CS370

# Symbolic Programming
# Declarative Programming

LECTURE 10: Programming Style

**Jong C. Park**
park@cs.kaist.ac.kr
**Computer Science Department**
**Korea Advanced Institute of Science and Technology**
**http://nlp.kaist.ac.kr/~cs370**

# Programming Style

- General principles of good programming
- How to think about Prolog programs
- Programming style
- Debugging
- Improving efficiency

# General principles

## Criteria

- Correctness
- User-friendliness
- Efficiency
- Readability
- Modifiability
- Robustness
- Documentation

# General principles

## Top-down Stepwise Refinement

- rough solutions: most relevant
- succinct and simple: likely to be correct
- small refinement steps: intellectually manageable

# How to think about Prolog programs

## ⦿ Use of recursion

- boundary cases and general cases
- Example

```
maplist(List,F,NewList)
maplist([ ],_,[ ]).
maplist([X|Tail],F,[NewX|NewTail]) :-
  G =.. [F,X,NewX],
  call(G),
  maplist(Tail,F,NewTail).
?- maplist([2,6,5],square,Squares).
```

# How to think about Prolog programs

## ⊙ Generalization

- enables recursive formulation;
- makes the original a special case.
- Example

eightqueens(Pos)

nqueens(Pos,N)

- boundary case: N = 0
- general case: N > 0

```
eightqueens(Pos) :- nqueens(Pos,8).
```

# How to think about Prolog programs

## Using pictures

- With pictorial representation, essential relations are easily perceived.
- Mapping to Prolog
  - nodes and arcs in graphs can be modeled by objects and relations.
  - trees are mapped by structured objects
  - declarative meaning makes the order of translation irrelevant

# Programming style

- **Why follow stylistic conventions?**
- **Some rules**

  short clauses　　　short procedures

  mnemonic names　　　layout of the programs

  consistent stylistic conventions

  cut and not should be used with much care

  assert/retract with much care

  semicolon

# Programming style

- **Tabular organization**
  - for long procedures
    - Clear structure
    - Incrementality
    - Modification
- **Commenting**

# Debugging

## ⦿ Principle of debugging

- Test smaller units first.

## ⦿ Debugging aids

trace

notrace

spy(P)

nospy(P)

# Improving Efficiency

## Factors

- execution time
- space requirements
- the time for program development
- frequency of use

# Improving Efficiency

## ⦿ Areas of successful Prolog applications

- symbolic solutions for equations, planning, and databases
- general problem solving, prototyping
- implementation of programming languages
- discrete and qualitative simulation
- architectural design, machine learning
- natural language understanding, expert systems

# Improving Efficiency

- Improving the efficiency of an eight queens program

```
member(Y,[1,2,3,4,5,6,7,8]).

member(Y,[1,5,2,6,3,7,4,8]).
```

# Improving Efficiency

## ⦿ Map colouring program

- The goal is to assign each country in a given map one of four given colours such that no two neighbouring countries are painted with the same colour.
- A map is specified by the neighbor relation

  ngb(Country, Neighbours)

  ngb(andorra,[france,spain]).

  ngb(austria, [czech_republic,germany,hungary,italy, liechtenstein,slovakia,slovenia,switzerland])
- Let a solution be represented as a list of pairs of the form: Country/Colour
  - [albania/C1,andorra/C2,austria/C3,...]

# Improving Efficiency

## ⦿ Map colouring program

- Define the predicate colours(Country_colour_list)
- Assume the colors yellow, blue, red, and green.

```
colours([ ]).
colours([Country/Colour|Rest]) :- colours(Rest),
        member(Colour,[yellow,blue,red,green]),
        not(member(Country1/Colour,Rest),
            neighbor(Country,Country1)).
neighbor(Country,Country1) :-
        ngb(Country,Neighbours),
        member(Country1,Neighbours).
```

# Improving Efficiency

## Map colouring program

- Inefficient
  - if there is a large number of countries, such as Europe.

```
country(C) :- ngb(C,_).
```

```
?- setof(Cntry/Colour,country(Cntry),CountryColourList),
   colours(CountryColourList).
```

# Improving Efficiency

## ⦿ Map colouring program

```
makelist(List) :- collect([germany],[ ],List).
collect([ ],Closed,Closed).           % no more candidates
collect([X|Open],Closed,List) :-
    member(X,Closed), !,              % X is already collected
    collect(Open,Closed,List).
collect([X|Open],Closed,List) :-
    ngb(X,Ngbs),
    conc(Ngbs,Open,Open1),
    collect(Open1,[X|Closed],List).
```

# Improving Efficiency

## List concatenation

- Simple concatenation

```
conc([ ],L,L).
conc([X|L1],L2,[X|L3]) :- conc(L1,L2,L3).
?- conc([a,b,c],[d,e],L).
conc([a,b,c],[d,e],L).
  conc([b,c],[d,e],L')
    conc([c],[d,e],L'')
      conc([ ],[d,e],L''')
        true
```

# Improving Efficiency

## ⦿ List concatenation

- Use of difference lists

```
[a,b,c] as [a,b,c,d,e]-[d,e],
           [a,b,c]-[ ],
           [a,b,c,d,e|T]-[d,e|T], ...
concat(A1-Z1,Z1-Z2,A1-Z2).
?- concat([a,b,c|T1]-T1,[d,e|T2]-T2,L).
T1 = [d,e|T2]
L = [a,b,c,d,e|T2]-T2
```

# Improving Efficiency

## ⦿Last call optimization and accumulators

- Use tail recursion: sumlist(List,Sum)

```
% without tail recursion
sumlist([ ], 0).
sumlist([X|Rest],Sum) :- sumlist(Rest,Sum0),
                         Sum is X + Sum0.
% with tail recursion
sumlist(List,Sum) :- sumlist(List,0,Sum).
sumlist([ ],Sum,Sum).
sumlist([First|Rest],PartialSum,TotalSum) :-
        NewPartialSum is PartialSum + First,
        sumlist(Rest,NewPartialSum,TotalSum).
```

# Improving Efficiency

## ⦿ Last call optimization and accumulators

- Use tail recursion with an accumulator

```
% reverse(List,ReversedList).
reverse([ ],[ ]).
reverse([X|Rest],Reversed) :- reverse(Rest,RevRest),
                              conc(RevRest,[X],Reversed).
% with an accumulator
reverse(List,Reversed) :- reverse(List,[ ],Reversed).
reverse([ ],Reversed,Reversed).
reverse([X|Rest],PartReversed,TotalReversed) :-
  reverse(Rest,[X|PartReversed],TotalReversed).
```

# Improving Efficiency

## ⦿ Simulating arrays with arg

- direct indexing with arg and functor

  functor(A,f,100) gives rise to A = f(_,_,_,...,_).

  A[60] := 1 can be done by arg(60,A,1).

  X := A[60] corresponds to arg(60,A,X).

- updating the values
  - several possibilities

# Improving Efficiency

## ⦿Asserting derived facts: fib(N,F)

1, 1, 2, 3, 5, 8, 13, ...

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

?- fib(6,F).
```

# Improving Efficiency

- **Asserting derived facts: fib2(N,F)**

```
fib2(1,1).
fib2(2,1).
fib2(N,F) :- N>2, N1 is N-1,
             fib2(N1,F1), N2 is N-2,
             fib2(N2,F2), F is F1+F2,
             asserta(fib2(N,F)).
```

# Improving Efficiency

- **Asserting derived facts: fib3(N,F)**

```
fib3(N,F) :- forwardfib(2,N,1,1,F).
forwardfib(M,N,F1,F2,F2) :- M >= N.
forwardfib(M,N,F1,F2,F) :-
    M<N, NextM is M+1,
    NextF2 is F1+F2, forwardfib(NextM,N,F2,NextF2,F).
```

# Summary

- General principles of good programming
- How to think about Prolog programs
- Programming style
- Debugging
- Improving efficiency