Introduction to Prolog

- What is Prolog?
  - Prolog (Programming in Logic) is a programming language for AI and non-numerical programming in general.
- What is new?
  - Conventional (well established, standard) languages are procedurally oriented, while Prolog introduces the declarative view.

1. Syntax of Prolog

(a) clauses

- A Prolog program consists of clauses.
- A Prolog clause is a Horn clause.
- Each clause terminates with a full stop.
- Prolog clauses are of three types: rules, facts, and questions.
- Example1: rule – offspring(X, Y) :- parent(Y, X).
- Example2: fact – parent(tom, ann).
- Example3: questions – parent(X, ann).

(b) procedures

- A group of clauses which the head of each clause are the same is called a procedure.
- Example:

```
connects(san_francisco, oakland, bart_tran).
connects(san_francisco, fremont, bart_tran).
connects(concord, daly_city, bart_tran).
```

(c) rules

- A rule is a clause with one or more conditions. For a rule to be true, all of its conditions must also be true.
- The right-hand side of the rule is a condition and the left-hand side of the rule is the conclusion.
- The ':-' sign means 'if'. The clause in Example1 can be stated as X is an offspring of Y if Y is a parent of X.
- The left-hand side is called the head of the rule and the right-hand side is called the body of the rule. The body of a rule is a list of goals separated by commas.
- Commas are understood as conjunctions.
- Example: p :- q, r, s, t. means p is true if q and r and s and t are true.

(d) facts

- Facts are always unconditionally true. It only has a head part (the conclusion) with an empty body.

- Example: has(student, books).

(e) questions

- Questions are queries for retrieving facts through rules. It only has the body (the goals).
- Example: fly(penguin).

(f) arguments

- Arguments can be an atom, a number, a variable, or a structure.

(g) atoms

- Atoms are concrete objects (but numbers are NOT atoms).
- The first letter of an atom is always a small letter.
- Strings of letters: tom, nil, x25, x_25AB, x_, x__y, top_part
- Strings of special characters: <—>, ===>, ..., .:. (with caution)
- Quoted strings: 'Tom', 'South America'

(h) numbers

- The syntax of integers is simple except one must remember that the range of integers is limited to an interval, in SICSTUS Prolog, between -16383 to 16383.
- Examples: 3.1416, -0.001, 22.223, 100

(i) variables

- Variables are general objects.
- The first letter of a variable is always a Capital letter.
- Example: X and Y in Example1.
- Variables can be substituted by another object (becomes instantiated).
- variables are assumed to be universally quantified (for all).
- Example: For all X and Y, if X is a offspring of Y then Y is a parent of X (Example1).
- When a variable appears in a clause only once, it is called *anonymous* variable which is represented by a _ alone or start with it.
- Examples: _, _23, _X, _var
- A variable is *instantiated* if it is bound to a non-variable term. That is, to an atom, a number, or a structure.

(j) structures (functions)

- Structures are functors that have zero or more arguments.
- Example: start_date(6, may, 1996).
- Example: triangle(point(4,2), point(6,4), point(7,1)).
- The **start_date** and **point** are called functor, and **triangle** is called the principle functor.
- A functor is defined by two things: name and arity (number of arguments). The general form of a functor is functor/arity.
- Examples: start_date/3, triangle/3, point/2

(k) predicates – a functor that specifies some relationship existing in the problem domain. The difference between a predicate and a function is that the function will return a value to the calling procedure, while a predicate does not returns value but a true or false statement.

- Examples: end_of_file/0 and integer/1 are built-in predicates
- Built-in predicates come with the system which do not need to be compiled or consulted before it is used.

(l) recursive

- A running procedure calls itself with different arguments and for the purpose of solving some subset of the original problem.
- Recursive is one of the fundamental principles of programming in Prolog.
- Example:

```
predecessor(X, Z) :-
     parent(X, Z).

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

(m) comments

- Example: /* this is a comment */
- Example: % This is also a comment

2. **matching**

- Given two terms, we say that they *match* if:
  - they are identical or
  - they become identical after they are instantiated
- If two terms are match then the process succeeds. Otherwise, the process fails.
- Examples (arithmetic expressions): $6 =:= 6, 6 = \backslash = 7$
- Examples (others): six == six, six \== seven

3. **Order of Clauses and Goals**

- danger of indefinite looping
- Example 1: p :- p.
- Example 2: The monkey and the banana program in Assignment # 1
- order of clauses is important
- Example 3:

```
likes(Person, Something) :-
     animal(Something).
```

```
likes(mary, snake) :-
     !, fail.
```

The most general clause should be tried last while the most specific clause should be tried first.

- Example 4:

```
example(1, 2, 3, 4, 5) :- do_p1.
example(1, 2, 3, 4, _) :- do_p2.
example(1, 2, 3, _, _) :- do_p3.
example(1, 2, _, _, _) :- do_p4.
example(1, _, _, _, _) :- do_p5.
example(_, _, _, _, _) :- do_p6.
```

Declarative and Procedural Meaning of Prolog Programs

1. the declarative meaning

   - concerned only with the relations defined by the program and determines *what* will be the output of the program
   - determines whether a given goal is true and if so, for what values of variables it is true.
   - Given a goal G, the declarative meaning says:
     G is true if and only if
     - there is a clause C in the program and there is an instance I of C such that
       (1) the head of I is identical to G and
       (2) all the goals in the body of I are true
   - Example:

```
given G = hasachild(peter).

The program has a clause
      C = hasachild(X) :- parent(X, Y).
and a fact
      parent(peter, tom).
Then
      hasachild(peter) :- parent(peter, Z).
and
      parent(peter,tom).
is an instance of G
```

2. the procedural meaning

   - specifies *how* Prolog answers questions.
   - means to try to satisfy a list of goals

- output a success or failure indicator and an instantiation of variables (p.48-50, Bratko)
- Example:

```
Program:

dark(Z) :- black(Z); brown(Z).

big(bear).
big(elephant).
small(cat).

brown(bear).
black(cat).
gray(elephant).

Trace the question dark(X), big(X):
| ?- trace, dark(X), big(X). -- the initial goal.

   Call: dark(_51) ?   -- scan the program from top to bottom looking for
                          a clause whose head matches the first goal
   Call: black(_51) ?  -- found clause dark(Y) :- black(Y), replace the
                          first goal by the instantiated body of this
                          clause and the new goals are black(X), big(X).
   Exit: black(cat) ?  -- scan the program to find a match for black(X)
                          and black(cat) is found.
   Exit: dark(cat) ?   -- the initial goal dark(X) has been instantiated to
   Call: big(cat) ?    -- dark(cat) and the new goal is sinked to big(cat).
   Fail: big(cat) ?    -- scan the program again and big(cat) is not found
   Redo: dark(cat) ?   -- backtracking to dark(cat)
   Redo: black(cat) ?  -- backtracking to black(cat)
   Fail: black(_51) ?  -- the goal black(X) is failed
   Call: brown(_51) ?  -- try next alternative dark(Y) :- brown(Y).
   Exit: brown(bear) ? -- scan the program and brown(bear) is found
   Exit: dark(bear) ?  -- the new goal is reduced into big(bear)
   Call: big(bear) ?   -- scan the program to find big(bear)
   Exit: big(bear) ?   -- big(bear) is found

X = bear ?             -- X is instantiated to bear

yes                    -- goal list is empty and terminated with success
```

How to run Prolog?

1. All Prolog clauses, questions end with a full stop.
2. Single quotation marks are used if files has extensions.

3. ';' means *or*.

4. Commas represent *and*.

   - Example: p :- q, r.
   - Example: p :- q ; r.

5. type 'sicstus' to invoke Prolog

```
aristotle[463]% sicstus
SICStus 2.1 #9: Wed Oct 11 15:33:09 CST 1995
| ?-
['back_trac.pl'].
{consulting /u/jian/public_html/420/sample/back_trac.pl...}
{/u/jian/public_html/420/sample/back_trac.pl consulted, 33 msec 1072 bytes}
yes
| ?- f(1, Y), 2 < Y.

Y = 4 ?
yes

| ?- halt.
```

6. consult(program_name). – load a program into Prolog through the interpreter. Interpreted code runs 8 times slower than compiled code, but you can debug interpreted code in more detail than compiled code (also known as interpret(program_name)).

   - Example: consult('ass/ass1/monkey.pl').

7. compile(program_name) – load a program into Prolog through the compiler. Compiled code runs 8 times faster than interpreted code, but you cannot debug compiled code in as much detail as interpreted code.

   - Example: compile(falmily).

8. The difference between counsult/1, load/1, and compile/1.

```
| ?- compile(conc).
{compiling /u/jian/public_html/420/sample/conc...}
{/u/jian/public_html/420/sample/conc compiled, 33 msec 464 bytes}

yes
| ?- trace, conc([a,b], [c,d], List).
{The debugger will first creep -- showing everything (trace)}
   1  Call: conc([a,b],[c,d],_83) ?
   1  Exit: conc([a,b],[c,d],[a,b,c,d]) ?
```

```
List = [a,b,c,d] ?

yes
{trace}

| ?- consult(user:conc).
{consulting /u/jian/public_html/420/sample/conc...}
{/u/jian/public_html/420/sample/conc consulted, 0 msec 48 bytes}

yes
| ?- trace, conc([a,b], [c,d], List).
{The debugger will first creep -- showing everything (trace)}
   1  Call: conc([a,b],[c,d],_83) ?
   2  Call: conc([b],[c,d],_714) ?
   3  Call: conc([],[c,d],_934) ?
   3  Exit: conc([],[c,d],[c,d]) ?
   2  Exit: conc([b],[c,d],[b,c,d]) ?
   1  Exit: conc([a,b],[c,d],[a,b,c,d]) ?

List = [a,b,c,d] ?

yes
{trace}
```

Lists, Operators, Arithmetic, Using Structures

Syntax of Lists

1. Lists are simple data structure widely used in non-numeric programming.
2. A list is a sequence of any number of items.
   - Example: [apple, pear, grape, orange]
3. Lists have two cases: empty or non-empty
   - Examples: [ ] and [1, 2, 3, 4, 5]
4. An empty list is an atom in Prolog
5. The first element of a non-empty list is called the *head*, and the rest of the list is called the *tail*.
   - Example: [Head|Tail] = [1|2,3,4,5]; [H1, H2|Tail] = [1,2|3,4,5]
6. A list can be an element of a list.
   - Example1: [ann, [tennis, music], tom, [skiing, food]]

- Example2: [[[[1], 2], 3], 4, [5]]

Operations on Lists

1. membership

   - member(X, List).
   - We want to find if X is a member of the List.
   - Case 1: X is equal to the first element of List
   - Case 2: X is not equal to the first element of List
   - Program:

```
member(X, [X|_]).
member(X, [_|Tail]) :-
     member(X, Tail).

trace on member(a, [x,1,3,q,z,a,b]).

1  Call: member(a,[x,1,3,q,z,a,b]) ?
2  Call: member(a,[1,3,q,z,a,b]) ?
3  Call: member(a,[3,q,z,a,b]) ?
4  Call: member(a,[q,z,a,b]) ?
5  Call: member(a,[z,a,b]) ?
6  Call: member(a,[a,b]) ?
6  Exit: member(a,[a,b]) ?
5  Exit: member(a,[z,a,b]) ?
4  Exit: member(a,[q,z,a,b]) ?
3  Exit: member(a,[3,q,z,a,b]) ?
2  Exit: member(a,[1,3,q,z,a,b]) ?
1  Exit: member(a,[x,1,3,q,z,a,b]) ?

trace on member(a, [x,1,3,q,z]).

1  Call: member(a,[x,1,3,q,z]) ?
2  Call: member(a,[1,3,q,z]) ?
3  Call: member(a,[3,q,z]) ?
4  Call: member(a,[q,z]) ?
5  Call: member(a,[z]) ?
6  Call: member(a,[]) ?
6  Fail: member(a,[]) ?
5  Fail: member(a,[z]) ?
4  Fail: member(a,[q,z]) ?
3  Fail: member(a,[3,q,z]) ?
2  Fail: member(a,[1,3,q,z]) ?
1  Fail: member(a,[x,1,3,q,z]) ?
```

2. concatenation

    - conc(List1, List2, NewList).
    - We want to combine List1 and List2 into a NewList.
    - Actually the job is to insert List2 after List1
    - Program:

```
conc([], List2, List2).
conc([X|Tail1], List2, [X|Tail]) :-
     conc(Tail1, List2, Tail).

trace of conc([a,b], [c,d], New):

1  Call: conc([a,b],[c,d],_83) ?
2  Call: conc([b],[c,d],_712) ?
3  Call: conc([],[c,d],_932) ?
3  Exit: conc([],[c,d],[c,d]) ?
2  Exit: conc([b],[c,d],[b,c,d]) ?
1  Exit: conc([a,b],[c,d],[a,b,c,d]) ?

New = [a,b,c,d]

trace of conc([], [a,b,c], New).

1  Call: conc([],[a,b,c],_93) ?
1  Exit: conc([],[a,b,c],[a,b,c]) ?

New = [a,b,c]

trace of conc([a,b,c], [], New).

1  Call: conc([a,b,c],[],_93) ?
2  Call: conc([b,c],[],_384) ?
3  Call: conc([c],[],_603) ?
4  Call: conc([],[],_821) ?
4  Exit: conc([],[],[]) ?
3  Exit: conc([c],[],[c]) ?
2  Exit: conc([b,c],[],[b,c]) ?
1  Exit: conc([a,b,c],[],[a,b,c]) ?

New = [a,b,c]
```

3. adding an item

    - add(Item, Position, List, NewList).
    - We want to add an Item at Position of the List.

- Two cases: Reach to the Position, not yet at the Position.
- Program:

```
add(Item, 0, List, [Item|List]).
add(Item, Position, [L|Tail1], [L|Tail2]) :-
     P1 is Position - 1,
     add(Item, P1, Tail1, Tail2).

trace of add(k, 3, [c,a,e], NewList).

1  Call: add(k,3,[c,a,e],_113) ?
2  Call: _755 is 3-1 ?
2  Exit: 2 is 3-1 ?
3  Call: add(k,2,[a,e],_761) ?
4  Call: _1419 is 2-1 ?
4  Exit: 1 is 2-1 ?
5  Call: add(k,1,[e],_1425) ?
5  Exit: add(k,1,[e],[k,e]) ?
3  Exit: add(k,2,[a,e],[a,k,e]) ?
1  Exit: add(k,3,[c,a,e],[c,a,k,e]) ?

NewList = [c,a,k,e]
```

4. deleting an item

- delete(Item, List, NewList).
- We want to delete the Item from the List.
- Case1: Item matches the first element in List
- Case2: Item does match the first element in List
- Program:

```
delete(Item, [Item|List], List).
delete(Item, [L|Tail1], [L|Tail2]) :-
     delete(Item, Tail1, Tail2).

trace of delete(k, [c,a,k,e], NewList).

1  Call: delete(k,[c,a,k,e],_85) ?
2  Call: delete(k,[a,k,e],_722) ?
3  Call: delete(k,[k,e],_942) ?
3  Exit: delete(k,[k,e],[e]) ?
2  Exit: delete(k,[a,k,e],[a,e]) ?
1  Exit: delete(k,[c,a,k,e],[c,a,e]) ?

NewList = [c,a,e] ?
```

5. sublist

- Make use of existing programs to do new things.
- sublist(S, L).
- We want to know if list S is a sublist of the list L.
- Program:

```
sublist(S, L):-
     conc(_, _, L),
     conc(S, _, L).

trace of sublist([a,b,c], [a,b,c,c,e]).

1  Call: sublist([a,b,c],[a,b,c,c,e]) ?
2  Call: conc(_401,_402,[a,b,c,c,e]) ?
2  Exit: conc([],[a,b,c,c,e],[a,b,c,c,e]) ?
3  Call: conc([a,b,c],_398,[a,b,c,c,e]) ?
4  Call: conc([b,c],_398,[b,c,c,e]) ?
5  Call: conc([c],_398,[c,c,e]) ?
6  Call: conc([],_398,[c,e]) ?
6  Exit: conc([],[c,e],[c,e]) ?
5  Exit: conc([c],[c,e],[c,c,e]) ?
4  Exit: conc([b,c],[c,e],[b,c,c,e]) ?
3  Exit: conc([a,b,c],[c,e],[a,b,c,c,e]) ?
1  Exit: sublist([a,b,c],[a,b,c,c,e]) ?
```

- Problems with this program

```
trace of sublist([a,b,c], [a,k,d,l,c]) with improved program
1  Call: sublist([a,b,c],[a,k,d,l,c]) ?
2  Call: conc([a,b,c],_398,[a,k,d,l,c]) ?
3  Call: conc([b,c],_398,[k,d,l,c]) ?
3  Fail: conc([b,c],_398,[k,d,l,c]) ?
2  Fail: conc([a,b,c],_398,[a,k,d,l,c]) ?
1  Fail: sublist([a,b,c],[a,k,d,l,c]) ?
1  Call: sublist([a,b,c],[a,k,d,l,c]) ?
2  Call: conc(_724,_725,[a,k,d,l,c]) ?
2  Exit: conc([],[a,k,d,l,c],[a,k,d,l,c]) ?
3  Call: conc([a,b,c],_721,[a,k,d,l,c]) ?
4  Call: conc([b,c],_721,[k,d,l,c]) ?
4  Fail: conc([b,c],_721,[k,d,l,c]) ?
3  Fail: conc([a,b,c],_721,[a,k,d,l,c]) ?
2  Redo: conc([],[a,k,d,l,c],[a,k,d,l,c]) ?
3  Call: conc(_950,_725,[k,d,l,c]) ?
3  Exit: conc([],[k,d,l,c],[k,d,l,c]) ?
2  Exit: conc([a],[k,d,l,c],[a,k,d,l,c]) ?
```

```
4  Call: conc([a,b,c],_721,[k,d,l,c]) ?
4  Fail: conc([a,b,c],_721,[k,d,l,c]) ?
2  Redo: conc([a],[k,d,l,c],[a,k,d,l,c]) ?
3  Redo: conc([],[k,d,l,c],[k,d,l,c]) ?
4  Call: conc(_1171,_725,[d,l,c]) ?
4  Exit: conc([],[d,l,c],[d,l,c]) ?
3  Exit: conc([k],[d,l,c],[k,d,l,c]) ?
2  Exit: conc([a,k],[d,l,c],[a,k,d,l,c]) ?
5  Call: conc([a,b,c],_721,[d,l,c]) ?
5  Fail: conc([a,b,c],_721,[d,l,c]) ?
2  Redo: conc([a,k],[d,l,c],[a,k,d,l,c]) ?
3  Redo: conc([k],[d,l,c],[k,d,l,c]) ?
4  Redo: conc([],[d,l,c],[d,l,c]) ?
5  Call: conc(_1391,_725,[l,c]) ?
5  Exit: conc([],[l,c],[l,c]) ?
4  Exit: conc([d],[l,c],[d,l,c]) ?
3  Exit: conc([k,d],[l,c],[k,d,l,c]) ?
2  Exit: conc([a,k,d],[l,c],[a,k,d,l,c]) ?
6  Call: conc([a,b,c],_721,[l,c]) ?
6  Fail: conc([a,b,c],_721,[l,c]) ?
2  Redo: conc([a,k,d],[l,c],[a,k,d,l,c]) ?
3  Redo: conc([k,d],[l,c],[k,d,l,c]) ?
4  Redo: conc([d],[l,c],[d,l,c]) ?
5  Redo: conc([],[l,c],[l,c]) ?
6  Call: conc(_1610,_725,[c]) ?
6  Exit: conc([],[c],[c]) ?
5  Exit: conc([l],[c],[l,c]) ?
4  Exit: conc([d,l],[c],[d,l,c]) ?
3  Exit: conc([k,d,l],[c],[k,d,l,c]) ?
2  Exit: conc([a,k,d,l],[c],[a,k,d,l,c]) ?
7  Call: conc([a,b,c],_721,[c]) ?
7  Fail: conc([a,b,c],_721,[c]) ?
2  Redo: conc([a,k,d,l],[c],[a,k,d,l,c]) ?
3  Redo: conc([k,d,l],[c],[k,d,l,c]) ?
4  Redo: conc([d,l],[c],[d,l,c]) ?
5  Redo: conc([l],[c],[l,c]) ?
6  Redo: conc([],[c],[c]) ?
7  Call: conc(_1828,_725,[]) ?
7  Exit: conc([],[],[]) ?
6  Exit: conc([c],[],[c]) ?
5  Exit: conc([l,c],[],[l,c]) ?
4  Exit: conc([d,l,c],[],[d,l,c]) ?
3  Exit: conc([k,d,l,c],[],[k,d,l,c]) ?
2  Exit: conc([a,k,d,l,c],[],[a,k,d,l,c]) ?
8  Call: conc([a,b,c],_721,[]) ?
8  Fail: conc([a,b,c],_721,[]) ?
```

```
2  Redo: conc([a,k,d,l,c],[],[a,k,d,l,c]) ?
3  Redo: conc([k,d,l,c],[],[k,d,l,c]) ?
4  Redo: conc([d,l,c],[],[d,l,c]) ?
5  Redo: conc([l,c],[],[l,c]) ?
6  Redo: conc([c],[],[c]) ?
7  Redo: conc([],[],[]) ?
7  Fail: conc(_1828,_725,[]) ?
6  Fail: conc(_1610,_725,[c]) ?
5  Fail: conc(_1391,_725,[l,c]) ?
4  Fail: conc(_1171,_725,[d,l,c]) ?
3  Fail: conc(_950,_725,[k,d,l,c]) ?
2  Fail: conc(_724,_725,[a,k,d,l,c]) ?
1  Fail: sublist([a,b,c],[a,k,d,l,c]) ?

yes
{trace}
```

- Preventing Backtracking
- Program with a cut:

```
sublist(S, L):-
     conc(S, _, L), !.

trace of sublist([a,b,c], [a,k,d,l,c]) with a cut

{trace}
| ?- sublist([a,b,c], [a,k,d,l,c]).
   1  1  Call: sublist([a,b,c],[a,k,d,l,c]) ?
   2  2  Call: conc([a,b,c],_398,[a,k,d,l,c]) ?
   3  3  Call: conc([b,c],_398,[k,d,l,c]) ?
   3  3  Fail: conc([b,c],_398,[k,d,l,c]) ?
   2  2  Fail: conc([a,b,c],_398,[a,k,d,l,c]) ?
   1  1  Fail: sublist([a,b,c],[a,k,d,l,c]) ?

no
{trace}
```

6. permutation

- A permutation is a bijection onto itself (n!).
- Program perm(List, X) with backtracking:

```
perm([], []).
perm(List, [X|Tail] ) :-
        delete(X, List, List1),
        perm(List1, Tail).
```

```
trace perm(List, X):

{trace}
| ?- trace, perm([a,b,c],X).
{The debugger will first creep -- showing everything (trace)}
   1  Call: perm([a,b,c],_93) ? s
   1  Exit: perm([a,b,c],[a,b,c]) ?

X = [a,b,c] ? ;
   1  Redo: perm([a,b,c],[a,b,c]) ? s
   1  Exit: perm([a,b,c],[a,c,b]) ?

X = [a,c,b] ? ;
   1  Redo: perm([a,b,c],[a,c,b]) ? s
   1  Exit: perm([a,b,c],[b,a,c]) ?

X = [b,a,c] ? ;
   1  Redo: perm([a,b,c],[b,a,c]) ?
   2  Redo: perm([a,c],[a,c]) ? s
   2  Exit: perm([a,c],[c,a]) ?
   1  Exit: perm([a,b,c],[b,c,a]) ?

X = [b,c,a] ? ;
   1  Redo: perm([a,b,c],[b,c,a]) ? s
   1  Exit: perm([a,b,c],[c,a,b]) ?

X = [c,a,b] ? ;
   1  Redo: perm([a,b,c],[c,a,b]) ? s
   1  Exit: perm([a,b,c],[c,b,a]) ?

X = [c,b,a] ? ;
   1  Redo: perm([a,b,c],[c,b,a]) ? s
   1  Fail: perm([a,b,c],_93) ?

no
{trace}
```

- Program permutation(List) with built-in function ‘findall’:

```
permutation(List) :-
        findall(One, perm(List, One), Perms),
        output(Perms).

perm([], []).
perm(List, [X|Tail] ) :-
        delete(X, List, List1),
        perm(List1, Tail).
```

```
output([]).
output([P|Ps]) :-
        write(P), nl,
        output(Ps).

trace permutation(List):
aristotle[119]% sicstus
SICStus 2.1 #9: Wed Oct 11 15:33:09 CST 1995
| ?- ['library.pl'].
{consulting /u/jian/public_html/420/sample/library.pl...}
{/u/jian/public_html/420/sample/library.pl consulted, 200 msec 16736 bytes}

yes
| ?- trace, permutation([a,b,c]).
{The debugger will first creep -- showing everything (trace)}
1  Call: permutation([a,b,c]) ?
2  Call: findall(_675,user:perm([a,b,c],_675),_677) ?
3  Call: perm([a,b,c],_675) ? s
3  Exit: perm([a,b,c],[a,b,c]) ?
3  Redo: perm([a,b,c],[a,b,c]) ? s
3  Exit: perm([a,b,c],[a,c,b]) ?
3  Redo: perm([a,b,c],[a,c,b]) ? s
3  Exit: perm([a,b,c],[b,a,c]) ?
3  Redo: perm([a,b,c],[b,a,c]) ? s
3  Exit: perm([a,b,c],[b,c,a]) ?
3  Redo: perm([a,b,c],[b,c,a]) ? s
3  Exit: perm([a,b,c],[c,a,b]) ?
3  Redo: perm([a,b,c],[c,a,b]) ? s
3  Exit: perm([a,b,c],[c,b,a]) ?
3  Redo: perm([a,b,c],[c,b,a]) ? s
3  Fail: perm([a,b,c],_675) ?
2  Exit: findall(_675,user:perm([a,b,c],_675),[[a,b,c],[a,c,b],
                               [b,a,c],[b,c,a],[c,a,b],[c,b,a]]) ?
2  Call: output([[a,b,c],[a,c,b],[b,a,c],[b,c,a],[c,a,b],[c,b,a]]) ? s
[a,b,c]
[a,c,b]
[b,a,c]
[b,c,a]
[c,a,b]
[c,b,a]
2  Exit: output([[a,b,c],[a,c,b],[b,a,c],[b,c,a],[c,a,b],[c,b,a]]) ?
1  Exit: permutation([a,b,c]) ?
```

7. operator notation

- improve readability of programs

- operator types: infix (xfx, xfy, yfx), prefix (fx, fy), postfix (xf, yf)
- where the precedence of x < f while the precedence of y $\leq$ f and they are used to disambiguate expersssions.
- infix has two arguements and one operator while prefix has one arguement and one operator
- Examples:
  - yfx: a - b - c $\rightarrow$ (a - b) - c
  - xfy: a - b - c $\rightarrow$ a - (b - c)
  - prefix (fx): not not P $\rightarrow$ not (not P)
  - prefix (fy): not not P $\rightarrow$ (not not) P
- define an operator
- Example: $\tilde{}$(A & B) $\iff$ $\tilde{A} \vee \tilde{B}$ can be represented in Prolog after defining some operators

```
:- op(800, xfx, '<===>').
:- op(700, xfy, 'v').
:- op(600, xfy, '&').
:- op(500, fy, '~').

~(A & B) <===> ~A v ~B.
```

- the numbers presents the precedence of the operators

8. **arithmetic**

- operators used for basic arithmetic: +, -, *, /, //, mod (modulo, the remainder of integer division)
- for 'X = 1 + 2' the answer is not 'X = 3' but 'X = 1 + 2'
- The operator *is* is provided to force evaluation.
- for 'X is 1 + 2' the answer is 'X = 3'
- for 'X is 10 / 3' the answer is 'X = 3.3333333333333335'
- for 'X is 10 // 3' the answer is 'X = 3'
- operators for comparison: >, <, >=, =<, =:=, = \ =
- Example: find the greatest common diviser fo two numbers.
- Program:

```
gcd(X, X, X).
gcd(X, Y, D) :-
     X < Y,
     Y1 is Y - X,
gcd(X, Y1, D).
gcd(X, Y, D) :-
     Y < X,
     gcd(Y, X, D).

1. trace of gcd(3, 5, D).

 1  Call: gcd(3,5,_93) ?
 2  Call: 3<5 ?
 2  Exit: 3<5 ?
 3  Call: _372 is 5-3 ?
 3  Exit: 2 is 5-3 ?
 4  Call: gcd(3,2,_93) ?
 5  Call: 3<2 ?
 5  Fail: 3<2 ?
 5  Call: 2<3 ?
 5  Exit: 2<3 ?
 6  Call: gcd(2,3,_93) ?
 7  Call: 2<3 ?
 7  Exit: 2<3 ?
 8  Call: _2050 is 3-2 ?
 8  Exit: 1 is 3-2 ?
 9  Call: gcd(2,1,_93) ?
10  Call: 2<1 ?
10  Fail: 2<1 ?
10  Call: 1<2 ?
10  Exit: 1<2 ?
11  Call: gcd(1,2,_93) ?
12  Call: 1<2 ?
12  Exit: 1<2 ?
13  Call: _3728 is 2-1 ?
13  Exit: 1 is 2-1 ?
14  Call: gcd(1,1,_93) ?
14  Exit: gcd(1,1,1) ?
11  Exit: gcd(1,2,1) ?
 9  Exit: gcd(2,1,1) ?
 6  Exit: gcd(2,3,1) ?
 4  Exit: gcd(3,2,1) ?
 1  Exit: gcd(3,5,1) ?
D = 1
```

- Example: count the length of a list
- Program:

```
            countList([], 0).
            countList([_|Tail], N) :-
                countList(Tail, N1),
                  N is 1 + N1.

2. trace of countList([a,b,d,e,f], N).

Call: countList([a,b,d,e,f],_99) ?
Call: countList([b,d,e,f],_389) ?
Call: countList([d,e,f],_614) ?
Call: countList([e,f],_838) ?
Call: countList([f],_1061) ?
Call: countList([],_1283) ?
Exit: countList([],0) ?
Call: _1061 is 1+0 ?
Exit: 1 is 1+0 ?
Exit: countList([f],1) ?
Call: _838 is 1+1 ?
Exit: 2 is 1+1 ?
Exit: countList([e,f],2) ?
Call: _614 is 1+2 ?
Exit: 3 is 1+2 ?
Exit: countList([d,e,f],3) ?
Call: _389 is 1+3 ?
Exit: 4 is 1+3 ?
Exit: countList([b,d,e,f],4) ?
Call: _99 is 1+4 ?
Exit: 5 is 1+4 ?
Exit: countList([a,b,d,e,f],5) ?
N = 5
```

9. **built-in predicates:**

   - Built-in predicates come with the system. They do not return values but true or false statements.
   - Example: <, >, = / =, ==

10. **buit-in functions:**

   - Built-in functions are also come with system. They return values.
   - abs(X), min(X, Y), max(X, Y), round(X), sin(X), cos(X), sqrt(X), log(X), exp(X, Y), ..., (See SICStus Prolog Manual, p. 50-52).
   - Example:

```
aristotle[45]% sicstus
SICStus 2.1 #9: Wed Oct 11 15:33:09 CST 1995
| ?- X is abs(-3.4).
X = 3.4 ?
yes
```

```
| ?- X is min(78, 3).
X = 3 ?
yes

| ?- X is max(5, 6).
X = 6 ?
yes

| ?- X is round(78.345).
X = 78.0 ?
yes
```

11. **more buit-in predicates and functions** Explain the handout 'A list of built-ins of Sicstus Prolog'.

Part 2. Debugging, Backtracking, Input and Output

Debugging

- A good principle of debugging is start with a smaller problem.
- trace – the goal's satisfaction is displayed during execution
- notrace – turn off trace
- spy/1 – spy on a particular function
- nospy/1 – turn off the spy point
- Some usful commands: (there is no on-line help), + (spy this), - (no spy this), a (abort), b (break), d (display), f (fail), r (retry), s (skip),
- Example:

```
| ?- trace, f(1, Y), 2 < Y.
{The debugger will first creep -- showing everything (trace)}
   1  Call: f(1,_65) ? h
Debugging options:
   <cr>   creep            c      creep
    l     leap             s      skip
    r     retry            r <i>  retry i
    f     fail             f <i>  fail i
    d     display          w      write
    p     print            p <i>  print partial
    g     ancestors        g <n>  ancestors n
    &     blocked goals    & <n>  nth blocked goal
    n     nodebug          =      debugging
    +     spy this         + <i>  spy conditionally
    -     nospy this       .      find this
    a     abort            b      break
    @     command          u      unify
    e     pending exception
    <     reset printdepth < <n>  set printdepth
    ^     reset subterm    ^ <n>  set subterm
    ?     help             h      help
   1  Call: f(1,_65) ?
   2  Call: 1<3 ? s
   2  Exit: 1<3 ?
   1  Exit: f(1,0) ?
   1  Call: 2<0 ? +
{Warning: Spying on prolog: < /2 is risky}
{Spypoint placed on prolog: < /2}
 + 1  Call: 2<0 ?
 + 1  Fail: 2<0 ?
   1  Redo: f(1,0) ? -
{There is no spypoint on user:f/2}
   1  Redo: f(1,0) ?
 + 2  Redo: 1<3 ? -
{Warning: Spying on prolog: < /2 is risky}
{Spypoint removed from prolog: < /2}
   2  Redo: 1<3 ? d
   2  Redo: <(1,3) ?
   2  Fail: 1<3 ?
   2  Call: 1=<3 ? f
   2  Fail: 1=<3 ? r
   2  Call: 1=<3 ?
   2  Exit: 1=<3 ?
   2  Call: 1<6 ? b
{Break level 1}
{1}
```

Backtracking

- The process of reviewing the goals that have been satisfied and attempting to resatisfy these goals by finding alternative solutions.
- Prolog will automatically backtrack for satisfying a goal.
- Example: The permutation program on page 79 of Bratko book which we have showd in class.
- To control unnecessary backtracking, we use the *cut* facility.
- Example 1:

```
Rule 1: if X < 3 then Y = 0
Rule 2: if 3 =< X and X < 6 then Y = 2
Rule 3: if 6 =< X then Y = 4

Prolog program f(X, Y):
f(X, 0) :- X < 3.
f(X, 2) :- X >= 3, X < 6.
f(X, 4) :- X >= 6.

Trace of f(X, Y):
| ?- trace, f(1, Y), 2 < Y.
{The debugger will first creep -- showing everything (trace)}
   1  Call: f(1,_65) ?
   2  Call: 1<3 ?
   2  Exit: 1<3 ?
   1  Exit: f(1,0) ?
   1  Call: 2<0 ?
   1  Fail: 2<0 ?
   1  Redo: f(1,0) ?
   2  Redo: 1<3 ?
   2  Fail: 1<3 ?
   2  Call: 1>=3 ?
   2  Fail: 1>=3 ?
   2  Call: 1>=6 ?
   2  Fail: 1>=6 ?
   1  Fail: f(1,_65) ?

no
{trace}



Prolog program f1(X, Y):
f1(X, 0) :- X < 3, !.
f1(X, 2) :- X =< 3, X < 6, !.
f1(X, 4) :- X =< 6.

| ?- trace, f1(1, Y), 2 < Y.
{The debugger will first creep -- showing everything (trace)}
   1  Call: f1(1,_85) ?
   2  Call: 1<3 ?
   2  Exit: 1<3 ?
   1  Exit: f1(1,0) ?
   1  Call: 2<0 ?
   1  Fail: 2<0 ?
   1  Redo: f1(1,0) ?
   1  Fail: f1(1,_85) ?
```

```
no
{trace}
```

The cut in Example 1 changed the procedrual meaning of the prolog.

- Example 2:

```
| ?- trace, f(7, Y).
{The debugger will first creep -- showing everything (trace)}
   1  Call: f(7,_65) ?
   2  Call: 7<3 ?
   2  Fail: 7<3 ?
   2  Call: 7>=3 ?
   2  Exit: 7>=3 ?
   2  Call: 7<6 ?
   2  Fail: 7<6 ?
   2  Redo: 7>=3 ?
   2  Fail: 7>=3 ?
   2  Call: 7>=6 ?
   2  Exit: 7>=6 ?
   1  Exit: f(7,4) ?

Y = 4 ?
yes
{trace}
```

All three rules are tried. If we change the program as the follows, the program will be guaranteed to be true.

```
Prolog program f2(X, Y):
f2(X, 0) :- X < 3, !.
f2(X, 2) :- X < 6, !.
f2(_, 4).
```

But if the cuts are omited, the program may produce some incorrect answers.

```
Prolog program f3(X, Y):
f3(X, 0) :- X < 3.
f3(X, 2) :- X < 6.
f3(_, 4).

Output for f3(X, Y):

| ?- f3(1, Y).

Y = 0 ? ;
```

```
Y = 2 ? ;
Y = 4 ? ;

no
```

Note: this time the cuts do not only affect the procedural behaviour, but also change the declaretive meaning (logical) of the program.

- A cut is a built-in predicate that succeeds when encountered. if backtracking should later return to the cut, the goal that matched the head of the clause containing the cut fails immediately. All the alternatives are discarded.
- More examples using cut:

```
max(X, Y, Max) :- X >= Y.
max(X, Y, Max) :- X < Y.
```

Since the rules are mutually exclusive, a more economical way is using cut to preventing back tracking.

```
max(X, Y, Max) :- X >= Y, !.
max(X, Y, Y).
```

Let us go back to our membership program:

```
member(X, [X|_]).
member(X, [_|Tail]) :-
member(X, Tail).

Output of member(X, List):

| ?- member(X, [a,b,c,a,b,c]).

X = a ? ;
X = b ? ;
X = c ? ;
X = a ? ;
X = b ? ;
X = c ? ;

no
```

If the cut is added to the first clause, the answer will be:

```
| ?- member1(X, [a,b,c,a,b,c]).

X = a ? ;

no
| ?-
```

- Negation as Failure
  - *not goal* is defined through the failure of the *goal.*
  - Example: Mary likes all animals but snakes.

    ```
    likes(mary, X) :-
         snake(X), !, fail.

    likes(mary, X) :-
         animal(X).
    ```

    The first clause will take care of snakes: if X is a snake then the cut will prevent backtracking (thus excluding the second clause) and fail will cause the failure of the query 'likes(mary, snake)'.

    ```
    likes(mary, X) :-
         snake(X), !, fail; animal(X).
            or
    likes(mary, X) :-
         animal(X), \+ snake(X).
    ```

- Green Cut and Red Cut
  - Example:

    ```
    p :- a, b.
    p :- c.

    which can be written as a logic formula:

    p <==> (a & b) V c
    ```

    If we change the order of the two clause, the declarative meaning of this program is remains the same. Let us now insert a cut:

    ```
    p :- a, !, b.
    p :- c.
    ```

    The declarative meaning is now:

```
p <==> (a & b) V (~a & c)
```

If we swap the clauses,

```
p :- c.
p :- a, !, b.
```

The declarative meaning becomes:

```
p <==> c V (a & b).
```

- A cut has effect on the declarative meaning is called a *red cut*; while a cut has effect only on the procedural meaning is called a *green cut*.
- A green cut is less delicate than a red cut. A red cut should be used with special care.

Input and Output

- communication with files
  - Prolog can open several files for input and output.
  - But at any time during execution of a Prolog program, only two them are active. One for input and one for output.
  - The built-in predicates *see/1* and *seen/0* opens and closes files for read; *tell/1* and *told/0* opens and closes files for write.
  - Example 1: tell(in_file), see(out_file), read(X), write(X), ...
  - Example 2: tell(user), see(outline), ..., told, seen.
  - Example 3: tell(in_file), see(user), ..., told, seen.
  - Example 4: tell(out_file1), see(in_file1), do some thing, tell(out_file2), see(in_files2), do some thing, ...
- read and write
  - Example 1: tell(out_file), read(X), write(X).
  - Example 2: write('Prolog is a AI programming language.').
  - Example 3:

    ```
    out_put([]).
    out_put([X|Tail]) :-
         write(X), tab(1), nl,
         out_put(Tail).
    ```

  - Example 4: on page 152, Bratko.
- manipulating characters

  - put/1 – output a character where C is the ASCII code of a character
  - Example: put(65), put(66), put(67). output will be ABC
  - get0/1 – read in a character where C is the ASCII code of a character
  - get/1 – read in non-blank characters

- The difference between read/1, write/1 and get/1, put/1 is that read/1 can input a term (terminated with a full stop) while get/1 only get one charicter.

```
| ?- get(C), put(C).
|: a.
a
C = 97 ?

| ?- read(C), write(C).
|: a.
a
C = a ?
```

- the function *name*
  - name(Atom, ASCII) – encoding ASCII to Atom or Atom to ASCII

- Examples of using the built-in function *name(2)*:

```
atoc(Ascii) :-
     name(Atom, [Ascii]),
     write('This is the character: '),
     write(Atom), nl.

ctoa(Atom) :-
     name(Atom, [Ascii]),
     write('This is the ASCII code: '),
     write(Ascii), nl.

 Some output of atoc(Ascii) and ctoa(Atom):
 | ?- ['library.pl'].
 {consulting /u/jian/public_html/420/sample/library.pl...}
 {/u/jian/public_html/420/sample/library.pl consulted, 150 msec 13712 bytes}

 yes
 | ?- atoc(68).
 This is the character: D

 yes
 | ?- ctoa('C').
 This is the ASCII code: 67

 yes
```

Programming Style and Techniques

1. **General principles of good programming**

   - **correctness** – should be correct. That is, the program should do what it is supposed to do. A common mistake when writing a program is to neglect this obvious criterion and pay more attention to other criteria, such as efficiency.
   - **efficiency** – should not waste computer time and memory space.
   - **readability** – should be easy to read and understand.
   - **modifiability** – should be easy to modify and to extend.
   - **robustness** – should be 'strong' enough to stay alive when incorrect or unexpected input entered and report the errors.
   - **documentation** – should be properly documented.

2. **How to think about Prolog programs?**

   - Prolog has two features: declarative and procedural (had given many examples)
   - Use of recursion: split the problem into trivial (boundary) cases and general cases.
   - Example: transform a list of letters into a list of ASCII code
   - name/2, a built-in function (see handout) which convert an atom into its Ascii code. For example, name(a, [97]).

   ```
   transf([], []).
   transf([L|Ls], [T|Ts]) :-
        name(L, [T]),
        transf(Ls, Ts).

   output of transf(2):
   | ?- trace, transf([a,b,c], Ascii).
   {The debugger will first creep -- showing everything (trace)}
      1  1  Call: transf([a,b,c],_73) ?
      2  2  Call: name(a,[_705]) ?
      2  2  Exit: name(a,[97]) ?
      3  2  Call: transf([b,c],_706) ?
      4  3  Call: name(b,[_1308]) ?
      4  3  Exit: name(b,[98]) ?
      5  3  Call: transf([c],_1309) ?
      6  4  Call: name(c,[_1910]) ?
      6  4  Exit: name(c,[99]) ?
      7  4  Call: transf([],_1911) ?
      7  4  Exit: transf([],[]) ?
      5  3  Exit: transf([c],[99]) ?
      3  2  Exit: transf([b,c],[98,99]) ?
      1  1  Exit: transf([a,b,c],[97,98,99]) ?
   ```

```
Ascii = [97,98,99] ?
```

- Generalization: generalize the original problem so that the solution to the generalized problem can be formulated recursively. The original problem is then solved as a special case.
- Example: add_to_list(Item, List, Newlist).
- change to: add_to_list(Item, Pos, List, Newlist).

3. **Some rules of good style**

   - Program clauses should be short. Do not contain a lot of goals in the body.
   - Names of variables, predicates, facts, should be meaningful.
   - The layout is important:
     - spacing, blank lines should be used for readability
     - clauses about the same procedure should be clustered together
     - indent the body of clauses
   - Avoiding using cuts especially red cuts
   - Commenting: what the program does, how it is used, what is the expected results,...