  - ⇔ it simply assumes that, because it can not prove a goal true, the goal must be false
    - ⇒ "*Suspects are innocent until prove guilty. They need not to be innocent*"
- Prolog is a true/fail system, rather than true/false system

```
likes(jake, chocolate).
likes(jake, orange).
likes(smith, apple).
likes(smith, orange).

:- like(jake, apple).
no
```

• The Negation Problem

```
parent(bill,jake).
parent(bill, shelly).
sibling(X,Y) :- parent(M,X), parent(M,Y).

:- sibling(X,Y).
X = jake
Y = jake
```

```
parent(bill,jake).
parent(bill, shelly).
sibling(X,Y) :- parent(X,M), parent(M,Y), not(X=Y).
```

```
:- member(X, [mary, fred, nang]).
X = mary
:- not (not (member(X, [mary, fred, nang]))).
X = _1
```

the Prolog's `not` operator is not equivalent to a logical NOT operator

$A :- B_1 \cap B_2 \cap B_3 \cap \ldots B_n$

It all the B propositions are true, it can be conclude that A is true. But regardless of the truth or falseness of any or all of the Bs, it can not be conclude that A is false.

(3) Intrinsic Limitations

- **Fundamental Goals of Logic Programming**
  - provide a nonprocedural programming, that is, a system by which programmers specify *what a program is supposed to do* but *need not specify how that is to be accomplished*
  - Example

```
sort(old_list, new_list) ⊂ permute(old_list, new_list) ∩
                              sorted(new_list)
sorted(list) ⊂ ∀j such that 1 ≤ j < n, list(j) ≤ list(j+1)
```

```
sorted([]).
sorted([X]).
sorted([X,Y|List]):- X <= Y, sorted([Y|List]).
```

- the problem is that it has no idea of how to sort, rather than simply to enumerate all permutations of the given list until it happens to create the one that has the list in sorted order – a very slow process

15.8 Applications of Logic Programming

- **RDBMS**
  - stores data in the form of table
  - query language is nonprocedural
    - ⇔ the user does not describe how to retrieve the answer; rather, he or she only describes the characteristics of answer
  - Implementation of DBMS in Prolog
    - ⇔ Simple tables of information can be described by Prolog structures, and relationship between table can be conveniently and easily described by Prolog rules
    - ⇔ the retrieval process is inherent in the resolution operation
    - ⇔ the goal statement of Prolog provide the queries for the RDBMS
    - ⇔ Advantages
      - ⇒ only a single language is required
      - ⇒ can store facts and inference rules
    - ⇔ Problems
      - ⇒ its lower level of efficiency
        - → logical inferences are simply much slower than ordinary table look-up methods using imperative programming techniques

- Natural Language Processing
  - Natural language interfaces to computer software
    - ⇔ for describing language syntax, forms of logic programming have been found to be equivalent to context-free grammars
    - ⇔ Parsing » Proof procedure in Prolog

- Expert systems
  - computer systems designed to emulate human expertise in some particular domain.
  - consist of a database of facts, an inference process, some heuristics about the domain, and some friendly human interface
  - learn from the process of being used, so their database must be capable of growing dynamically
  - Implementation of Expert System in Prolog
    - ⇔ using resolution as the basis for query processing
    - ⇔ using it ability to add facts and rules to provide the learning capability, and using trace facility to inform the user of the 밚easoning?behind a given result

- Education

15.9 Conclusion

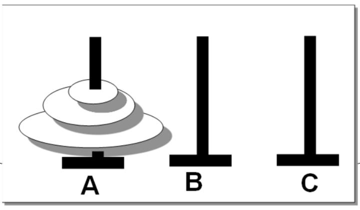
- Why Prolog ?
  - Prolog programs are likely to be more logically organized and written
  - Prolog processing is naturally parallel
  - a good tool for prototyping

부록 : 재미있는 Prolog Program

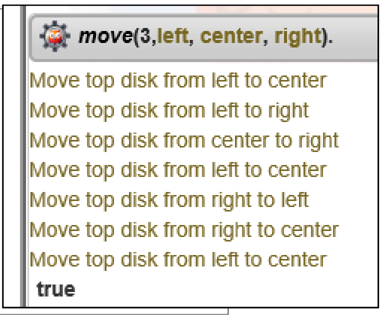
Quick Sorting

```
quicksort([ ], [ ]).
quicksort([X|Xs], Ys]) :-  partition(Xs, X, Littles, Bigs),
                           quicksort(Littles, Ls),
                           quicksort(Bigs, Bs),
                           append(Ls, [X|Bs], Ys).
partition([ ], Y, [ ], [ ]).
partition([X|Xs], Y, [X|Ls], Bs) :- X <= Y,
                                    partition(Xs, Y, Ls, Bs).
partition([X|Xs], Y, Ls, [X|Bs]) :- X > Y,
                                    partition(Xs, Y, Ls, Bs).
```

Tower of Hanoi



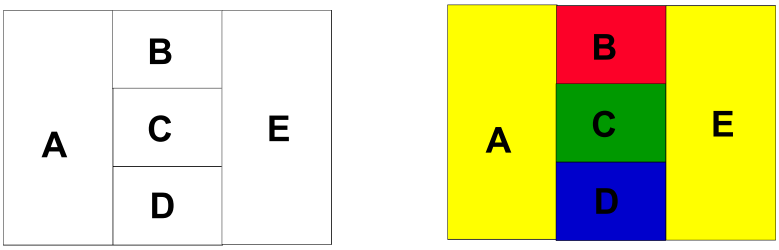
```
move(1,X,Y,_) :-
    write('Move top disk from '),
    write(X), write(' to '), write(Y),
    nl.

move(N,X,Y,Z) :-
    N>1, M is N-1,
    move(M,X,Z,Y), move(1,X,Y,_), move(M,Z,Y,X).
```



```
move(3,left, center, right).
Move top disk from left to center
Move top disk from left to right
Move top disk from center to right
Move top disk from left to center
Move top disk from right to left
Move top disk from right to center
Move top disk from left to center
true
```

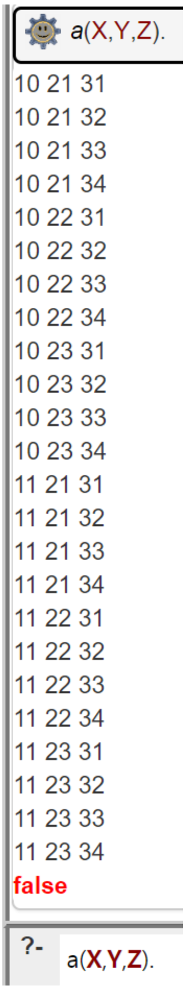
Map Coloring

```
mapcolor(A,B,C,D,E) :- color(A), color(B), color(C), color(D), color(E),
                       check_color(A,B,C,D,E).
check_color(A,B,C,D,E) :-
                        A \== B, A \==C, A \==D,
                        B \== C, C \== D,
                        B \== E, C \== E, D \== E.
color(yellow).
color(red).
color(green).
color(blue).
```



```
append([], L, L).
append([X|L1], L2, [X|L3]) :- append(L1, L2, L3).
/* :- append([1,2,3,4], [a,b,c,d], Y). */

member(X, [X|_]).
member(X, [_|Y]) :- member(X,Y).
/* :- member(1, [a,b,c,d]).
 * :- member(a, [1,2,3,4]).
 * :- member(a, [a,b,c,d]).
*/

alter([],[]).
alter([H|T], [X|Y]) :- change(H,X), alter(T,Y).
change(you,i).
change(are, [am,not]).
change(french, german).
change(do, no).
change(X,X).

/* :-  alter([you, are, a, computer]). */

nrev([], []).
nrev([H|T0], L) :-
            nrev(T0, T),
            append(T, [H], L).
```

```
a(X,Y,Z)  :- b1(X), b2(Y), b3(Z).

b1(10).
b1(11).

b2(21).
b2(22).
b2(23).

b3(31).
b3(32).
b3(33).
b3(34).
```



```
a(X,Y,Z).
10 21 31
10 21 32
10 21 33
10 21 34
10 22 31
10 22 32
10 22 33
10 22 34
10 23 31
10 23 32
10 23 33
10 23 34
11 21 31
11 21 32
11 21 33
11 21 34
11 22 31
11 22 32
11 22 33
11 22 34
11 23 31
11 23 32
11 23 33
11 23 34
false

?- a(X,Y,Z).
```

- a(A,B,C) query 에 대한 총 solution의 개수는 ?
- a(A,B,C) query 에 대한 모든 solution을 프린트 하도록 프로그램을 수정하

<참고> primitive goals :

- write(X) : 현재 X에 instantiate된 값을 프린트 한다.
- fail : 조건 없이 fail
- nl : 다음 line으로 cursor를 옮긴다. (next line)

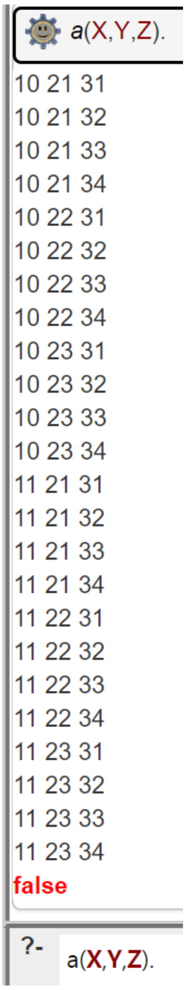
```
a(X,Y,Z) :- b1(X), b2(Y), b3(Z).

b1(10).
b1(11).

b2(21).
b2(22).
b2(23).

b3(31).
b3(32).
b3(33).
b3(34).
```

- a(A,B,C) query에 대한 총 solution의 개수는 ?
- a(A,B,C) query 에 대한 모든 solution을 프린트 하도록 프로그램을 수정하여라.

<참고> primitive goals :

- write(X) : 현재 X에 instantiate된 값을 프린트 한다.
- fail : 조건 없이 fail
- nl : 다음 line으로 cursor를 옮긴다. (next line)

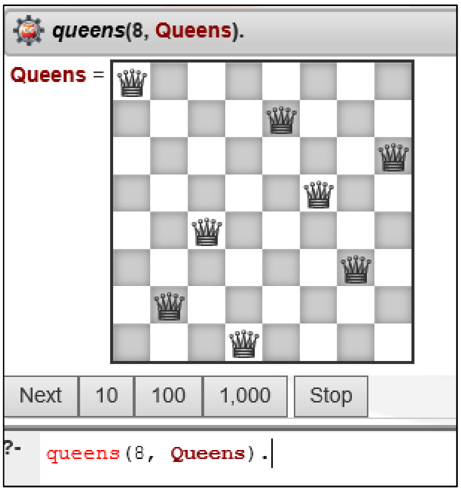
```
a(X,Y,Z) :- b1(X), b2(Y), b3(Z), print(X,Y,Z), fail.

b1(10).
b1(11).

b2(21).
b2(22).
b2(23).

b3(31).
b3(32).
b3(33).
b3(34).

print(X,Y,Z):- write(X), write(' '), write(Y),
               write(' '), write(Z), nl.
```



```
a(X,Y,Z).
10 21 31
10 21 32
10 21 33
10 21 34
10 22 31
10 22 32
10 22 33
10 22 34
10 23 31
10 23 32
10 23 33
10 23 34
11 21 31
11 21 32
11 21 33
11 21 34
11 22 31
11 22 32
11 22 33
11 22 34
11 23 31
11 23 32
11 23 33
11 23 34
false

?- a(X,Y,Z).
```

- a(A,B,C) query 에 대한 총 solution의 개수는 ? ➔ 2x3x4 = 24개
- a(A,B,C) query 에 대한 모든 solution을 프린트 하도록 프로그램을 수정하여라.

```
queens(N, Queens) :-
        length(Queens, N),
       board(Queens, Board, 0, N, _, _),
       queens(Board, 0, Queens).

board([], [], N, N, _, _).
board([_|Queens], [Col-Vars|Board], Col0, N, [_|VR], VC) :-
       Col is Col0+1,
       functor(Vars, f, N),
       constraints(N, Vars, VR, VC),
       board(Queens, Board, Col, N, VR, [_|VC]).

constraints(0, _, _, _) :- !.
constraints(N, Row, [R|Rs], [C|Cs]) :-
       arg(N, Row, R-C),
       M is N-1,
       constraints(M, Row, Rs, Cs).

queens([], _, []).
queens([C|Cs], Row0, [Col|Solution]) :-
       Row is Row0+1,
       select(Col-Vars, [C|Cs], Board),
       arg(Row, Vars, Row-Row),
       queens(Board, Row, Solution).


/** <examples>

?- queens(8, Queens).
```



queens(8, Queens).

Queens =

```
?- queens(8, Queens).
```

8. 입력된 리스트의 첫 번째 데이터와 마지막 데이터가 같은지 판단하는 프로그램을 가정하자. (25점)

(a) 아래 결과를 내는 Python 프로그램 함수 check(X)를 정의하여라. Recursion을 사용하지 말 것. (5점)

```
def check(L):
  /* 여기를 채워라, */

# main
A = [3,5,4,6,7]
B = [1,2,3,4,5]
C = [1,5,4,7,1,7,8,9,1]
print(check(A))      /* ➔ False */
print(check(B))      /* ➔ False */
print(check(C))      /* ➔ True */
```

(b) 아래 결과를 내는 Prolog 프로그램 check(X)를 정의하여라. (10점)

```
:- check([3,5,4,6,7]) ➔ Fail
:- check([1,2,3,4,5]) ➔ Fail
:- check([1,5,4,7,1,7,8,9,1]) ➔ True
```

(c) 위에서 작성한 Python 프로그램과 Prolog 프로그램의 길이가 L인 리스트를 check하는 경우의 각 프로그램의 평균 시간 복잡도 (Time Complexity)를 계산하라. (10점)

(a)

```
def check(L):
   if L[0] == L[len(L)-1]:
       return True
   else:
       return False
```

8. 입력된 리스트의 첫 번째 데이터와 마지막 데이터가 같은지 판단하는 프로그램을 가정하자. (25점)

(a) 아래 결과를 내는 Python 프로그램 함수 check(X)를 정의하여라. Recursion을 사용하지 말 것. (5점)

```
def check(L):
  /* 여기를 채워라, */

# main
A = [3,5,4,6,7]
B = [1,2,3,4,5]
C = [1,5,4,7,1,7,8,9,1]
print(check(A))      /* ➔ False */
print(check(B))      /* ➔ False */
print(check(C))      /* ➔ True */
```

(b) 아래 결과를 내는 Prolog 프로그램 check(X)를 정의하여라. (10점)

```
:- check([3,5,4,6,7]) ➔ Fail
:- check([1,2,3,4,5]) ➔ Fail
:- check([1,5,4,7,1,7,8,9,1]) ➔ True
```

(c) 위에서 작성한 Python 프로그램과 Prolog 프로그램의 길이가 L인 리스트를 check하는 경우의 각 프로그램의 평균 시간 복잡도 (Time Complexity)를 계산하라. (10점)

(b)

```
check([A,A]).
check([A,B|C]):- check([A|C]).
```

8. 입력된 리스트의 첫 번째 데이터와 마지막 데이터가 같은지 판단하는 프로그램을 가정하자. (25점)

(a) 아래 결과를 내는 Python 프로그램 함수 check(X)를 정의하여라. Recursion을 사용하지 말 것. (5점)

```
def check(L):
  /* 여기를 채워라, */

# main
A = [3,5,4,6,7]
B = [1,2,3,4,5]
C = [1,5,4,7,1,7,8,9,1]
print(check(A))      /* ➔ False */
print(check(B))      /* ➔ False */
print(check(C))      /* ➔ True */
```

(b) 아래 결과를 내는 Prolog 프로그램 check(X)를 정의하여라. (10점)

```
:- check([3,5,4,6,7]) ➔ Fail
:- check([1,2,3,4,5]) ➔ Fail
:- check([1,5,4,7,1,7,8,9,1]) ➔ True
```

(c) 위에서 작성한 Python 프로그램과 Prolog 프로그램의 길이가 L인 리스트를 check하는 경우의 각 프로그램의 평균 시간 복잡도 (Time Complexity)를 계산하라. (10점)

(a)

```
def check(L):
   if L[0] == L[len(L)-1]:
       return True
   else:
       return False
```

(b)

```
check([A,A]).
check([A,B|C]):- check([A|C]).
```

(c)

```
(정답)  Python : O(1)
        Prolog : O(n)
```

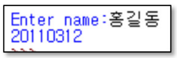
7. Python & Prolog (40점)
다음과 같은 table 형식의 자료가 있다고 가정한다. 이 자료 구조를 저장하는 python 및 Prolog 코드는 다음과 같다.

birthday_table

이름 (name)	생일 (birthday)
홍길동	20110312
김복순	20170543
김서강	20001011
박수철	20010324

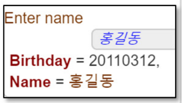
(a) 위의 자료 구조를 프로그램 내에 저장하기 위한 코드를 작성하라.
(가) Python의 dictionary를 이용하여 저장 (5점)
(나) Prolog Fact를 이용한 저장 (5점)

(b) (a)에서 정의한 table 정보에 대하여 사용자로 부터 이름을 입력 받아서 그 이름에 해당되는 생일을 출력하는 프로그램을
(가) Python으로 작성하라. (5점)
(나) Prolog를 질의 문을 작성하라.(10점



```
Enter name:홍길동
20110312
```

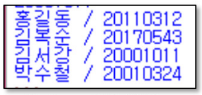
Python 수행 결과



```
Enter name
                홍길동
Birthday = 20110312,
Name = 홍길동
```

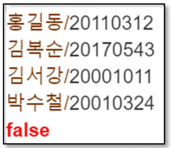
Prolog 수행 결과

(c) (a)에서 정의한 table에 저장되어 있는 모든 데이터를 이름/생일 형식으로 출력하는 프로그램을
(가) Python for loop를 이용하여 작성하라. (5점)
(나) Prolog 질의 문을 작성하라. (10점)



```
홍길동 / 20110312
김복순 / 20170543
김서강 / 20001011
박수철 / 20010324
```

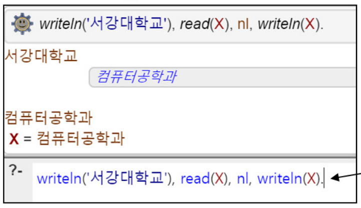
Python 수행 결과



```
홍길동/20110312
김복순/20170543
김서강/20001011
박수철/20010324
false
```

Prolog 수행 결과

<참고 1> Prolog 입출력 예제



```
writeln('서강대학교'), read(X), nl, writeln(X).
서강대학교
                컴퓨터공학과

컴퓨터공학과
 X = 컴퓨터공학과

?-  writeln('서강대학교'), read(X), nl, writeln(X).
```

질의

<참고 2> Prolog primitive subgoal

- `write('aa')` : "aa"를 출력하는 primitive goal
- `write(X)` : X에 instantiation된 값을 출력하는 primitive goal
- `read(X)` : 입력을 받아서 그 값을 X에 instantiation (대치) 시키는 primitive goal.
- `fail` : 항상 fail하는 primitive subgoal

7. Python & Prolog (40점)
다음과 같은 table 형식의 자료가 있다고 가정한다. 이 자료 구조를 저장하는 python 및 Prolog 코드는 다음과 같다.

birthday_table

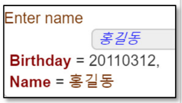
이름 (name)	생일 (birthday)
홍길동	20110312
김복순	20170543
김서강	20001011
박수철	20010324

(a) 위의 자료 구조를 프로그램 내에 저장하기 위한 코드를 작성하라.
(가) Python의 dictionary를 이용하여 저장 (5점)
(나) Prolog Fact를 이용한 저장 (5점)

(b) (a)에서 정의한 table 정보에 대하여 사용자로 부터 이름을 입력 받아서 그 이름에 해당되는 생일을 출력하는 프로그램을
(가) Python으로 작성하라. (5점)
(나) Prolog를 질의 문을 작성하라.(10점

```
Enter name:홍길동
20110312
```

Python 수행 결과



```
Enter name
                홍길동
Birthday = 20110312,
Name = 홍길동
```

Prolog 수행 결과

(c) (a)에서 정의한 table에 저장되어 있는 모든 데이터를 이름/생일 형식으로 출력하는 프로그램을
(가) Python for loop를 이용하여 작성하라. (5점)
(나) Prolog 질의 문을 작성하라. (10점)

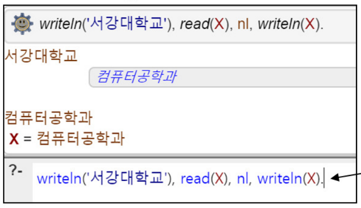
```
홍길동 / 20110312
김복순 / 20170543
김서강 / 20001011
박수철 / 20010324
```

Python 수행 결과

```
홍길동/20110312
김복순/20170543
김서강/20001011
박수철/20010324
false
```

Prolog 수행 결과

<참고 1> Prolog 입출력 예제



```
writeln('서강대학교'), read(X), nl, writeln(X).
서강대학교
                컴퓨터공학과

컴퓨터공학과
 X = 컴퓨터공학과

?-  writeln('서강대학교'), read(X), nl, writeln(X).
```

질의

<참고 2> Prolog primitive subgoal

- `write('aa')` : "aa"를 출력하는 primitive goal
- `write(X)` : X에 instantiation된 값을 출력하는 primitive goal
- `read(X)` : 입력을 받아서 그 값을 X에 instantiation (대치) 시키는 primitive goal.
- `fail` : 항상 fail하는 primitive subgoal

```
birthday_table(홍길동, 20110312).
birthday_table(김복순, 20170543).
birthday_table(김서강, 20001011).
birthday_table(박수철, 20010324).
```

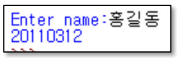
7. Python & Prolog (40점)
다음과 같은 table 형식의 자료가 있다고 가정한다. 이 자료 구조를 저장하는 python 및 Prolog 코드는 다음과 같다.

birthday_table

이름 (name)	생일 (birthday)
홍길동	20110312
김복순	20170543
김서강	20001011
박수철	20010324

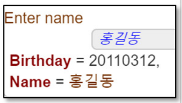
(a) 위의 자료 구조를 프로그램 내에 저장하기 위한 코드를 작성하라.
(가) Python의 dictionary를 이용하여 저장 (5점)
(나) Prolog Fact를 이용한 저장 (5점)

(b) (a)에서 정의한 table 정보에 대하여 사용자로 부터 이름을 입력 받아서 그 이름에 해당되는 생일을 출력하는 프로그램을
(가) Python으로 작성하라. (5점)
(나) Prolog를 질의 문을 작성하라.(10점



```
Enter name:홍길동
20110312
```

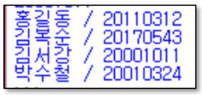
Python 수행 결과



```
Enter name
        홍길동
Birthday = 20110312,
Name = 홍길동
```

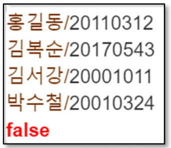
Prolog 수행 결과

(c) (a)에서 정의한 table에 저장되어 있는 모든 데이터를 이름/생일 형식으로 출력하는 프로그램을
(가) Python for loop를 이용하여 작성하라. (5점)
(나) Prolog 질의 문을 작성하라. (10점)



```
홍길동 / 20110312
김복순 / 20170543
김서강 / 20001011
박수철 / 20010324
```

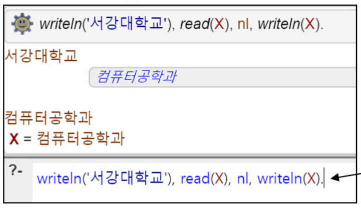
Python 수행 결과



```
홍길동/20110312
김복순/20170543
김서강/20001011
박수철/20010324
false
```

Prolog 수행 결과

<참고 1> Prolog 입출력 예제



```
writeln('서강대학교'), read(X), nl, writeln(X).
서강대학교
        컴퓨터공학과

컴퓨터공학과
X = 컴퓨터공학과

?- writeln('서강대학교'), read(X), nl, writeln(X).
```

질의

<참고 2> Prolog primitive subgoal

- `write('aa')` : "aa"를 출력하는 primitive goal
- `write(X)` : X에 instantiation된 값을 출력하는 primitive goal
- `read(X)` : 입력을 받아서 그 값을 X에 instantiation (대치) 시키는 primitive goal.
- `fail` : 항상 fail하는 primitive subgoal

(a)

(b)

```
:- writeln('Enter name'), read(Name),
   birthday_table(Name, Birthday).
```

(c)

```
:- birthday_table(Birthday, Name),
   writeln(Birthday), writeln( / ),
   writeln(Name), nl, fail.
```

7. Python & Prolog (40점)
다음과 같은 table 형식의 자료가 있다고 가정한다. 이 자료 구조를 저장하는 python 및 Prolog 코드는 다음과 같다.

birthday_table

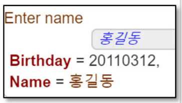
이름 (name)	생일 (birthday)
홍길동	20110312
김복순	20170543
김서강	20001011
박수철	20010324

(a) 위의 자료 구조를 프로그램 내에 저장하기 위한 코드를 작성하라.
(가) Python의 dictionary를 이용하여 저장 (5점)
(나) Prolog Fact를 이용한 저장 (5점)

(b) (a)에서 정의한 table 정보에 대하여 사용자로 부터 이름을 입력 받아서 그 이름에 해당되는 생일을 출력하는 프로그램을
(가) Python으로 작성하라. (5점)
(나) Prolog를 질의 문을 작성하라.(10점

```
Enter name:홍길동
20110312
```

Python 수행 결과



```
Enter name
        홍길동
Birthday = 20110312,
Name = 홍길동
```

Prolog 수행 결과

(c) (a)에서 정의한 table에 저장되어 있는 모든 데이터를 이름/생일 형식으로 출력하는 프로그램을
(가) Python for loop를 이용하여 작성하라. (5점)
(나) Prolog 질의 문을 작성하라. (10점)

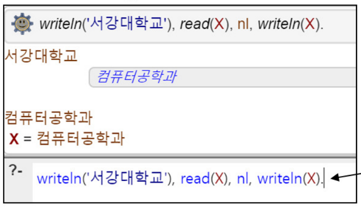
```
홍길동 / 20110312
김복순 / 20170543
김서강 / 20001011
박수철 / 20010324
```

Python 수행 결과

```
홍길동/20110312
김복순/20170543
김서강/20001011
박수철/20010324
false
```

Prolog 수행 결과

<참고 1> Prolog 입출력 예제



```
writeln('서강대학교'), read(X), nl, writeln(X).
서강대학교
        컴퓨터공학과

컴퓨터공학과
X = 컴퓨터공학과

?- writeln('서강대학교'), read(X), nl, writeln(X).
```

질의

<참고 2> Prolog primitive subgoal

- `write('aa')` : "aa"를 출력하는 primitive goal
- `write(X)` : X에 instantiation된 값을 출력하는 primitive goal
- `read(X)` : 입력을 받아서 그 값을 X에 instantiation (대치) 시키는 primitive goal.
- `fail` : 항상 fail하는 primitive subgoal

(a)

```
birthday_table(홍길동, 20110312).
birthday_table(김복순, 20170543).
birthday_table(김서강, 20001011).
birthday_table(박수철, 20010324).
```

7. Prolog Programming (30점)

(a) backtracking의 동작 방법을 아래 프로그램 수행 방법을 예로 하여설명하라. (5점).

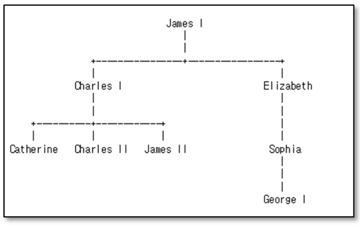
```
a :- b(X,Y), c(X,Y).
a :- d(A,B), e(A,B).

:- a. /* Query */
```

(b) 다음의 Prolog 프로그램에 대하여 아래 각각의 질의의 결과는 무엇인가 ?
(각각 질의에 대하여 생성되는 모든 결과를 써라 (all solution)) (10점)

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

alter([],[]).
alter([H|T], [X|Y]) :- change(H,X), alter(T,Y).
change(you,i).
change(are, [am,not]).
change(french, german).
change(do, no).
change(X,X).

(가) :- append(X,Y, [1,2,3]).   /* 5점 */
(나) :- alter([you, are, a, computer], X).  /* 5점 */
```

(c) 다음의 예제를 이용하여 각 관계를 나타내는 프로그램 (한 개의 Rule)을 작성하라. (15점)

```
male(james1).
male(charles1).
male(charles2).
male(james2).
male(george1).
female(catherine).
female(elizabeth).
female(sophia).

parent(charles1, james1).
parent(elizabeth, james1).
parent(charles2, charles1).
parent(catherine, charles1).
parent(james2, charles1).
parent(sophia, elizabeth).
parent(george1, sophia).
```



옆의 프로그램에 의하여 정의된 가계도

(가) 자매(X,Y)   /* X와 Y는 자매이다. (5점) */
(나) 이모(X,Y)   /* Y는 X의 이모이다. (5점), (가)에서 정의한 "자매(X,Y)"를 이용할 것 */
(다) 조부모(X,Y) /* Y는 X의 조부모이다. (5점) */

7. Prolog Programming (30점)

(a) backtracking의 동작 방법을 아래 프로그램 수행 방법을 예로 하여설명하라. (5점).

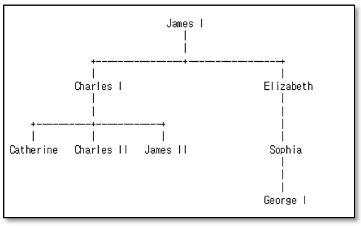
```
a :- b(X,Y), c(X,Y).
a :- d(A,B), e(A,B).

:- a. /* Query */
```

(b) 다음의 Prolog 프로그램에 대하여 아래 각각의 질의의 결과는 무엇인가 ?
(각각 질의에 대하여 생성되는 모든 결과를 써라 (all solution)) (10점)

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

alter([],[]).
alter([H|T], [X|Y]) :- change(H,X), alter(T,Y).
change(you,i).
change(are, [am,not]).
change(french, german).
change(do, no).
change(X,X).

(가) :- append(X,Y, [1,2,3]).   /* 5점 */
(나) :- alter([you, are, a, computer], X).  /* 5점 */
```

(c) 다음의 예제를 이용하여 각 관계를 나타내는 프로그램 (한 개의 Rule)을 작성하라. (15점)

```
male(james1).
male(charles1).
male(charles2).
male(james2).
male(george1).
female(catherine).
female(elizabeth).
female(sophia).

parent(charles1, james1).
parent(elizabeth, james1).
parent(charles2, charles1).
parent(catherine, charles1).
parent(james2, charles1).
parent(sophia, elizabeth).
parent(george1, sophia).
```



옆의 프로그램에 의하여 정의된 가계도

(가) 자매(X,Y)   /* X와 Y는 자매이다. (5점) */
(나) 이모(X,Y)   /* Y는 X의 이모이다. (5점), (가)에서 정의한 "자매(X,Y)"를 이용할 것 */
(다) 조부모(X,Y) /* Y는 X의 조부모이다. (5점) */

(가) :- append(X,Y, [1,2,3]).   /* 5점 */
(정답)
- X=[], Y=[1,2,3]
- X=[1], Y=[2,3]
- X=[1,2], Y=[3]
- X=[1,2,3], Y=[2]

(나) :- alter([you, are, a, computer], X). /* 5점 */
(정답)
- X = [i, [am, not], a, computer]
- X = [i, are, a, computer]
- X = [you, [am, not], a, computer]
- X = [you, are, a, computer]

(가) 자매(X,Y)   /* X와 Y는 자매이다. (5점) */
(정답)

```
sister(X,Y) :- female(X), female(Y),
               parent(X, Z), parent(Y, Z), not(X=Y).
```

(나) 이모(X,Y)   /* Y는 X의 이모이다. (5점) */
(정답)

```
aunt(X,Y) :- parent(X,Z), sister(Y,Z).
    /* sister()에 female을 check하는 것이 있기 때문에
       sister(Y)를 check 하지 않아도 됨. */
```

(다) 조부모(X,Y) /* Y는 X의 조부모이다. (5점) */
(정답)

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