Chapter 15

Logic Programming Languages

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

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]);
    }
}
```

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

15.2 A Brief Introduction to Predicate Calculus

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

(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

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

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

```
append([],[],[]).
append(a,[],[a]).
append([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	∩	`a ∩ b`	`a` **and** `b`
disjunction	∪	`a ∪ b`	`a` **or** `b`
equivalence	≡	`a ≡ b`	`a` **is equivalent to** `b`
implication	⊃	`a ⊃ b`	`a` **implies** `b`
	⊂	`a ⊂ b`	`b` **implies** `a`

  - **variables can appear in propositions but only when introduced by special symbols called quantifiers**
    - ⇔ **example**
      - ⇒ `∀X. (woman(X) ⊃ human(X))`
        - → for any value of X, if X is a woman, then X is a human
      - ⇒ `∃X.(mother(mary,X) ∩ 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* `A`*s 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

$$\begin{array}{l} T \subset P_2 \\ Q_1 \subset T \end{array} \xrightarrow{\textit{inference}} Q_1 \subset P_2$$

```
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 C 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).
```


- 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 !)
```

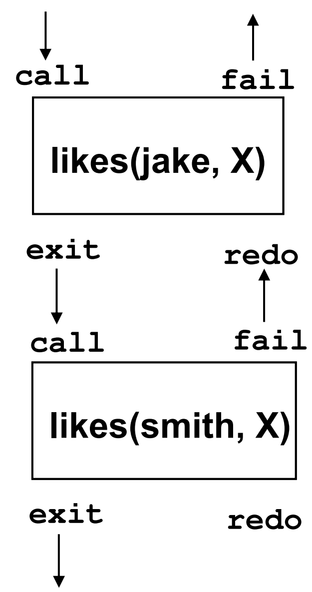
```
speed(ford, 100).
speed(pony, 105).
speed(pride, 95).
time(ford, 20).
time(pony, 21).
time(pride, 24).
distance(X,Y) :- speed(X, Speed),
                 time(X, Time),
                 Y is Speed * Time.
```

```
:- 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
```

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

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

- **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
```


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

- th 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

$A\text{:- } B_1 \cap B_2 \cap B_3 \cap \ldots B_n$

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 `B`s, 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 뱮easoning?behind a given result**

- **Education**

15.9 Conclusion

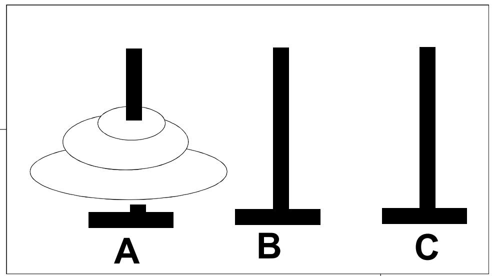
- **Why Prolog ?**
  - **Prolog programs are likely to be more logically organized and written**
  - **Prolog processing is naturally parallel**
  - **a good tool for prototyping**

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(0, _, _, _, _).
move(N, A, B, C) :- M is N - 1,
                    move(M, A, C, B),
                    inform(A,B),
                    move(M, C, B, A).
inform(X, Y) :-
    write([move, a, disk, from, the, X, pole, to, the, Y pole]),
    nl.
```


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).
```