

**BAHIR DAR UNIVERSITY
BAHIR DAR INSTITUTE OF TECHNOLOGY
SCHOOL OF COMPUTING**

Introduction to Artificial Intelligence

SertseA

3/11/2015

| Lab .Se | Topics | Pag e |
|------------------|--|----------|
| Lab1 | Introducing Prolog and SWI-Prolog Environment..... | 2 |
| Lab2 | Syntax and Data objects of Prolog..... | 10 |
| Lab3 | Mathematical and logical operation in Prolog..... | 15 |
| Lab4 | Cuts and negation | 18 |
| Lab 5 | Lists and Recursion in Prolog..... | 22 |
| Lab 6 | Predefined predicates of List important in AI..... | 26 |
| Lab 7 | Input Output and Interfacing in Prolog..... | 31 |
| Lab 8 | AI in Prolog fact representation and implementation of different AI function.... | 33 |
| Lab 9 and 10 | Implementation of Breadth and Depth First Search In Prolog goal findingí í . | 35 |
| Lab 11 and 12 | Implementation of Breadth and Depth First Search In Prolog path finding and path Checking..... | 38 |
| Lab13 | Implementation of Heuristic Function in Prolog. Straight line distance and Manhattan distance cost computation in prolog..... | 41 |

Introduction

This Laboratory manual is the first volume by the author for the course Introduction to Artificial intelligence (Cosc-3071) using Prolog programming language. The manual is designed by including concepts that can provide students some fundamental concepts of logic programming using SWI-Prolog Compilers. On this regard the manual contains:

- practices and introductory contents which shows how to write and compiles logic programs using SWI-Prolog Compilers
- practical secessions and exercises which deals with the fundamental concepts of logic programming such as description of the knowledge bases (facts and rules) and queries for searching a solution for a specific demand of information and knowledge of a user in the domain of a problem

The manual also includes introductory topics which provides a highlight for the learner about syntaxes and semantics of prolog programming. On this regards practical secessions and exercises on writing and executing fundamental prolog programmes which deal with syntaxes made from the programming elements such as:

- terms and variables declaration and usages.
- operators (arithmetic, assignment, and logical operators) usage .
- list and recursions.
- Input/output techniques and other specific techniques of prolog programming has been discussed very well.

The objective of all the above concepts and techniques is to enable students to write and analyze Artificial intelligence problems using logic programming. thus on this aspect the manual consists practices and exercises that implements and common AI techniques such as problem and knowledge representation, state space representation, AI searching (different tree searching) to students.

Laboratory Session One: Introducing Prolog and its Environment

Objectives of the Session

The objective of this session is to introduce students how they:

- start and use SWI-Prolog Compiler.
- write and execute their first prolog program.
- describe queries in SWI-Prolog Environment

Outcomes

- it enables students to open and close SWI-Prolog Compilers
- It enables students to write and execute codes in SWI- prolog environment.

I. What is Prolog?

Prolog (programming in logic) is one of the most widely used programming languages in artificial intelligence research. As opposed to imperative languages such as C or Java (the latter of which also happens to be object-oriented) it is a declarative programming language.

That means, when implementing the solution to a problem, instead of specifying how to achieve a certain goal in a certain situation, we specify what the situation (rules and facts) and the goal (query) are and let the Prolog interpreter derive the solution for us.

Prolog is very useful in some problem areas, such as artificial intelligence, natural language processing, and databases and so on but pretty useless in others, such as graphics or numerical algorithms.

A program is partly like a database but much more powerful since we can also have general rules to infer new facts!

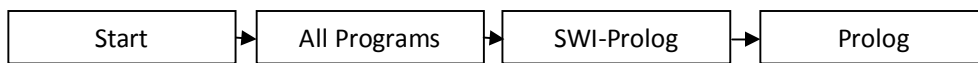
A Prolog interpreter can follow these facts/rules and answer queries by sophisticated search. There are different version and products of prolog interpreters. The Commonly used interpreters in academic environment for prolog is SWI-Prolog.

II. Introduction to SWI PROLOG

Swi-Prolog can be freely downloaded from <http://www.swi-prolog.org> and can easily install it.

Starting Prolog and loading a program in a Swi-prolog

The SWI-Prolog executable swipl-win.exe can be started from the Start Menu or by opening a .pl file holding Prolog program text from the Windows explorer. The .PL file extension can be changed during installation. See section 3.2.



1.3 Create a source file of prolog

There are two options

A. Using Notepad Editor

- ✓ Open notepad Using Windows start menu
- ✓ Write the source code
- ✓ Save the source code file in a pl extension (mysample1.pl)

B. Using Swi-Prolog Editor

- ✓ After you open Swi_Prolog interpreter
- ✓ Click file from the Menu bar
- ✓ Click New and from the file menu
- ✓ A new save as dialog box will appear
- ✓ Provide name and save the new pl file and write the source codes later

1.4 Consulting a source file

- ✓ 1. In a prolog window click file menu
- ✓ 2. Click consult
- ✓ 3. Select appropriate file and press open button

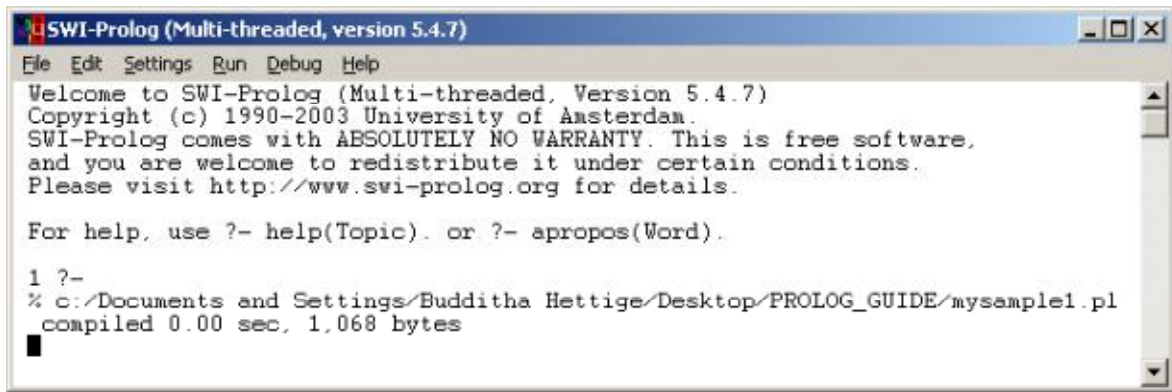


Fig 1.1 SWI-Prolog interfaces

Getting Started Prolog Programming:

Practice 1: bigger.pl a program that tell us the relation between animals using logic programming

In the introduction it has been said that Prolog is a declarative (or descriptive) language.

- Programming in Prolog means describing the world.
- Using such programs means asking Prolog questions about the previously described world.
- The simplest way of describing the world is by stating facts, like the following one:

Let us assume that, we have some simple fact that we want to declare like 'elephant is bigger animal than horse' for the prolog compiler. The first step is generating a new prolog editor either of the described above way and write the following structure.

```
bigger (elephant, horse).
```

And save the source code with *bigger.pl* file name(see the picture described in fig 1.2).

The above states, quite intuitively describes, the fact that an elephant is bigger than a horse. (Whether the world described by a Prolog program has anything to do with our real world is, of course, entirely up to the programmer imagination.)

Let's add a few more facts to our little program:

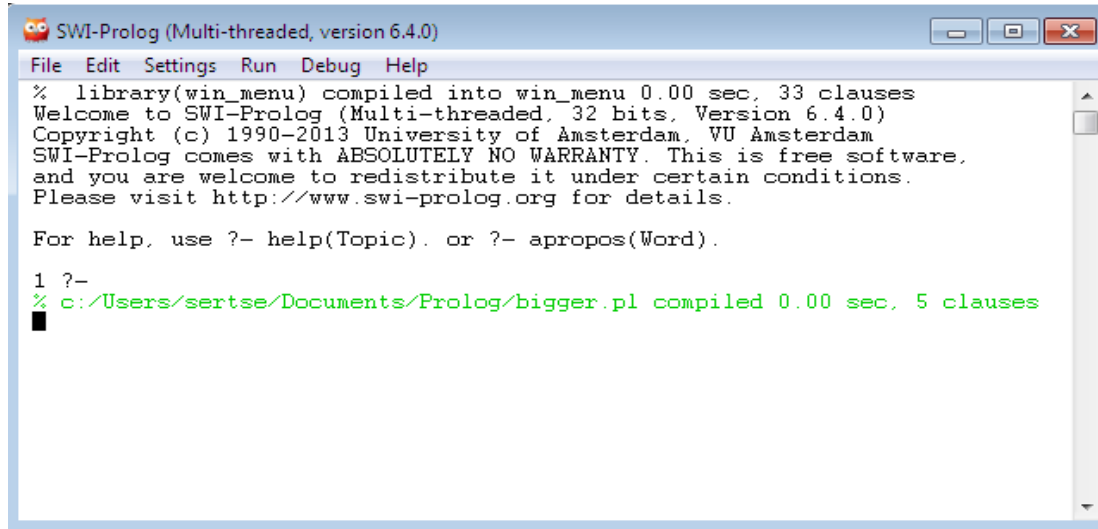
```
bigger(elephant, horse).
bigger(horse, donkey).
bigger(donkey, dog).
bigger(donkey, monkey).
```

Fig 1.2 source code with prolog editor

This is a syntactically correct program, and after having compiled it we can ask the Prolog system questions (or queries in proper Prolog-jargon) about it. Here's an example how we can provides questions (or queries in proper Prolog-jargon):

Even though there are different ways for presenting queries for prolog engine, queries for prolog system in SWI-prolog environment queries usually presented in the interactive interface of the SWI-prolog interpreter shown in the following picture.

In this environment prior to query presentation, the user first should call the prolog source file (bigger.pl) to memory buffer for consultation of the stored facts and rules in the file usually named as knowledge base for some domain specific query need of a user. (Refer consulting source file topics in the previous portion).



```
SWI-Prolog (Multi-threaded, version 6.4.0)
File Edit Settings Run Debug Help
% library(win_menu) compiled into win_menu 0.00 sec, 33 clauses
Welcome to SWI-Prolog (Multi-threaded, 32 bits, Version 6.4.0)
Copyright (c) 1990-2013 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

1 ?-
% c:/Users/sertse/Documents/Prolog/bigger.pl compiled 0.00 sec, 5 clauses
■
```

Fig 1.3 interactive interface of SWI-Prolog

The user of bigger.pl knowledge base may present the following sample queries and can obtain the respected answer see fig 1.4.

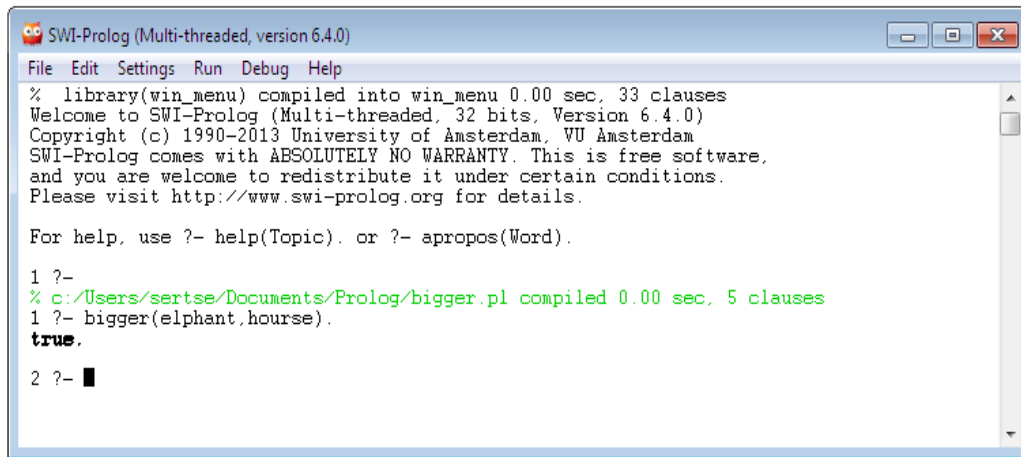


Fig 1.4 user query and answer of the prolog engine in the interactive interface

In a similar fashion we can present different queries for the prolog engine. The following are a few instances of user query and the answer from the prolog engine that consulted the prolog engine.

```
?- bigger(donkey, dog). The answer for this query will be
true

Or

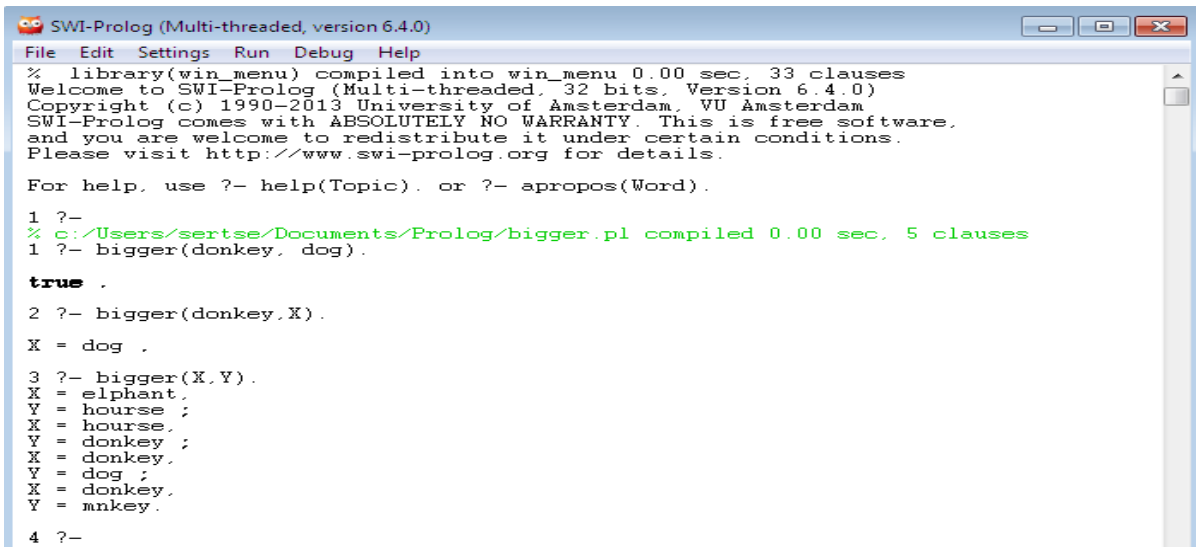
?-bigger(donkey,X). the answer will be
Y = dog ;
Y = monkey.

or

?- bigger(X,Y).the answer will be
X = elephant,
Y = horse ;
X = horse,
Y = donkey ;
X = donkey,
Y = dog ;
X = donkey,
Y = monkey.

or

?- bigger(monkey, elephant).
The answer will be false
```

```
SWI-Prolog (Multi-threaded, version 6.4.0)
File Edit Settings Run Debug Help
% library(win_menu) compiled into win_menu 0.00 sec, 33 clauses
Welcome to SWI-Prolog (Multi-threaded, 32 bits, Version 6.4.0)
Copyright (c) 1990-2013 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

1 ?-
% c:/Users/sertse/Documents/Prolog/bigger.pl compiled 0.00 sec, 5 clauses
1 ?- bigger(donkey, dog).

true .

2 ?- bigger(donkey, X) .
X = dog .

3 ?- bigger(X, Y) .
X = elephant,
Y = hourse ;
X = hourse,
Y = donkey ;
X = donkey,
Y = dog ;
X = donkey,
Y = mnkey .

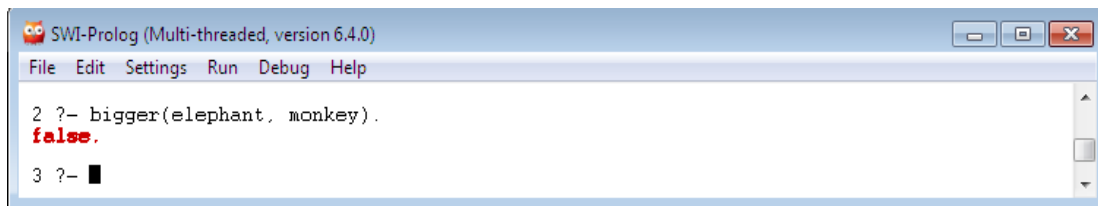
4 ?-
```

Fig 1.6 queries and positive responds as interpreted by SWI-Prolog engine

All the answers we got are exactly the answer we expected?

But what happens when we ask the other way round?

?- bigger(elephant, monkey).



```
SWI-Prolog (Multi-threaded, version 6.4.0)
File Edit Settings Run Debug Help

2 ?- bigger(elephant, monkey).
false.

3 ?-
```

Fig 1.7 queries and false responds for unknown facts interpreted by SWI-Prolog engine

According to this elephants are not bigger than monkeys. This is clearly wrong as far as our real world is concerned, but if you check our little program again (check contents of bigger.pl at fig 1.2), you will find that it says nothing about the relationship between elephants and monkeys. This kind of problem can be solved by generalized rules.

Practices: 2 Family.pl

Let us see other features of prolog programming with family.pl. Sometimes it may be too tedious to describe every real world facts to prolog engine explicitly through fact description. Instead, if there are some common features and clear relationships among real world facts, programmer may write rules in their knowledge base that used to generalize about facts for prolog engine from available facts during interpretation of queries (see fig 1.8). In this example the programmer define rules for father, mother, daughter, and son facts using available facts such as parent, male, and female predicates.

```

male(abebe).
male(tadesse).
female(aster).
female(almaz).m
parent(abebe,tadesse).
parent(almaz,tadesse).
parent(abebe,aster).
parent(almaz,aster).
father(X,Y):-parent(X,Y),male(X).
mother(X,Y):-parent(X,Y),female(X).
doughter(X,Y):-parent(Y,X),female(X).
son(X,Y):-parent(Y,X),male(X).

```

Fig 1.8 knowledge base for family relationship made using facts and rules

In the above example one can see that, facts such as parent, male, and female predicates are written in different format than their precedent set of facts. First the argument for each predicates written in capital letter (variable writing form for prolog programming) since, variables usually used to generalize about some common features of objects in the real world.

Notes: Variables in prolog programming written using alphanumeric character that starts with capital letter and constant facts written using any combination of alphanumeric characters that starts with small letters. (Next time we will see in detail about prolog object type).

Figure 1.9 shows queries and responds of SWI-Prolog engine by consulting family.pl knowledge base.

```

SWI-Prolog (Multi-threaded, version 6.4.0)
File Edit Settings Run Debug Help
1 ?- father(X,Y).
X = abebe,
Y = tadesse ;
X = abebe,
Y = aster ;
false.

2 ?- father(abebe,X).
X = tadesse ;
X = aster.

3 ?- father(abebe,aster).
true.

4 ?- father(abebe,tadesse).
true ;
false.

5 ?- mother(X,Y).
X = almaz,
Y = tadesse ;
X = almaz,
Y = aster.

6 ?- ■

```

Exercise of Lab session one

1. Take your own case and write prolog codes and test with different form of queries
2. Consider family.pl as your knowledge base and define other rule for brother, sister, and father and mother.

Laboratory Seession Two: Syntax of Prolog

Objectives of this laboratory seession is to provide students an insight and practical skills about object structure of a prolog language and its usage in different data representation.

Learning Outcomes

It enables students to be familiarize with prolog data object structure

It enables students to be familiarize with syntaxs of prolog in creating the basic data objects of prolog (terms, variables).

Data objects/Terms

Types of objects that a prolog interpreter recognizes by its syntactic forms are called data-object/terms.

The central data structure in Prolog is that of a term. There are terms of four kinds: atoms, numbers, variables, and compound terms. Atoms and numbers are sometimes grouped together and called atomic terms or simple object. A combination of two or more atomic terms can form compound term or structure.

Simple objects (terms) can be a static facts (constant) that represent specific real-world facts or events or a generalized fact that can symbolically represent similar real-world objectsø fact called variable. We have already seen the different between variables and constants terms in the previous section examples.

Variable start with upper case letter while constant start with small letter. No additional information such as data type declaration has been communicated to prolog in order to recognize the objects.

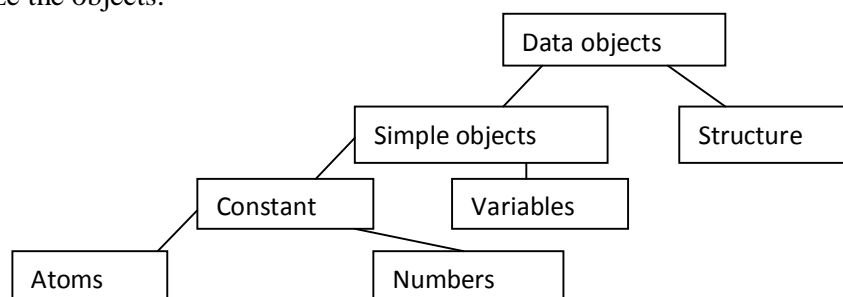


Fig 2.1 anatomy of terms

Static facts (constant) can be either atoms or numbers.

Atoms: Atoms are usually strings made up of lower- and uppercase letters, digits, and the underscore, starting with a lowercase letter. The following are all valid Prolog atoms:

Example

elephant, b, abcXYZ, x_123, another_pint_for_me_please

On top of that also any series of arbitrary characters enclosed in single quotes denotes an atom.

'This is also a Prolog atom.'

Finally, strings made up solely of special characters like + - * = < > : & (check the manual of our SWI-Prolog system for the exact set of these characters) are also atoms.

Examples:

+, ::, <----->, ***

Numbers. All Prolog implementations have an integer type: a sequence of digits, optionally preceded by a - (minus) for negative number.

Variables. Variables are strings of letters, digits, and the underscore, starting with a capital letter or an underscore. Examples:

X, Elephant, _4711, X_1_2, MyVariable, _

The last one of the above examples (the single underscore) constitutes a special case. It is called the anonymous variable and is used when the value of a variable is of no particular interest. Multiple occurrences of the anonymous variable in one expression are assumed to be distinct, i.e., their values don't necessarily have to be the same.

Structure/Compound terms.

Compound terms are made up of a functor (a Prolog atom) and a number of arguments (Prolog terms, i.e., atoms, numbers, variables, or other compound terms) enclosed in parentheses and separated by commas.

The following are some examples for compound terms:

```
is_bigger(horse, X),
f(g(X, _), 7),
'My Functor'(dog)
```

Structured objects (or simply structures are objects that have several components. The components themselves can, in turn, be structures. For example,



Date is an example of structured or compound terms as it is represented in a tree and as it is represented in a prolog.

The date can be viewed as a structure with three components day, month, and year. Although composed of several components, structures are treated in the program as single objects. In order to combine the components in to a single object we have to choose a function

You can write more structure forms in the following way

```
haliday('Ethiopian ChristMass',date(january,7,2013)).
haliday('Eid_al_adha',date(january,10,2013)).
haliday('Epiphany',date(january,19,2013)).
haliday('Adwa Victory day',date(march,2,2013)).
haliday('The phropheet Birth Day',date(april,11,2013)).
haliday('Ethiopian Good Friday',date(april,21,2013)).
haliday('Ethiopian Easter Sunday',date(april,23,2013)).
haliday('International Labor Day',date(may,1,2013)).
haliday('Freedom Day',date(may,5,2013)).
haliday('Derg Downfall Day',date(may,28,2013)).
haliday('Ethiopian New Year',date(september,11,2013)).
haliday('Finding of the True
Cross',date(september,27,2013)).
haliday('Eid-al-Fitr',date(october,24,2013)).
haliday('Eid al-Adha',date(december,31,2013)).
```

Structure (Compound terms) in a prolog can be of the following three forms.

Exercise

1. Write the appropriate prolog statement for the following statements.
 - a. Abebe likes aster.
 - b. Kebede is handsome.
 - c. Today is holiday.
 - d. Abebe likes French food.
 - e. Haile G/Silassie was famous runner since from 1996E.C.
 - f. Aster married to Alemayeu.
 - g. Every man is mortal.
 - h. Africa connected with the rest of world through Addis Ababa
2. See the following table and represent the data in prolog to keep the appropriate information students information.

| Student Name | Department | Batch | Sex | Date of Birth |
|----------------|-------------------------|-------|-----|-----------------|
| Alemu Kebede | Computer Science | 4th | M | January 4, 1995 |
| Dawit Taye | Computer Science | 3rd | M | March 2, 1997 |
| Sofonias Alemu | Information system | 4th | M | January 1, 1991 |
| Aster Ezira | Information System | 3rd | F | January 4, 1995 |
| Lulit Sewmehon | Information Technoogy | 4th | F | January 4, 1995 |
| Senait Alemu | Information technologyt | 3rd | F | January 4, 1995 |

3. ^(s) Assume given a set of facts of the form `father(name1,name2)` (`name1` is the father of `name2`).
 - a. Define a predicate `brother(X,Y)` which holds iff `X` and `Y` are brothers.
 - b. Define a predicate `cousin(X,Y)` which holds iff `X` and `Y` are cousins.
 - c. Define a predicate `grandson(X,Y)` which holds iff `X` is a grandson of `Y`.

- d. Define a predicate `descendent(X,Y)` which holds iff X is a descendent of Y.
4. See the following knowledge base interpret what the program does with your own statement.

```
inafrica(nairobi).
inafrica(kampala).
inafrica(dareselam).
inafrica(yawunde).
inafrica(johansberg).
inafrica(asmara).
inafrica(diredawa).
inafrica(mokadisho).
ineurope(london).
ineurope(paris).
ineurope(brussles).
ineurope(jeneva).
ineurope(veina).
ineurope(madrid).
inasia(tokyo).
inasia(hongkong).
inasia(delhi).
inasia(bejing).
inasia(bamkook).
inasia(seol).
inamerica(newyork).
inamerica(dc).
inlatin(swapolo).
inrestoftheworld(X):-ineurope(X);inasia(X);inamerica(X);inlatin(X).
connected(addisababa,with(X,Y)):-inafrica(X),inrestoftheworld(Y).
```


Laboratory Session 3 : Logical Operators and Arithmetic Operators in Prolog

Objectives

Provide a concept how to use prolog operators such as arithmetic, assignment and logical operators in prolog.

Learning Outcome

After the completion of this laboratory session, student can use operators in appropriate place of the program elements

Prolog defines a set of prefix, infix, and postfix operators that includes the classical arithmetic symbols: $+$, $-$, $*$, and $/$. The Prolog interpreter considers operators as functors and transforms expressions into terms.

Thus, $2 * 3 + 4 * 2$ is equivalent to $+(*(2, 3), *(4, 2))$.

The mapping of operators onto terms is governed by rules of priority and classes of associativity:

Arithmetic in Prolog

Prolog is not the programming language of choice for carrying out heavy duty mathematics. It does, however, provide arithmetical capabilities. The pattern for evaluating arithmetic expressions is (where Expression is some arithmetical expression) the general for arithmetic expression in prolog is

X is Expression

or

is(X,expression).

example

writing X is $10+5*6/3.12$ to prolog consol gives you the following result

X = 20.

the above statement can be rewritten in the form of is(X,expression) as follows

is(X,+(10,/(*(5,6),3))).

to find the predicate equivalent for the arithmetic expression you should first know the operator precedence

- Multiplication
- division
- addition
- **subtraction**

for example if you want to convert Y is $2*6/2+12$ start first forming the multiplication predicate then the division predicate the addition and finally the is predicate.

`is(Y, +(/(* (2,6), 2), 12)).`

to find the equivalent expression for a particular arithmetic expression you can use `display(expression).`

`is(X, +(10, /(* (5,6), 3)))`

Typical Arithmetic Operators

+ addition

_subtraction

/division

*** multiplication**

mod modulo

Population Examples

`pop(usa,280).`

`pop(india,1000).`

`pop(china,1200).`

`pop(brazil,130).`

`pop(ethiopia,90).`

`area(usa,3).`

`area(india,1).`

`area(china,4).`

`area(brazil,3).`

`area(ethiopia,1.3).`

`density(X,Y) :-`

`pop(X,P),`

`area(X,A),`

`Y is P/A.`

Typical relational operators.

$X = Y$ X and Y stand for the same number

$X \neq Y$ X and Y stand for different numbers

$X < Y$ X is less than Y

$X > Y$ X is greater than Y

$X \leq Y$ X is less than or equal to Y

$X \geq Y$ X is greater than or equal to Y

Exercises

1. Write a prolog program that can calculate area of a circle by taking the radius of a circle from the user.
 $Area = \pi * r^2$
2. Write a PROLOG program to convert Celsius temperature in to Fahrenheit $F = (C * 9/5 + 32)$
3. Write a PROLOG Program to find the Area and Volume of a cube.
4. Find the Roots of the given equation $\{ ax^2 + bx + c = 0 \}$
5. re write the following expression in the forms of is(X,Expression format).
 - a. X is $12/2 * 14 - 2 + 13$

Laboratory Session 4: Cuts, Negation, and Related Predicates

Objectives

The objective of this session is to introduce students how they:

- Use cut, negation and other predicates in pruning unnecessary output of the program

Outcomes

- enables to deal with AI problems with built in list predicates of Prolog

Cuts

The cut predicate, written `!`, is a device to prune some backtracking alternatives. It modifies the way Prolog explores goals and enables a programmer to control the execution of programs. When executed in the body of a clause, the cut always succeeds and removes backtracking points set before it in the current clause. Figure 4.1 shows the execution model of the rule $p(X) :- q(X), !, r(X)$ that contains a cut.

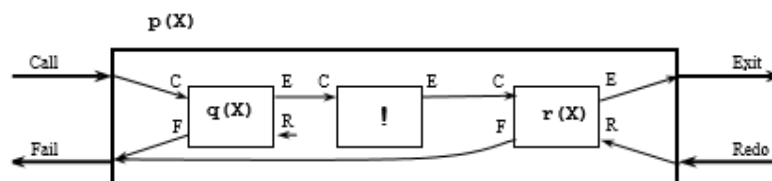


Fig. 4.1 The execution box representing the rule $p(X) :- q(X), !, r(X)$.

Let us suppose that a predicate P consists of three clauses:

```
P :- A1, ..., Ai, !, Ai+1, ..., An.  
  
P :- B1, ..., Bm.  
  
P :- C1, ..., Cp.
```

Executing the cut in the first clause has the following consequences:

1. All other clauses of the predicate below the clause containing the cut are pruned. That is, here the two remaining clauses of P will not be tried.

2. All the goals to the left of the cut are also pruned. That is, A_1, \dots, A_i will no longer be tried.
3. However, it will be possible to backtrack on goals to the right of the cut.

```
P :- A1, ..., Ai, !, Ai+1, ..., An.  
  
P :- B1, ..., Bm.  
  
P :- C1, ..., Cp.
```

Cuts are intended to improve the speed and memory consumption of a program. However, wrongly placed cuts may discard some useful backtracking paths and solutions. Then, they may introduce vicious bugs that are often difficult to track. Therefore, cuts should be used carefully.

An acceptable use of cuts is to express determinism. Deterministic predicates always produce a definite solution; it is not necessary then to maintain backtracking possibilities. A simple example of it is given by the minimum of two numbers:

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

Once the comparisons done, there is no means to backtrack because both clauses are mutually exclusive. This can be expressed by adding two cuts:

```
minimum(X, Y, X) :- X < Y, !.  
  
minimum(X, Y, Y) :- X >= Y, !.
```

Some programmers would rewrite minimum/3 using a single cut:

```
minimum(X, Y, X) :- X < Y, !, minimum(X, Y, Y).
```

The idea behind this is that once Prolog has compared X and Y in the first clause, it is not necessary to compare them again in the second one. Although the latter program may be more efficient in terms of speed, it is obscure. In the first version of minimum/3, cuts respect the logical meaning of the program and do not impair its legibility. Such cuts are called green cuts. The cut in the second minimum/3 predicate is to avoid writing a condition explicitly.

Such cuts are error-prone and are called red cuts. Sometimes red cuts are crucial to a program but when overused, they are a bad programming practice.

Negation

A logic program contains no negative information, only queries that can be proven or not. The Prolog built-in negation corresponds to a query failure: the program cannot prove the query. The negation symbol is written `!` or `not` in older Prolog systems:

- If G succeeds then `!G` fails.
- If G fails then `!G` succeeds. The Prolog negation is defined using a cut:

```
!(P) :- P, !, fail. !(P) :- true.
```

where `fail` is a built-in predicate that always fails. Most of the time, it is preferable to ensure that a negated goal is ground: all its variables are instantiated. Let us illustrate it with the somewhat odd rule:

```
mother(X, Y) :- !male(X), child(Y, X).
```

and facts:

```
child(telemachus, penelope).  
  
male(ulysses).  
  
male(telemachus).
```

The query

```
?- mother(X, Y).
```

fails because the subgoal `male(X)` is not ground and unifies with the fact `male(ulysses)`. If the subgoals are inverted:

```
mother(X, Y) :- child(Y, X), !male(X).
```

the term `child(Y, X)` unifies with the substitution `X = penelope` and `Y = telemachus`, and since `male(penelope)` is not in the database, the goal `mother(X, Y)` succeeds. Predicates similar to `!` include `if-then` and `if-then-else` constructs. `if-then` is expressed by the built-in `'->'` operator. Its syntax is

Condition -> Action

as in

```
print_if_parent(X, Y) :- (parent(X, Y) -> write(X), nl,
write(Y), nl).
```

```
?- print_if_parent(X, Y). penelope telemachus
X = penelope, Y = telemachus
```

Just like negation, $\phi \rightarrow \phi^2$ is defined using a cut:

$\phi \rightarrow \phi(P, Q) :- P, !, Q.$

The if-then-else predicate is an extension of $\phi \rightarrow \phi^2$ with a second member to the right. Its syntax is

```
Condition -> Then ; Else
```

If Condition succeeds, Then is executed, otherwise Else is executed.

The once/1 Predicate

The built-in predicate once/1 also controls Prolog execution. once(P) executes P once and removes backtrack points from it. If P is a conjunction of goals as in the rule:

```
A :- B1, B2, once((B3, ..., Bi)), Bi+1, ..., Bn.
```

the backtracking path goes directly from B_{i+1} to B_2 , skipping B_3, \dots, B_i . It is necessary to bracket the conjunction inside once twice because its arity is equal to one. A single level of brackets, as in `once(B3, ..., Bi)`, would tell Prolog that once/1 has an arity of $i-3$. once(Goal) is defined as:

`once(Goal) :- Goal, !`

Laboratory Session 5: Lists and Recursion

Objectives

The objective of this session is to introduce students how they:

- Deal with list data structure of prolog
- Define their own rule to access and manipulate list structure

Outcomes

- Enables to deal with lists access and manipulation using prolog.

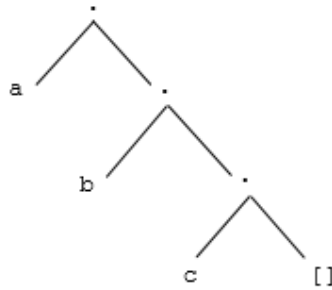
Lists are data structures essential to many programs. A Prolog list is a sequence of an arbitrary number of terms separated by commas and enclosed within square brackets. For example:

- `[a]` is a list made of an atom.
- `[a, b]` is a list made of two atoms.
- `[a, X, father(X, kebede)]` is a list made of an atom, a variable, and a compound term.
- `[[a, b], [[[father(X, kebede)]]]]` is a list made of two sub- lists.
- `[]` is the atom representing the empty list.

Although it is not obvious from these examples, Prolog lists are compound terms and the square bracketed notation is only a shortcut. The list functor is a dot: `./2`, and `[a, b]` is equivalent to the term `.(a,.(b,[]))`. Computationally, lists are recursive structures. They consist of two parts:

- a head, the first element of a list,
- and a tail, the remaining list without its first element.

The head and the tail correspond to the first and second argument of the Prolog list functor. Figure 5.1 shows the term structure of the list `[a, b, c]`. The tail of a list is possibly empty as in `.(c,[])`.



The term structure of the list [a, b, c].

The notation $H|T$ splits a list into its head and tail, and $[H | T]$ is equivalent to $.(H, T)$. Splitting a list enables us to access any element of it and therefore it is a very frequent operation. Here are some examples of its use:

```

?- [a, b] = [H | T]. H = a, T = [b]

?- [a] = [H | T]. H = a, T = []

?- [a, [b]] = [H | T]. H = a, T = [[b]]

?- [a, b, c, d] = [X, Y | T]. X = a, Y = b, T = [c, d]

?- [[a, b, c], d, e] = [H | T]. H = [a, b, c], T = [d, e]
  
```

The empty list cannot be split:

```

?- [] = [H | T]. No
  
```

using the head and tail structure of the lists data structure a programmer can create his own rule that can access and manipulate different elements of the lists.

Example

the following codes can define a rule that can access(display)the first element any arbitrary list with a query form `first([Listelements],First)`.

```
first(X,[First|_]):-X=First.
```

on the same structure a programmer may define a rule that checks weather some elements are a member of a list or not and displays an element of arbitrarily list definition.

Example member.pl

```
member(X, [X | _]).
```

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

the above example shows a typical usage of recursion structure in list operation.

Recursion

Predicates can be defined recursively. Roughly speaking, a predicate is recursively defined if one or more rules in its definition refers to itself. in recursion program there are two basic statement. the first statement is called a termination case in which it uses to terminate the recursion when some condition satisfied. the second and the main program statement is a program that defines the predicate programs and calls the predicates inside its body.

in the above example of member.pl program the first line or statement is a termination statement which stops searching further if it finds the same value for variable X in the list head part.

```
1.....member(X, [X | _]).
```

the second statement is the one that recursively find the equivalent values of variable X in the tail part of the list, by traversing each element of the list starting from the head part to the last elements. in this as far as the last elements the list is an empty set, it signals searching should be terminated.

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

Exercises

1. write the outputs of following statement in the prolog compiler interface .
 - a. [a,b,c,d] = [a,[b,c,d]].
 - b. X=[a,b,c,d]
 - c. [a,b,c,d] = [a|[b,c,d]].
 - d. [a,b,c,d] = [a,b,[c,d]].
 - e. [a,b,c,d] = [a,b|[c,d]].
 - f. [a,b,c,d] = [a,b,c,[d]].
 - g. [a,b,c,d] = [a,b,c|[d]].
 - h. [a,b,c,d] = [a,b,c,d,[]].
 - i. [a,b,c,d] = [a,b,c,d|[]].
 - j. [] = _.

k. $[] = [_]$.

l. $[] = [_|[]]$.

m. $[H|T]=[1,2,3,4,5,6]$

2. **write a predicates that finds the second, the fifth and the last element of the any arbitrary lists.**
3. Write a predicate `twice(In,Out)` whose left argument is a list, and whose right argument is a list consisting of every element in the left list written twice. For example, the query

`twice([a,4,buggle],X).` should return $X = [a,a,4,4,buggle,buggle]$ and the query
`twice([1,2,1,1],X).` should return $X=[1,1,2,2,1,1,1,1]$

Laboratory session 6: Some List-Handling Predicates

Objectives

The objective of this session is to introduce students how they:

- Uses different built in predicates of list

Outcomes

- enables to deal with AI problems with built in list predicates of Prolog

Many applications require extensive list processing. This section describes some useful predicates. Generally, Prolog systems provide a set of built-in list predicates. Consult your manual to see which ones; there is no use in reinventing the wheel.

The member/2 Predicate

The member/2 predicate checks whether an element is a member of a list:

```
?- member(a, [b, c, a]).  
Yes  
  
?- member(a, [c, d]).  
No
```

member/2 is defined as:

```
member(X, [X | _]). % Termination case  
member(X, [_ | YS]) :- % Recursive case  
    member(X, YS).
```

We could also use anonymous variables to improve legibility and rewrite member/2 as

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

member/2 can be queried with variables to generate elements member of a list, as in:

```
?- member(X, [a, b, c]).  
  
X = a ;  
  
X = b ;  
  
X = c ;  
  
False
```

Or lists containing an element:

```
?- member(a, Z).  
  
Z = [a | Y] ; Z = [Y, a | X] ; etc.
```

Finally, the query:

```
?- \+ member(X, L).  
  
true
```

where X and L are ground variables, returns Yes if member(X, L) fails and No if it succeeds.

The append/3 Predicate

The append/3 predicate appends two lists and unifies the result to a third argument:

```
?- append([a, b, c], [d, e, f], [a, b, c, d, e, f]). Yes  
  
?- append([a, b], [c, d], [e, f]). No  
  
?- append([a, b], [c, d], L). L = [a, b, c, d]  
  
?- append(L, [c, d], [a, b, c, d]). L = [a, b]  
  
?- append(L1, L2, [a, b, c]). L1 = [], L2 = [a, b, c] ; L1 = [a], L2 = [b, c] ;  
etc., with all the combinations.
```

etc., with all the combinations.

append/3 is defined as

```
append([], L, L).
```

```
append([X | XS], YS, [X | ZS]) :- append(XS, YS, ZS).
```

The delete/3 Predicate

The delete/3 predicate deletes a given element from a list. Its synopsis is:

`delete(List, Element, ListWithoutElement).`

example queries

```
?- delete([2,3,4,5,6],6,X).  
  
X = [2, 3, 4, 5].
```

It is defined as:

```
delete([], _, []).  
  
delete([E | List], E, ListWithoutE):- !, delete(List, E,  
ListWithoutE).  
  
delete([H | List], E, [H | ListWithoutE]):- H \= E, !,  
delete(List, E, ListWithoutE).
```

The three clauses are mutually exclusive, and the cuts make it possible to omit the condition $H \neq E$ in the second rule. This improves the program efficiency but makes it less legible.

The intersection/3 Predicate

The intersection/3 predicate computes the intersection of two sets represented as lists:

`intersection(InputSet1, InputSet2, Intersection).`

```
?- intersection([a, b, c], [d, b, e, a], L).  
  
L = [a, b]
```

InputSet1 and InputSet2 should be without duplicates; otherwise intersection/3 approximates the intersection set relatively to the first argument:

```
?- intersection([a, b, c, a], [d, b, e, a], L).  
  
L = [a, b, a]
```

The predicate is defined as:

```
% Termination case
intersection([], _, []).

% Head of L1 is in L2
intersection([X | L1], L2, [X | L3]) :- member(X, L2),
!, intersection(L1, L2, L3). % Head of L1 is not in L2
intersection([X | L1], L2, L3) :- \+ member(X, L2), !,
intersection(L1, L2, L3).
```

As for delete/3, clauses of intersection/3 are mutually exclusive, and the programmer can omit the condition `\+ member(X, L2)` in the third clause.

The reverse/2 Predicate

The reverse/2 predicate reverses the elements of a list. There are two classic ways to define it. The first definition is straight forward but consumes much memory. It is often called the naïve reverse:

```
reverse([], []). reverse([X | XS], YS) :- reverse(XS, ,
RXS), append(RX, [X], Y).
```

A second solution improves the memory consumption. It uses a third argument as an accumulator.

```
reverse(X, Y) :- reverse(X, [], Y).

reverse([], YS, YS). reverse([X | XS], Accu, YS) :- reverse(XS,
[X | Accu], YS).
```

Exercises

1. Write the output of the following Prolog Statements.

- a. `append([1,2,3,4,5],[1,2,3,4,5],X).`
- b. `append([1,2,3,4,5], X).`
- c. `append([1,2,3,4,5],[1,2,3,[],45],X).`

- d. `member([1,2,3,45],X).`
- e. `member(X, [1,2,3,45],).`
- f. `delete([1,2,3,4,5],6,X).`
- g. `append([1,2,3,4,5],[1,2,3,45],X).`
- h. `intersection([a, b, c], [d, b, e, a]).`
- i. `intersection([a, b, c], [d, b, e, a], [a,b],X).`

Laboratory Session 7: Input and Output in Prolog

Objectives of the Session

The objective of this session is to introduce students how they:

- create a user interface for their program
- create an interactive program using prologs

Outcomes

- enables to write an interactive programs

write() : Writes a single term to the terminal. write(term) is true if term is a Prolog term.

For example: write(a), or write(-How are you?ø)

write_ln() Writes a term to the terminal followed by a new line.

tab(X) Writes an X number of spaces to the terminal.

read(X) Reads a term from the keyboard and instantiates variable X to the value of the read term.

_ This term to be read has to be followed by a dot . and a white space character (such as an enter or space).

_ For example:

```
hello :-  
    write('What is your name ?'),  
    read(X),  
    write('Hello'), tab(1), write(X).
```

You can write more than one horn clause in prolog file

```
sum :-  
    write('Enter the first Number ?'),  
    read(X),  
    write('Enter the Second Number'),  
    read(Y),  
    Z is X+Y,  
    write('The sum of the two number is'), write(Z).  
subtract:-  
    write('Enter the first Number ?'),
```

```

        read(X),
        write('Enter the Second Number'),
        read(Y),
        Z is X-Y,
        write('The first number minus the second number is'),
write(Z).
divide :-
        write('Enter the first Number ?'),
        read(X),
        write('Enter the Second Number'),
        read(Y),
        Z is X/Y,
        write('The fiirst number divided for second number is'),
write(Z).
product:-
        write('Enter the first Number ?'),
        read(X),
        write('Enter the Second Number'),
        read(Y),
        Z is X*Y,
        write('The product of the two number is'), write(Z).
modules:-
        write('Enter the first Number ?'),
        read(X),
        write('Enter the Second Number'),
        read(Y),
        Z is mod(X,Y),
        write('The modulo of first number to the second is'),
write(Z).

```

Exercises

1. Create a program that request users a mark and calculate the average.
2. Read about assert and retract commands and use them to automate progrms

Laboratory Session 8: Fact representation for AI Problem; directed graph, and defining successor function

Objectives of the Session

The objective of this session is to introduce students how they:

- can represent AI problems particularly paths problems in prolog

Outcomes

- can implement represent problems(toy and real) in prolog

The most concise and easy way to remember illustration of AI searches is using the road map model that connects different locations shown in figure 6.1

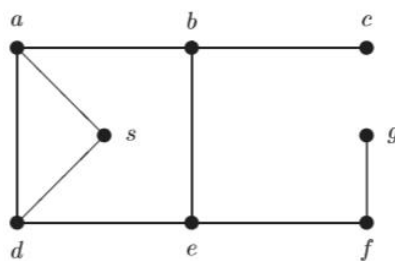


figure 6.1 a simple road map model connects

As there are different types of searching techniques in AI, in this laboratory session we will see implementation of one type of blind-searching techniques called breadth first Search for the road map model that has described in figure 6.1.

For the convenience of understanding the codes clearly, We alienated the facts that describes the road maps facts from the actual depth first implementation. The fact description implementation module created and saved by the name fact.pl. Later we will call this module in the search algorithm bodies by its file name.

```
:- module(edges, [link/2]).  
connect(a,b).  
connect(a,d).  
connect(a,s).
```

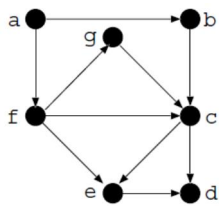
```

connect(b,c) .
connect(b,e) .
connect(d,e) .
connect(d,s) .
connect(e,f) .
connect(f,g) .
link(Node1,Node2) :- connect(Node1,Node2) .
link(Node1,Node2) :- connect(Node2,Node1) .

```

Exercises

1. Represent the following directed graph in prolog predicate form



2. assess all the possible roads from Bahir Dar University Poly campus and Peda Campus and represent in a prolog files

Laboratory Session 9 and 10: Implementation of Breadth First and Depth first search goal finding.

Objectives of the Session

The objective of this session is to introduce students how they:

- implementing depth and breadth first search for goal

Outcomes

- can implement depth and breadth searches for any AI Problem

the important topics here in this laboratory session is to show students the implementation of breadth search algorithm.

As we have remember from our lecture discussion breadth-first search characterized by expanding the shallowest node of the tree for finding the goal of the problem iteratively until it comes with the solution if there is any.

Important functions and lists to implement this searching techniques described in the following table .

| | | |
|---|--------------------|--|
| 1 | Goal-test function | That check whether the nodes to be expanded is a goal node or not before given |
| 2 | Successor function | A function that generates the child nodes from the current nodes if the current node is not a goal node. |
| 3 | Merger function | A function that checks weather the newly generated nodes are not a member of explored nodes(a nodes that have been already tested with goal test function and their child nodes are generated usually kept as closed list) and merge with a node that has kept in an open list that are waiting for further to be checked by goal-test function and by successor function. Newopenlist={listof nodes generated by successor function} n {nodes lists that is not in closed |

| | | |
|---|-------------|--|
| | | list} plus previous open list compliment current node. |
| 4 | Open list | Lists that contain the set of nodes that has been generated by successor-function through each and further weighting for checking by goal test function and successor-function. always updated by merger-function. <u>Note Remember</u> Breadth first search use FIFO data structure for accessing lists and implemented in queue data structure. |
| 5 | Closed List | Lists that contain a set of nodes that have already cheked with goal-test function and were not a goal. |

```

BreadthFirst(StartNode,GoalNode)
comment: Depth First Search with a List of Closed Nodes
RootNode ← StartNode
OpenList← [RootNode]
ClosedList← []
[H|T] ← OpenList while [H = GoalNode]
do
{
    SuccList← successors of H .....(1)
    OpenList← (SuccList∩ClosedList copmliment) .....(2)
    ClosedList← [H|ClosedList]..... (3)
    if OpenList = [] then return (failure)
    [H|T] ← OpenListreturn (success)
}

```

The following figure shows the general algorithm for breadth first search

```

:-use_module(fact).
breadth_first(Start,Goal):-write('Open: '),
write([Start]), write(', Closed: '), write([]), nl,
loop([Start],[],Goal).

```

```

loop([Goal|_],_,Goal):- write('The Goal is : '),
write(Goal).
loop([CurrNode|OtherNodes],ClosedList,Goal):-
successors(CurrNode,SuccNodes),
    write('Node '), write(CurrNode),
    write(' is being expanded.'),
    findall(Node,(member(Node,SuccNodes),
        not(member(Node,ClosedList)))),Nodes),
    append(OtherNodes,Nodes,NewOpenNodes),
    write('Sucessor nodes: '), write(SuccNodes),nl,
    write('Open Nodes: '), write(NewOpenNodes),
    write(',          Closed Node : '),
write([CurrNode|ClosedList]),nl,
    loop(NewOpenNodes,[CurrNode|ClosedList],Goal).
successors(Node,SuccNodes):-
findall(Successor,link(Node,Successor),SuccNodes).

```

```

6 ?- depth_first(s,g).
Open: [s], Closed: []
Node s is being expanded.Sucessor nodes: [a,d]
Open Nodes: [a,d], Closed Node : [s]
Node a is being expanded.Sucessor nodes: [b,d,s]
Open Nodes: [b,d,d], Closed Node : [a,s]
Node b is being expanded.Sucessor nodes: [c,e,a]
Open Nodes: [c,e,d,d], Closed Node : [b,a,s]
Node c is being expanded.Sucessor nodes: [b]
Open Nodes: [e,d,d], Closed Node : [c,b,a,s]
Node e is being expanded.Sucessor nodes: [f,b,d]
Open Nodes: [f,d,d,d], Closed Node : [e,c,b,a,s]
Node f is being expanded.Sucessor nodes: [g,e]
Open Nodes: [g,d,d,d], Closed Node : [f,e,c,b,a,s]
The Goal is : g
true

```

Exercises

1. Implement depth first search and iterative deepening search for similar problem of goal finding.

Laboratory Session 11 and 12: Implementation of Breadth First Search and Depth first search with Path finder.

Objectives of the Session

The objective of this session is to introduce students how they:

- Implement a path finder in blind search algorithms.

Outcomes

- it enables students to implementing blind searches for any problem.
1. Extend the above Implementation of BFS to be more interactive and finds the paths instead of the nodes alone.

for this laboratory session we implement the depth first search which finds paths for a particular path finding problem. What makes different this implementation from previous topic is that

- its first argument will take the list of open paths (and not that of open nodes). This is the argument where we accumulate (maintain) the agenda.
- Into an additional (fourth) argument will the path from Start to Goal be copied as soon as it appears at the head of the agenda. The search is then finished.
- The second and third arguments of will hold, as before, the list of closed nodes and the goal node, respectively.

The following is the algorithm for path finding problem using depth first search

comment: Depth First with Closed Nodes and Open Paths

procedure EXTENDPATH($[x_1, \dots, x_N], list$)

comment: To return $[]$ if the first argument is $[]$

for $i \leftarrow 1$ **to** N

do $\{list_i \leftarrow [x_i|list]$

return $([list_1, \dots, list_N])$

main

$RootNode \leftarrow StartNode$

$OpenPaths \leftarrow [[RootNode]]$

$ClosedNodes \leftarrow []$

$[[H|T]|TailOpenPaths] \leftarrow OpenPaths$

while $H \neq GoalNode$

do $\left\{ \begin{array}{l} SuccList \leftarrow \text{successors of } H \\ NewOpenNodes \leftarrow (SuccList \cap ClosedList^c) \\ NewPaths \leftarrow \text{EXTENDPATH}(NewOpenNodes, [H|T]) \\ OpenPaths \leftarrow NewPaths \uparrow TailOpenPaths \\ \text{if } OpenPaths = [] \\ \quad \text{then return (failure)} \\ [[H|T]|TailOpenPaths] \leftarrow OpenPaths \end{array} \right.$

$Path \leftarrow \text{REVERSE}([H|T])^5$

output $(Path)$

connect(a,b) .

connect(a,d) .

connect(a,s) .

connect(b,c) .

connect(b,e) .

connect(d,e) .

connect(d,s) .

connect(e,f) .

connect(f,g) .

link(Node1,Node2) :- connect(Node1,Node2) .

link(Node1,Node2) :- connect(Node2,Node1) .

```

depth_first(Start,Goal,PathFound) :-
dfs_loop([[Start]],[],Goal,PathFoundRev),
reverse(PathFoundRev,PathFound).

dfs_loop([[Goal|PathTail]|_],_,Goal,[Goal|PathTail]).

dfs_loop([[CurrNode|T]|Others],ClosedList,Goal,PathFound) :-
successors(CurrNode,SuccNodes),

    findall(Node,(member(Node,SuccNodes),
not(member(Node,ClosedList))),Nodes),

    extend_path(Nodes,[CurrNode|T],Paths),
append(Others,Paths,NewOpenPaths),

dfs_loop(NewOpenPaths,[CurrNode|ClosedList],Goal,PathFound).

successors(Node,SuccNodes) :-
findall(Successor,link(Node,Successor),SuccNodes).

extend_path([],_,[]).

extend_path([Node|Nodes],Path,[Node|Path]|Extended) :-

    extend_path(Nodes,Path,Extended).

```

```

4 ?- depth_first(s,g,PathFound).
PathFound = [s, d, e, f, g] ;
PathFound = [s, a, b, e, f, g] ;
PathFound = [s, a, d, e, f, g] ;
_

```

Exercises

1. Implement the path finder solution for breadth and iterative deepening search

Laboratory 13: Heuristic Functions Implementation

Objectives of the Session

The objective of this session is to introduce students how they:

- Represent cost information for edges
- Calculate a heuristic for both straight line and Manhattan distance.

Outcomes

- it enables students to use heuristic and edge information for implementing searches for a problem search as robot navigation and minimum path findings

Edge cost representation

In case of the edge cost representation the programmer first explicitly describe the edge distance of each of the nodes.

Example: we have an estimated edge cost for any two nodes then we can represent as

```
e_cost(a,b,10).  
e_cost(a,d,12).  
e_cost(a,s,10).  
e_cost(b,c,8).  
e_cost(b,e,7).  
e_cost(d,e,12).  
e_cost(d,s,14).  
e_cost(e,f,9).  
e_cost(f,g,8).
```

Then we can connect the costs with the previous link file using the following codes.

```
:-use_module(fact).  
edge_cost(Node1,Node2,Cost) :- link(Node1,Node2), e_cost(Node1,Node2,Cost);  
e_cost(Node2,Node1,Cost).
```

Heuristic or estimated Costs.

There are two common types of heuristic information in AI problems these are straight line distance and Manhattan distance costs. In case of Straight line distance the heuristic calculate the distance between two points which have X and Y coordinates.

The coordinate point can be represented in the following codes

```
in(s,2,4).  
in(a,3,7).  
in(b,5,6).  
in(c,9,4).  
in(d,2,9).  
in(e,5,8).  
in(f,6,10).  
in(g,5,7).
```

Then we can calculate the heuristic in the following information

```
herustic(Node,Goal,D) :- in(Node,X1,Y1), in(Goal,X2,Y2), D is sqrt((X1 - X2)^2 +  
(Y1 - Y2)^2).
```

For Manhattan distance you can find the heuristic by counting the deviation of the object from the goal state using the following code implementation.

```
manhaton(Node1,Node2,Cost) :- link(Node1,Node2), in(Node1,X1,Y1),  
in(Node2,X2,Y2), Cost is abs(X1 - X2) + abs(Y1 - Y2).
```

Exercises

- A. Write a code that find the total edge cost for a nodes from the start point to current point.

- B. Write a code that calculate the $G(X)$ costs assume $G(X)=f(X)+h(X)$. $h(x)$ is a herustic function for some nodes and $f(x)$ is the total edge cost an object consumed from the start to the current object location.
- C. Implement A^* search in the previous codes using the above information and show the minimum distance for a navigation.