Camelia LEMNARU
Rodica POTOLEA

LOGIC PROGRAMMING

A Hands-on Approach

U.T. PRESS
CLUJ-NAPOCA, 2018
ISBN 978-606-737-292-2

Editura U.T. PRESS
Str. Observatorului nr. 34
C.P. 42, O.P. 2, 400775 Cluj-Napoca
Tel.:0264-401.999
e-mail: utpress@biblio.utcluj.ro
http://biblioteca.utcluj.ro/editura

Director: Ing. Călin D. Câmpean

Recenzia: Conf.dr.ing. Tudor Mureşan
Ş.l.dr.ing. Ciprian Oprişa

Copyright © 2018 Editura U.T.PRESS
Reproducerea integrală sau parţială a textului sau ilustraţiilor din această carte este posibilă numai cu acordul prealabil scris al editurii U.T.PRESS.

ISBN 978-606-737-292-2

Table of Contents

0. Introduction to Prolog. Unification. Sicstus Prolog 3

1. Facts and rules. Family tree 6

3. Lists. List Operations (I) 14

4. The Cut (!). List Operations - Forward and Backward Recursion (II) 22

5. Sorting Methods 29

6. Deep lists 35

7. Trees. Operations on trees 39

8. Incomplete structures - lists and trees 45

9. Difference Lists. Side Effects 50

10. Graphs. Paths in Graphs 58

11. Graphs Search Algorithms 63

Bibliography 66

0. Introduction to Prolog. Unification. Sicstus Prolog

In this introductory session you will get acquainted with the Prolog data types and unification rules, and

0.1 Prolog Data Types

Prolog's single data type is the *term*. Terms are either: *atoms*, *numbers*, *variables* or *compound terms (structures).*

Atoms and numbers

An atom is a general-purpose name with no inherent meaning. It is composed of a sequence of characters that is parsed by the Prolog reader as a single unit. Atoms are usually bare words in Prolog code, written with no special syntax. However, atoms containing spaces or certain other special characters must be surrounded by single quotes. Atoms beginning with a capital letter must also be quoted, to distinguish them from variables. The empty list, written [], is also an atom. Other examples of atoms include:

x, blue, 'Taco', and 'some atom'.

Numbers can be integers or real numbers (not used very much in typical Prolog programming).

Variables

Variables are denoted by a string consisting of letters, numbers and underscore characters, and beginning with an upper-case letter or underscore. Variables closely resemble variables in logic in that they are placeholders for arbitrary terms. A variable can become instantiated (bound to equal a specific term) via unification. A single underscore (_) denotes an anonymous variable and means "any term". Unlike other variables, the underscore does not represent the same value everywhere it occurs within a predicate definition.

Compound Terms

A compound term is composed of an atom called a "functor" and a number of "arguments", which are again terms. Compound terms are ordinarily written as a functor followed by a comma-separated list of argument terms, which is contained in parentheses. The number of arguments is called the term's arity. An atom can be regarded as a compound term with arity zero.

Examples of compound terms are: 'Person_Friends'(zelda,[tom,jim]) and truck_year('Mazda', 1986). Users can declare arbitrary functors as operators with different precedence to allow for domain-specific notations. The notation f/n is commonly used to denote

a term with functor f and arity n.

Special cases of compound terms:

- Lists are defined inductively. The list [1, 2, 3] would be represented internally as '.'(1, '.'(2, '.'(3, []))). A syntactic shortcut is [H | T], which is mostly used to construct rules. A list can be processed by processing the first element, and then the rest of the list, in a recursive manner.

Lists can be constructed and deconstructed in a variety of ways:

  - Element enumeration: [abc, 1, f(x), Y, g(A,rst)]
  - Prepending single element: [abc | L1]
  - Prepending multiple elements: [abc, 1, f(x) | L2]
  - Term expansion: '.'(abc, '.'(1, '.'(f(x), '.'(Y, '.'(g(A,rst), [])))))

- Strings

0.2 Prolog Unification Rules

- **Constants** unify with themselves only:
  1. **Atoms** unify if and only if they are the same atom.
  2. **Numbers** unify if and only if they are the same number.
  3. **Strings** unify if and only if they are the same string.
  4. **Lists** unify if and only if
     - their heads unify, and
     - their tails unify.
  5. **Structures** unify if and only if
     - their names unify,
     - they have the same number of arguments, and
     - their arguments unify.
- **Variable** *V* unifies with **Term** *T* just in case one of the following conditions is satisfied:
  - *V* is an instantiated variable.
    - If *T* is not a variable, then *V* and *T* unify if and only if the term instantiated on *V* unifies with *T*.
    - If *T* is an instantiated variable, then *V* and *T* unify if and only if the term instantiated on *V* unifies with the term instantiated on *T*.
    - If *T* is an uninstantiated variable, then *V* and *T* are unified by instantiating on *T* the term that is instantiated on *V*.
  - *V* is an uninstantiated variable.
    - If *T* is not a variable, then *V* and *T* are unified by instantiating *T* on *V*.
    - If *T* is an instantiated variable, then *V* and *T* are unified by instantiating on *V* the term that is instantiated on *T*.
    - If *T* is an uninstantiated variable, then *V* and *T* unify and become synonyms for the same variable.

Exercise 0.1: Check Sicstus Prolog manual for:

(1) Terms (4.1.2)
(2) Compound Terms (4.1.3)
(3) Unification (4.8.1.2)

0.3 Quiz exercises

0.3.1 Which is the nature of the following Prolog terms:

a. X b. 'X' c. _138	• hello • Hello • 'Hello'	• [a, b, c] • [A, B, C] • [Ana, are, 'mere']

0.3.2 Look up the following built-in predicates in the Sicstus Prolog manual: var(Term), nonvar(Term), number(Term), atom(Term), atomic(Term) – section 4.8.1.1 *Type Checking*. Test the validity of your answers from exercise 3, by using them.

0.3.3 Execute the following unification queries. Explain the results in a text file:

a. ?- a = a.
b. ?- a = b.
c. ?- 1 = 2.
d. ?- 'ana' = 'Ana'.
e. ?- X = 1, Y = X.
f. ?- X = 3, Y = 2, X = Y.
g. ?- X = 3, X = Y, Y = 2.
h. ?- X = ana.
i. ?- X = ana, Y = 'ana', X = Y.
j. ?- a(b,c) = a(X,Y).
k. ?- a(X,c(d,X)) = a(2,c(d,Y)).
l. ?- a(X,Y) = a(b(c,Y),Z).
m. ?- tree(left, root, Right) = tree(left, root, tree(a, b, tree(c, d, e))).
n. ?- k(s(g),t(k)) = k(X,t(Y)).
o. ?- father(X) = X.
p. ?- loves(X,X) = loves(marsellus,mia).
q. ?- [1, 2, 3] = [a, b, c].
r. ?- [1, 2, 3] = [A, B, C].
s. ?- [abc, 1, f(x) | L2] = [abc|T].
t. ?- [abc, 1, f(x) | L2] = [abc, 1, f(x)].

1. Facts and rules. Family tree

In this session you will learn how to write a Prolog program, and how to call Prolog predicates. We will start by a simple example – the genealogy tree. We will define a series of relationships – *father, mother, sibling, brother, sister, aunt, uncle, grandmother, grandfather* and *ancestor*. Let's begin with a (partial) depiction of such a tree:

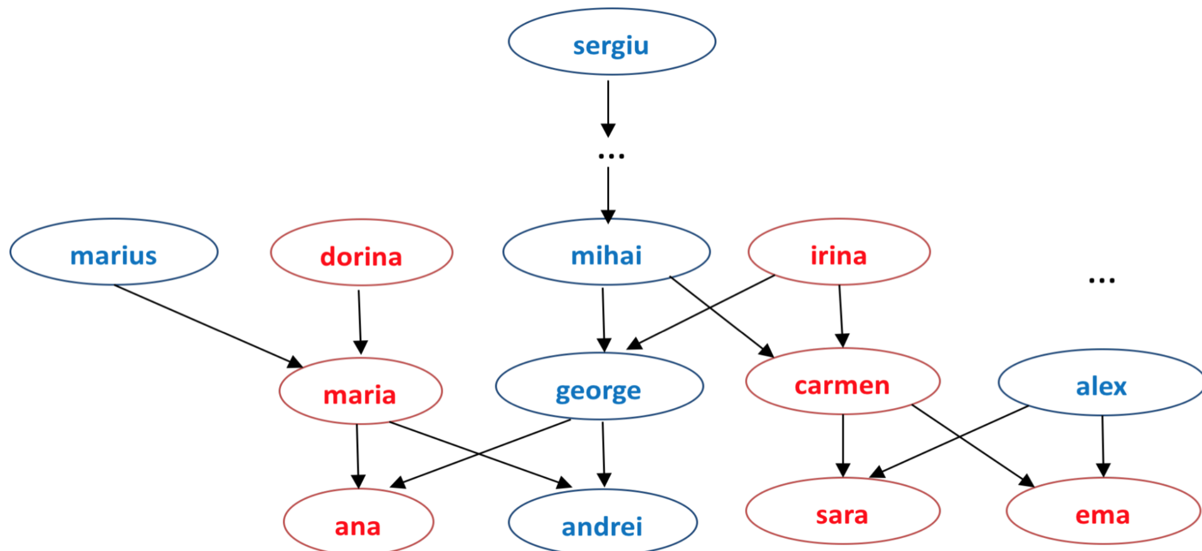


Figure 1.1 – Genealogy tree example

We will build our predicate base incrementally. So, let's start by declaring a number of facts – i.e. ground truths.

Hint: Predicate specifications are written in a source file (a simple text file). You may save it with the extension .pl (recognized by Sicstus Prolog and other Prolog engines, such as SWI Prolog – a free version of Prolog). A line comment in Sicstus starts with %. Block comments are enclosed between / … */.*

```
woman(ana). % Remember, predicate names are constant (start with lowercase letter)
woman(sara).
woman(ema).
woman(maria).
% etc…
man(andrei).
man(george).
man(alex).
%etc…
parent(maria, ana). % maria is ana's parent
parent(george,ana). % george also is ana's parent
parent(maria,andrei).
parent(george,andrei).
```

Therefore, we have defined three predicates: man/1, woman/1 and parent/2, each

consisting of a series of facts. Predicate parent(X,Y) is to be interpreted as *X is the parent of Y*.

Now let's define a predicate for the *mother* relationship. Of course, we need two arguments – i.e. mother/2 – and we will employ the parent/2 and woman/1 predicates:

```
mother(X,Y):-woman(X), parent(X,Y).
% X is Y's mother, it X is a woman and X is the parent of Y
```

Exercise 1.1: Define predicate father/2.

Let us now call the predicates. To let Sicstus know about your defined predicates, you have to use the consult/1 built-in predicate – which you can also access directly from the *File-> Consult* menu. In the file dialog, select your source file. If all goes well (i.e. not syntax errors in your source file), you should see something like this:

```
| ?- :-
consult('E:/Scohol/Catedra/PL/Problems/genealogy.pl').
% consulting e:/scohol/catedra/pl/problems/genealogy.pl...
% consulted e:/scohol/catedra/pl/problems/genealogy.pl in module user, 0 msec 1248 bytes
| ?-
```

Calling a predicate in Prolog is known as "*asking a question*". Below you may find a listing of Prolog queries, with the answers provided by the engine, on the predicates defined so far (do not copy-paste this in the Sicstus console, it will not work):

```
| ?- man(george). % is george a man?
yes
| ?- man(X).  % who is a man?
X = andrei ? ;  % use ; or n to repeat the question and ask for another answer
X = george ? ;
X = alex ? ;
no
| ?- parent(X,andrei). % who are andrei's parents?
X = maria ? ;
X = george ? ;
no
| ?- parent(maria,X). % who are maria's children?
X = ana ? ;
X = andrei ? ;
no
| ?- mother(ana,X). % who are ana's children?
no
| ?- mother(X, ana). % who is ana's mother?
X = maria ? ; % repeat the question, i.e. does ana have another mother besides maria?
no
```

Ok, now try the above queries on your own in Sicstus.

Exercise 1.2: Complete the predicates man/1, woman/1 and parent/2, to have the entire genealogy tree in *Fig. 1.1* covered.

Exercise 1.3: Re-consult your source file in Sicstus, and execute the following queries:

a. ?- father(alex, X).
b. ?- father(X,Y).
c. ?- mother(dorina, maria).

Let us now extend the predicate base with several other predicates:

```
% sibling/2: X and Y are siblings if they have a common parent, and they are different
sibling(X,Y):-parent(Z,X), parent(Z,Y), X\=Y.
% sister/2: X is Y's sister if X is a woman and X and Y are siblings
sister(X,Y):-sibling(X,Y), woman(X).
% aunt/2: X is Y's aunt if she is the sister of Z, who is a parent for Y.
aunt(X,Y):-sister(X,Z),parent(Z,Y).
```

Exercise 1.4: Extend the predicate base with predicates brother/2, uncle/2, grandmother/2 and grandfather/2.

Exercise 1.5: Re-consult your source file in Sicstus, and trace the execution of the following queries (by repeating the question):

a. ?- aunt(carmen, X).
b. ?- grandmother(dorina,X).
c. ?- grandfather(X,ana).

Hint: to activate the trace option, simply type ?-trace. *in Sicstus' query prompt.You will be able to follow the execution of your queries call by call. To deactivate it, use* ?-notrace.

Last, let us focus on writing a predicate ancestor/2: X is the ancestor of Y if it if linked to Y via a certain number of parent relations. In Fig. 1.1, *sergiu* is an ancestor of *mihai*, *sergiu*, *andrei*, *carmen* and *sara.*

Exercise 1.6: Write the ancestor/2 predicate, and execute several queries on it to test its correctness.

2. Simple arithmetic. Recursion

Greatest Common Divisor (GCD)

Let us write a predicate which computes the *greatest common divisor* of two natural numbers. We will apply *Euclid's algorithm*, for which you have the pseudocode below:

gcd(a,a) = a
gcd(a,b) = gcd(a – b, b), if b<a
gcd(a,b) = gcd(a, b – a), if a<b

The above algorithm is a mathematical recurrence, which means that we will need to write a recursive predicate. Since a Prolog predicate does not return a value other than *yes/no (T/F)*, we need to add the result to the predicate parameter list. Therefore, our predicate will be gcd/3, or gcd(X,Y,Z), where X and Y are the two natural numbers, and Z is their *gcd*. The first clause is a fact, stating that the *gcd* of two equal numbers is their value:

```
gcd(X,X,X). % clause 1
```

One more thing you need to know before writing clauses 2 and 3 is that, in Prolog, mathematical expressions are not evaluated implicitly. Therefore, you need to force their evaluation, by using the ***is*** operator (X is <expression>). Therefore:

```
gcd(X,Y,Z):- X>Y, R is X-Y, gcd(R,Y,Z). % Y<X, clause 2
gcd(X,Y,Z):- X<Y, R is Y-X, gcd(X,R,Z). % X<Y, clause 3
```

Let us follow the execution of several queries for the *gcd/3* predicate (use the trace command):

```
?- gcd(3, 3, X).
      1    1 Call: gcd(3,3,_407) ?
?     1    1 Exit: gcd(3,3,3) ?  % unifies with clause 1, stop
X = 3 ? ;                        % solution, repeat the question
     1    1 Redo: gcd(3,3,3) ? % attempt to unify query with following clause
     2    2 Call: 3>3 ?  % first call in body of clause 2
     2    2 Fail: 3>3 ?  % fail, attempt to unify with following clause
     3    2 Call: 3<3 ?   % first call in body of clause 3
     3    2 Fail: 3<3 ?  % fail, no clauses left
     1    1 Fail: gcd(3,3,_407) ? %fail
no
?- gcd(3, 7, X).
     1    1 Call: gcd(3,7,_407) ?  % initial call
     2    2 Call: 3>7 ?  % unify with head of the second clause, first call in body
     2    2 Fail: 3>7 ?  % fail
     3    2 Call: 3<7 ?  % unify with clause 3, first call in body
     3    2 Exit: 3<7 ? % success
     4    2 Call: _861 is 7-3 ? % second call in body of clause 3
     4    2 Exit: 4 is 7-3 ? % success
     5    2 Call: gcd(3,4,_407) ? % third call in body of clause 3
```

```
      6      3 Call: 3>4 ? % unify with clause 2, first call in body
      6      3 Fail: 3>4 ? % fail
      7      3 Call: 3<4 ? % unify with clause 3, first call in body
      7      3 Exit: 3<4 ? % success
      8      3 Call: _3317 is 4-3 ? % second call in body of clause 3
      8      3 Exit: 1 is 4-3 ? % success
      9      3 Call: gcd(3,1,_407) ? % … and so on…
     10      4 Call: 3>1 ?
     10      4 Exit: 3>1 ?
     11      4 Call: _5773 is 3-1 ?
     11      4 Exit: 2 is 3-1 ?
     12      4 Call: gcd(2,1,_407) ?
     13      5 Call: 2>1 ?
     13      5 Exit: 2>1 ?
     14      5 Call: _8229 is 2-1 ?
     14      5 Exit: 1 is 2-1 ?
     15      5 Call: gcd(1,1,_407) ?
?    15      5 Exit: gcd(1,1,1) ?
?    12      4 Exit: gcd(2,1,1) ?
?     9      3 Exit: gcd(3,1,1) ?
?     5      2 Exit: gcd(3,4,1) ?
?     1      1 Exit: gcd(3,7,1) ?
X = 1 ? ;
      1      1 Redo: gcd(3,7,1) ?
      5      2 Redo: gcd(3,4,1) ?
      9      3 Redo: gcd(3,1,1) ?
     12      4 Redo: gcd(2,1,1) ?
     15      5 Redo: gcd(1,1,1) ?
     16      6 Call: 1>1 ?
     16      6 Fail: 1>1 ?
     17      6 Call: 1<1 ?
     17      6 Fail: 1<1 ?
     15      5 Fail: gcd(1,1,_407) ?
     18      5 Call: 2<1 ?
     18      5 Fail: 2<1 ?
     12      4 Fail: gcd(2,1,_407) ?
     19      4 Call: 3<1 ?
     19      4 Fail: 3<1 ?
      9      3 Fail: gcd(3,1,_407) ?
      5      2 Fail: gcd(3,4,_407) ?
      1      1 Fail: gcd(3,7,_407) ?
```

no

Exercise 2.1: Trace the execution of the following queries, repeating the question:

1. ?- gcd(30, 24,X).
2. ?- gcd(15, 2, X).
3. ?- gcd(4, 1, X).

Factorial

The *factorial* of a number is again defined as a recurrent mathematical relation:

fact(0)=1
*fact(n) =n*fact(n-1), n>0*

Let's write a predicate which computes the factorial of a natural number (below). Note that again you need to use the *is* operator to force the evaluation of a mathematical expression:

```
fact(0,1).
fact(N,F):-N1 is N-1, fact(N1,F1), F is F1*N.
```

Exercise 2.2: Follow the execution of the following queries, repeating the question:

1. ?- fact(6, 720).
2. ?- N=6, fact(N, 120).
3. ?- fact(6, F).
4. ?- fact(N,720).
5. ?- fact(N,F).

Questions 2.1: Why do you think the execution enters an infinite loop when repeating the question? Why do you get an error for queries 3 and 5?
Answers: a. The last call in the deduction tree is fact(0,_someInternalFreeVariable), which has been matched with the first clause. When repeating the question, this call is matched with the second clause, N reaches -1, in the next call -2, ...a.s.o. To prevent this, we should add at the beginning of the second clause: N>0; *therefore, the body of the second clause is:*

```
N>0, N1 is N-1, fact(N1,F1), F is F1*N.
```

b. this predicate is not reversible; i.e. you cannot change the direction of the input/output parameters.

The above version for the factorial predicates builds the solution as recursion returns, i.e. for the factorial of *n*, it assumes that we have already computed the factorial of *n-1* (just like in the recurrence formula). Is there another way to write the predicate which computes the factorial of a number? The answer is, of course, ***yes***: assume we start the computation from *n*, at each step multiply the partial result with the current natural number and get to the previous natural number; stop when we reach *0*. Let's see how such a predicate looks like:

```
fact1(0, FF, FF).
fact1(N, FP, FF):-N>0, N1 is N-1, FP1 is FP*N, fact1(N1, FP1, FF).
```

Question 2.2: How do you call/query the fact1/3 predicate, to get the factorial of 6, for example?

Answer: ?- fact1(6, 1, F)*, i.e. you must initialize the accumulation parameter with the neutral (default) element, which for "*" is 1.*

Exercise 2.2: Follow the execution of the following queries, repeating the question:

1. ?- fact1(6, 1, F).
2. ?- fact1(2, 0, F).

For such predicates in which you employ an accumulation parameter, which has to be initialized at call time, you may write a pretty call, which hides this initialization. For example, for the fact1/3 predicate, you may write:

```
fact1_pretty(N,F):-fact1(N,1,F).
```

This way, you no longer have to worry about the correct initialization value for the accumulator.

FOR loop

Even if repetitive control structures are not specific to Prolog programming, they can be easily implemented. Let us take a look at an example for the *for* loop:

for(int i=n; i>0; i--) {….}

In Prolog, this would look like:

```
for(In,In,0):-!.
for(In,Out,I):-
        NewI is I-1,
        do(In,Intermediate),
        for(Intermediate,Out,NewI).
```

Exercise 2.3: Write a predicate forLoop/3 which computes the sum of all integers smaller than some integer (e.g. forLoop(0, Sum , 9) should output: 45). Trace the execution on several queries on your predicate.

2.2 Quiz Exercices

2.2.1 ***Least Common Multiplier**:* write a predicate which computes the least common multiplier of two natural numbers (*Hint: the least common multiplier of two natural numbers is equal to the ratio between their product and their gcd*).

2.2.2 ***Fibonacci Sequence**:* write a predicate which computes the nth number in the Fibonacci sequence. The recurrence formula for the Fibonacci sequence is:

fib(0)=1
fib(1)=1
fib(n) =fib(n-1)+fib(n-2), n>1

2.2.3 **Repeat….until:** write a predicate which simulates a *repeat…until* loop and prints all integers between *Low* and *High*.

Hint: the structure of such a loop is:
repeat
<do something>
until <some condition>

2.2.4 **While:** write a predicate which simulates a *while* loop and prints all integers between *Low* and *High*.

Hint: the structure of such a loop is:
while <some condition>
<do something>
end while

2.3 Problems

2.3.1 ***Triangle Inequality:*** the triangle inequality states that for any triangle, the sum of the lengths of any two sides must be greater than the length of the remaining side. Write a predicate triangle/3, which verifies if the arguments can form the sides of a triangle.

2.3.2 ***2nd order equation***: write a predicate solve_eq2/4 which solves a second order equation of the form $ax^2+bx+c=0$.

3. Lists. List Operations (I)

The **list** is the simplest yet the most useful Prolog structure. A list is a sequence of any number of objects.

Example 3.1: L = [1, 2, 3], R = [a, b, c], T = [john, marry, tim, ana].

The standard representation of a list in Prolog is:

- *the empty list*: [], which is an atomic element
- *the list containing at least 1 element*: [H|T], where H is the first element in the list (the *head*), while T represents the rest of the list (its *tail*). Thus, H can be any Prolog object, while T is necessarily a list.

Example 3.2: L = [1, 2, 3], L = [1|[2, 3]], L = [1|[2|[3]]], L = [1|[2|[3|[]]]].

Exercise 3.1: Evaluate the following unification queries:

```
L = [a, [b, [c]]].
[a, b, c, d] = [a|[b, [c|[d]]]].
R = [a|[b, c]].
[a, b, c, d] = [a, b|[c|[d]]].
```

3.1 List Operations

Lists may be employed to model sets. There are a few differences between the two types, such as: the order of elements in a set doesn't matter, while the elements in a list appear in a certain order; also, lists may contain duplicate elements. However, the fundamental list operations are similar to the operations on sets: membership, append, add/remove element, etc. Moreover, set-specific operations, such as union, intersection, difference, Cartesian product, can easily be implemented using lists.

3.1.1 Member – Search for an element

We will develop the predicate member(X, L), which determines whether element X belongs to list L. The query to member(X, L) will succeed if element X can be found in list L, and fail otherwise.

The following reasoning can be applied: element X is member in list L if X is the head of the list or if X is in L's tail:

$$If\ L = [H|T], then\ X \in L \Leftrightarrow X = H \vee X \in T$$

Thus, the predicate member(X, L) has two clauses:

```
member(X, [H|T]):- H=X.
member(X, [H|T]):-member(X,T).
```

In the first clause of the predicate, we can replace the explicit unification H=X by an implicit one, i.e. use the same variable name. Thus, the first clause becomes:

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

Also, T appears only once in the first clause – it is a singleton variable. Since we are not interested in its value – we don't care what is in the list's tail if we have found the element in its head – we can use the "*don't care*" variable (_). The same is true for variable H in the second

clause. Sicstus Prolog issues a warning for each singleton variable in the predicate specifications, since the existence of a singleton variable may also indicate a flaw in the specification of the predicate (e.g. a variable is spelled incorrectly). Thus, we rewrite the two clauses like this:

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

Example 3.3: Let's analyze the execution of the following query, using the trace *option:*

```
?- member(3, [1, 2, 3, 4]).
```

Info: Trace is an option which can be activated by using the trace. *command. Te deactivate, use* notrace.

Warning: Versions of Sicstus Prolog newer than v4 have implemented the member predicate in their core library. Therefore, to test your own implementation of member, you have to rename it, to member1 for example.

Therefore, after renaming predicate member to member1, the execution trace of the query in example 3.3 is:

```
| ?- member1(3, [1, 2, 3, 4]).
      1      1 Call: member1(3,[1,2,3,4]) ?  % unifies with second clause -> new call
      2      2 Call: member1(3,[2,3,4]) ?  % unifies with second clause -> new call
      3      3 Call: member1(3,[3,4]) ?  % unifies with first clause -> stop, success
?     3      3 Exit: member1(3,[3,4]) ?  % exit call 3
?     2      2 Exit: member1(3,[2,3,4]) ?  % exit call 2
?     1      1 Exit: member1(3,[1,2,3,4]) ?  % exit call 1
yes
```

Hint: line comments in Prolog are marked by %.

Example 3.4: Let us trace the execution of the following query, repeating the question:

```
| ?- X=3, member1(X, [1, 2, 3, 4]).
      1      1 Call: _371=3 ?  % call X = 3
      1      1 Exit: 3=3 ?  % exit X = 3, X is now instantiated to constant 3
      2      1 Call: member1(3,[1,2,3,4]) ?
      3      2 Call: member1(3,[2,3,4]) ?
      4      3 Call: member1(3,[3,4]) ?
?     4      3 Exit: member1(3,[3,4]) ?        % same as example 3.3
?     3      2 Exit: member1(3,[2,3,4]) ?
?     2      1 Exit: member1(3,[1,2,3,4]) ?
X = 3 ? ;                                       % repeat the question
      2      1 Redo: member1(3,[1,2,3,4]) ?
      3      2 Redo: member1(3,[2,3,4]) ?           % go to the last node built, and redo it
      4      3 Redo: member1(3,[3,4]) ?            % unification of this query with clause 1 is
  % destroyed; unifies with clause 2 => new
  % call
      5      4 Call: member1(3,[4]) ?       % unifies with clause 2 -> new call
      6      5 Call: member1(3,[]) ?  % doesn't unify with any clause =>
      6      5 Fail: member1(3,[]) ?   % call 5 fails
      5      4 Fail: member1(3,[4]) ?  % no other possible resolution for call 4 => fails
      4      3 Fail: member1(3,[3,4]) ?  % no other resolution for call 3 => fails
```

3 2 Fail: member1(3,[2,3,4]) ? % *no other resolution for call 2 => fails*
2 1 Fail: member1(3,[1,2,3,4]) ? % *no other resolution for call 1 => fails*
no

You may have observed that the second query in example 3.4 is identical to the query in example 3.3. Due to the artifice of using a variable in the goal query which is previously instantiated to its intended value (X = 3), we were able to repeat the question. Sicstus does not allow you to repeat the question unless there were some variables in your query – for which it will return the instantiations, if the query ended with success.

Otherwise, it will simply answer yes, or no. Use this artifice whenever you need to repeat queries which might have several answers, and Sicstus does not allow you to repeat the question.

Exercise 3.2: Execute and trace the following queries for the predicate member1(X, L). *Repeat the question and study the behavior of the predicate:*

1. ?- member1(a, [a, b, c, a]).
2. ?- X=a, member1(X, [a, b, c, a]).
3. ?- member1(a, [1, 2, 3]).

The possibility of asking different types of queries for the same predicate is an aspect of the flexibility Prolog offers. Thus, through the query member1(X, [a, b, c, d]) we can extract an element from an instantiated list:

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

X = a? ;
X = b? ;
X = c? ;
X = d? ;
no.

This is a nondeterministic behavior. Let us analyze it more closely: the query member1(X, [a, b, c, d]) will match with the first clause of the predicate (a fact), resulting in the following unifications:

$X = X_1$ $\quad\quad$ $X = X_1 = a$
$[a, b, c, d] = [X_1|T_1]$ $\quad\quad$ $T_1 = [b, c, d]$ $\quad\quad$ (1)

Since the query has been successfully unified with a fact, the execution ends with success, providing the first answer, X = a.

By repeating the question, the unifications in (1) are destroyed and the query member1(X, [a, b, c, d]) is matched against the head of the second clause:

$X = X_1$ $\quad\quad$ $X = X_1$
$[a, b, c, d] = [H_1|T_1]$ $\quad\quad$ $H_1 = a$
$\quad\quad$ $T_1 = [b, c, d]$

The unifications succeed; therefore the execution proceeds with the body of the second clause, resulting in the sub-query:

?- member1(X, [b, c, d]).

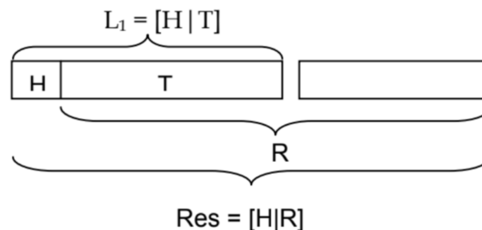
This query unifies with the first clause, resulting in the second solution: X = b. When we repeat the question again, the query above unifies with the second clause of the predicate, and generates a new sub-query. This leads to the third solution, X = c, etc.

Exercise 3.3: Trace the execution of the query: ?- member1(X, [a, b, c, d]).

Exercise 3.4: Reverse the order of the two clauses of predicate member1(X, L) *and repeat the queries performed so far. Comment on the changes observed.*

3.1.2 Append – Concatenation of two lists

The predicate append(L1, L2, R) joins the elements of lists L_1 and L_2 into a new list, R. The concatenation of two lists is illustrated below:



Again two cases are considered:

If L_1 is empty, then the result is the second list:

append([], L_2, Res):- L_2 = Res.

If L_1 has at least one element, i.e. L_1 = [H|T], then H is added in front of the list resulting from the concatenation of T and L_2:

append([H|T], L_2, Res):-append(T, L_2, R), Res = [H|R].

Thus, the concatenation of two lists can be written as:

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

We have replaced the explicit unifications with the implicit ones. Also, in the second clause we have constructed the result directly in the head of the second clause. We shall employ this mechanism often.

Warning: Predicate append is another basic predicate which has been implemented in newer versions of Sicstus Prolog. Therefore, to test your own implementation of append, you have to rename it, to append1 for example.

Example 3.5: Let us trace the execution of the following query:

```
| ?- append1([a, b], [c, d], R).
    1    1 Call: append1([a,b],[c,d],_451) ? % unifies with clause 2 -> recursive call 2
    2    2 Call: append1([b],[c,d],_930) ?  % unifies with clause 2 -> recursive call 3
    3    3 Call: append1([],[c,d],_1406) ?  % unifies with clause 1 -> stop, success
    3    3 Exit: append1([],[c,d],[c,d]) ?  % exit call 3
    2    2 Exit: append1([b],[c,d],[b,c,d]) ?  % exit call 2
    1    1 Exit: append1([a,b],[c,d],[a,b,c,d]) ?  % exit call 1
R = [a,b,c,d] ?; % resulting unifications; repeat the question (;)
no
```

Predicate append also allows for nondeterministic behavior – for example, to decompose a list in two parts. *Example 3.6* below explores this behavior.

Example 3.6: Let us study the execution of the following nondeterministic query (without tracing):

```
| ?- append1(T, L, [a, b, c, d]).

L = [a,b,c,d],
T = [] ? ;

L = [b,c,d],
T = [a] ? ;

L = [c,d],
T = [a,b] ? ;

L = [d],
T = [a,b,c] ? ;

L = [],
T = [a,b,c,d] ? ;

No.
```

Therefore, by repeating the question, we obtain all the possible decompositions of the list in two sub-lists. Let us now trace the same call:

```
| ?- append1(T, L, [a, b, c, d]).
      1    1 Call: append1(_191,_211,[a,b,c,d]) ?   % initial call, call 1
?     1    1 Exit: append1([],[a,b,c,d],[a,b,c,d]) ? % match and unify with the first
                                                      % clause ->stop, first answer
L = [a,b,c,d],
T = [] ? ;                                            % repeat the question
      1    1 Redo: append1([],[a,b,c,d],[a,b,c,d]) ?  % last call (call1) must match
                                                      % second clause now
      2    2 Call: append1(_760,_211,[b,c,d]) ?    % =
> recursive call, …
?     2    2 Exit: append1([],[b,c,d],[b,c,d]) ?   % … which matches clause 1 -> stop
?     1    1 Exit: append1([a],[b,c,d],[a,b,c,d]) ? % exit call 1

L = [b,c,d],                                          % second answer
T = [a] ? ;                                           % repeat query
      1    1 Redo: append1([a],[b,c,d],[a,b,c,d]) ?
      2    2 Redo: append1([],[b,c,d],[b,c,d]) ?      % this is last call (call2), redo it ->
                                                  % match second clause
      3    3 Call: append1(_1123,_211,[c,d]) ?        % => recursive call …
?     3    3 Exit: append1([],[c,d],[c,d]) ?          % which matches clause 1-> stop
?     2    2 Exit: append1([b],[c,d],[b,c,d]) ?   % exit call2
?     1    1 Exit: append1([a,b],[c,d],[a,b,c,d]) ?   % exit call1

L = [c,d],                                        % third answer
```

```
T = [a,b] ? ;                                   % repeat query
        1    1 Redo: append1([a,b],[c,d],[a,b,c,d]) ?
        2    2 Redo: append1([b],[c,d],[b,c,d]) ?
        3    3 Redo: append1([],[c,d],[c,d]) ?          % last call (call3), redo it -> match
                                                        % second clause
        4    4 Call: append1(_1485,_211,[d]) ?  % => recursive call…
?       4    4 Exit: append1([],[d],[d]) ?              % …which matches clause 1-> stop
?       3    3 Exit: append1([c],[d],[c,d]) ?           % exit call3
?       2    2 Exit: append1([b,c],[d],[b,c,d]) ?  % exit call2
?       1    1 Exit: append1([a,b,c],[d],[a,b,c,d]) ?   % exit call1

L = [d],                                        % fourth answer
T = [a,b,c] ? ;                                 % repeat query
        1    1 Redo: append1([a,b,c],[d],[a,b,c,d]) ?
        2    2 Redo: append1([b,c],[d],[b,c,d]) ?
        3    3 Redo: append1([c],[d],[c,d]) ?
        4    4 Redo: append1([],[d],[d]) ?              % last call (call4), redo it -> match
second clause
        5    5 Call: append1(_1846,_211,[]) ?   % => recursive clause …
?       5    5 Exit: append1([],[],[]) ?        % …which matches clause 1 -> stop
?       4    4 Exit: append1([d],[],[d]) ?              % exit call4
?       3    3 Exit: append1([c,d],[],[c,d]) ?          % exit call3
?       2    2 Exit: append1([b,c,d],[],[b,c,d]) ?  % exit call2
?       1    1 Exit: append1([a,b,c,d],[],[a,b,c,d]) ?  % exit call1

L = [],                                         % fifth solution
T = [a,b,c,d] ? ;                               % repeat the query
        1    1 Redo: append1([a,b,c,d],[],[a,b,c,d]) ?
        2    2 Redo: append1([b,c,d],[],[b,c,d]) ?
        3    3 Redo: append1([c,d],[],[c,d]) ?
        4    4 Redo: append1([d],[],[d]) ?
        5    5 Redo: append1([],[],[]) ?        % last call (5), redo it -> no other
                                                % choice
        5    5 Fail: append1(_1846,_211,[]) ?           % fail call5
        4    4 Fail: append1(_1485,_211,[d]) ?          % no other choice for call4, fail
        3    3 Fail: append1(_1123,_211,[c,d]) ?        % no other choice for call3, fail
        2    2 Fail: append1(_760,_211,[b,c,d]) ?       % no other choice for call2, fail
        1    1 Fail: append1(_191,_211,[a,b,c,d]) ?     % no other choice for call1, fail

no                                                      % fail
```

Exercise 3.5: Study (by tracing) the execution of the following queries:

```
?- append1([1, [2]], [3|[4, 5]], R).
?- append1(T, L, [1, 2, 3, 4, 5]).
?- append1(_, [X|_], [1, 2, 3, 4, 5]).
```

Repeat the question and study the behavior; explain the functionality of each query (what it achieves).

As you already know, the order of the clauses of a predicate is very important in Prolog. Will append work if we reverse the order of its two clauses? Which are the differences in

behavior for different append queries, if we reverse the order of the clauses? You shall explore these issues in the following exercise:

Exercise 3.6: Reverse the order of the two clauses of predicate append1*, and study (with trace) the execution of the following queries, trying to answer the questions above:*

```
?- append1([1, [2]], [3|[4, 5]], R).
?- append1(T, L, [1, 2, 3, 4, 5]).
?- append1(_, [X|_], [1, 2, 3, 4, 5]).
```

3.1.3 Delete – Remove an element from a list

In order to delete a given element from a list, we have to traverse the list until the element is found, and build the result from all the elements except the one to be deleted:

```
% element found in head of the list, don't add it to the result
delete(X, [X|T], T).
%traverse the list, add the elements H≠X back to the result
delete(X, [H|T], [H|R]):-delete(X, T, R).
delete(_, [], []).
```

The third clause covers the case when the element to delete is not a member of the list. Since we do not want the predicate to fail if it cannot find the element to delete, we specify a clause for this case.

Exercise 3.7: Study the execution of the following queries:

```
?- delete(3, [1, 2, 3, 4], R).
?- X=3, delete(X, [3, 4, 3, 2, 1, 3], R).
?- delete(3, [1, 2, 4], R).
?- delete(X, [1, 2, 4], R).
```

Redo the queries, and repeat the question for each query. How many answers does each query have? Which is the order of the answers?

As you may have observed, predicate delete removes one occurrence of the element at a time. When the question is repeated, it will remove the next occurrence, leaving unaffected all previous ones, and so on, until no occurrence of the element is found, when it answers no.

Let us try to write a predicate which removes all the occurrences of a given element from a list. To achieve this, the search must continue once an occurrence is found and removed. Thus, if we take a look at predicate delete it is easy to grasp the changes required: in clause 1 of the predicate we must place a recursive call to remove the other occurrences as well:

```
delete_all(X, [X|T], R):- delete_all(X, T, R).
delete_all(X, [H|T], [H|R]):- X\=H, delete_all(X, T, R).
delete_all(_, [], []).
```

Exercise 3.8: Study the execution of the following queries:

```
?- delete_all(3, [1, 2, 3, 4], R).
?- X=3, delete_all(X, [3, 4, 3, 2, 1, 3], R).
?- delete_all(3, [1, 2, 4], R).
?- delete_all(X, [1, 2, 4], R).
```

3.2 Quiz exercises

3.2.1 Write the predicate append3(L_1, L_2, L_3, R), which achieves the concatenation of 3 lists.

3.2.2 Write a predicate which adds an element at the beginning of a list.

3.2.3 Write a predicate which computes the sum of the elements of a list of integers.

3.3 Problems

3.3.1. Write a predicate which takes as input a list of integers, L, and produces two lists: the list containing the even elements from L and the list of odd elements from L.

```
?- separate_parity([1, 2, 3, 4, 5, 6], E, O).
E = [2, 4, 6]
O = [1, 3, 5] ? ;
no
```

Hint: search the manual for the operator modulo.

3.3.2. Write a predicate which removes all the duplicate elements in a list (keep either the first or the last occurrence).

```
?- remove_duplicates([3, 4, 5, 3, 2, 4], R).
R = [3, 4, 5, 2] ? ;          or          R = [5, 3, 2, 4];
no                                        no
```

3.3.3. Write a predicate which replaces all the occurrences of element K with NewK in list L.

```
?- replace_all(1, a, [1, 2, 3, 1, 2], R).
R = [a, 2, 3, a, 2] ? ;
no
```

3.3.4. Write a predicate which deletes every K^{th} element from a list.

```
?- drop_k([1, 2, 3, 4, 5, 6, 7, 8], 3, R).
R = [1, 2, 4, 5, 7, 8] ? ;
no
```

4. The Cut (!). List Operations – Forward and Backward Recursion (II)

In this lesson we shall continue the discussion on list predicates and also review a useful Prolog element – the cut (!) – and two techniques: forward and backward recursion.

4.1 The Cut (!)

The **cut (!)** is a Prolog feature which is used to cut alternative branches of computation and, thus, these branches are not explored by backtracking. It can improve the efficiency of Prolog programs; however, predicates which contain "!" are more difficult to follow.

The "!" acts as a marker, back beyond which Prolog will not go. When it passes this point all choices that is has made so far are "set"; i.e. they are treated as though they were the only possible choices.

A generic clause including a cut operator has the following form:

p :- b_1, . . . , b_k, !, b_{k+2}, b_n.

When a clause with a cut operator is executed, if the current goal unifies with p and b_1, … , b_k return success:

- every other clause of p that unifies with the current goal is discarded from the search tree
- every branch open in b_1, …, b_k is discarded from the search tree

Therefore, the first node in the execution tree which is allowed to backtrack is the first node to the left of the node for goal p – the node for p and the nodes for b_1, …, b_k are not allowed to backtrack.

In summary, what you need to know about cut is:

1. *Any variables which are bound to values at this point cannot take on other values*
2. *No other clauses of predicates called before the cut will be considered*
3. *No other subsequent clauses of the predicate at the head of the current rule will be considered*
4. *The cut always succeeds*

An immediate usage of the "!" predicate is when having two mutually exclusive sub-goals in two different clauses of the same predicate:

```
p:-q, r, …
p:-q, s, ...
```

Sub-goals q and q are mutually exclusive: if q succeeds, the second clause cannot succeed, and vice versa. By placing a "!" in the first clause after sub-goal q, we eliminate the need of explicitly calling q in the second clause:

```
p:-q, !, r, …
p: s, ...
```

Let us review the predicates in the previous lesson: member and delete. As you have seen there, both predicates allow for non-deterministic behavior. We can employ the cut to transform these predicates into deterministic predicates.

The deterministic version of member is presented below:

```
member1(X, [X|_]):-!.
member1(X, [_|T]):-member1(X, T).
```

Therefore, any call to member1 will have only one answer. When repeating the question, the answer is no.

Example 4.1: Let us follow the execution of the query:

```
| ?- member1(X, [a, b, c, d]).
      1      1 Call: member1(_383,[a,b,c,d]) ?
      1      1 Exit: member1(a,[a,b,c,d]) ?
X = a ? ;
no
```

As it can be seen from the trace of the call, the cut does not allow for the node corresponding to the call member1(_383,[a,b,c,d]) to be resolved through other clauses and variable _383 to be rebound to another value. Therefore, when repeating the question, the query fails.

Exercise 4.1: Trace and study the execution of the following queries for the deterministic version of the member1 *predicate:*

1. ?- X=3, member1(X, [3, 2, 4, 3, 1, 3]).
2. ?- member1(X, [3, 2, 4, 3, 1, 3]).

Predicate delete in lesson 3 removed one occurrence of the element at a time. What if we needed the deterministic version of this predicate, i.e. a predicate which deletes the first and only first occurrence of an element from a list? This version of predicate delete is presented below:

```
delete(X, [X|T], T):-!.
delete(X, [H|T], [H|R]):-delete(X, T, R).
delete(_, [], []).
```

Example 4.2: Let us follow the execution of the query:

```
| ?- X=3, delete(X, [4, 3, 2, 3, 1, 3], R).
      1      1 Call: _371=3 ?
      1      1 Exit: 3=3 ?
      2      1 Call: delete(3,[4,3,2,3,1,3],_519) ?  % unify with clause 2 -> call 2
      3      2 Call: delete(3,[3,2,3,1,3],_1709) ?  % unify with clause 1 -> !, stop, success
      3      2 Exit: delete(3,[3,2,3,1,3],[2,3,1,3]) ?  % exit call 2
?     2      1 Exit: delete(3,[4,3,2,3,1,3],[4,2,3,1,3]) ?  % exit call 1
R = [4,2,3,1,3],
X = 3 ? ;                                            % solution 1, repeat question
      2      1 Redo: delete(3,[4,3,2,3,1,3],[4,2,3,1,3]) ?  % only call 1 allowed to backtrack
                                                     % because of ! in clause 1
      2      1 Fail: delete(3,[4,3,2,3,1,3],_519) ?  % no other possible resolution for call 1
no                                                   % fail
```

Thus, this version of the delete predicate removes only the first occurrence of the element.

Exercise 4.2: Study the execution of the following query:

1. ?- delete(X, [3, 2, 4, 3, 1, 3], R).

4.2 List Operations

We shall continue with the discussion on list predicates with the following examples: length, reverse and minimum. Forward and backward recursion will be exemplified on all three predicates.

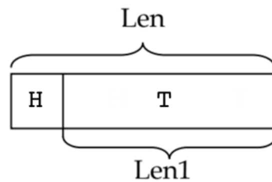
4.2.1 *Length*

The predicate which computes the length of a list is straightforward:

- The length of the empty list is 0
- The length of a non-empty list [H|T] is the length of T plus 1

Thus, the length predicate can be written as:

```
length([],0).
length([H|T], Len):-length(T,Len1), Len is Len1+1.
```



Hint: as opposed to other programming languages, in Prolog expressions are not evaluated implicitly. In order to evaluate an expression, you have to use is. *Usage*: Variable is <expression>.

Exercise 4.3: Study the execution of the following queries:

1. ?- length([a, b, c, d], Len).
2. ?- length([1, [2], [3|[4]]], Len).

This version of the length predicate applies a backward recursive approach: the result is built as the recursion returns; the result at level i needs the result at level i – 1. The result is initialized when the recursive calls stop and is built progressively, as each call returns. Thus, the final result is available at the top level.

Another approach is to count the elements of the list as the list is decomposed, and build the result as recursion proceeds (in an accumulator). This is the forward recursive approach. In order to do so, the accumulator must be initialized to 0 at the beginning. As the elements in the list are discovered, the length increases. This means that the final result will be available at the bottom level, when recursion ends. In order to make it available at the top level, we need to unify the accumulator with a free variable that is available at the top level.

Therefore, the forward recursive version of the length predicate is:

```
% when reaching the empty list, unify accumulator with the free result variable
length_fwd([], Acc, Res):-Res = Acc.
% as the list is decomposed, add 1 to the accumulator; pass Res unchanged
```

```
length_fwd([H|T], Acc, Res):-Acc1 is Acc+1, length_fwd(T, Acc1, Res).
```

Acc is the result accumulator, which must be initialized to 0 in the call:

```
?- length_fwd([a, b, c, d], 0, Res).
```

In order to make this call restriction transparent, we can write a wrapper predicate which performs the pretty call:

```
length_fwd_pretty(L, Len):-length_fwd(L, 0, Len).
```

Exercise 4.4: Study the execution of the following queries:

1. ?- length_fwd_pretty([a, b, c, d], Len).
2. ?- length_fwd_pretty([1, [2], [3|[4]]], Len).
3. ?- length_fwd([a, b, c, d], 3, Len).

4.2.2 *Reverse*

In order to reverse a list, the following strategy can be applied:

- the inverse of [] is [], and
- the inverse of a non-empty list [H|T] can be obtained by reversing T and adding H at the end of the resulting list:

```
reverse([], []).
reverse([H|T], Res):-reverse(T, R1), append(R1, [H], Res).
```

Exercise 4.5: Study the execution of the following queries:

1. ?- reverse([a, b, c, d], R).
2. ?- reverse([1, [2], [3|[4]]], R).

This is again the backward recursive version of the predicate: first we obtain the inverse of T (R_1), and construct the inverse of L = [H|T] by adding H at the end of R_1.

A forward recursive version of this predicate is:

```
reverse_fwd([], R, R).
reverse_fwd([H|T], Acc, R):-reverse_fwd(T, [H|Acc], R).
```

In the second clause, the elements of the list are added in the front of the accumulator as they are discovered. This makes them appear in reverse order in the accumulator. When the input list becomes empty (first clause), the inversed list is in the accumulator. By unifying the accumulator with the (until then) free result variable, the result is passed to the top level.

The pretty call for this predicate:

```
reverse_fwd_pretty(L, R):- reverse_fwd(L, [], R).
```

Exercise 4.6: Study the execution of the following queries:

1. ?- reverse_fwd_pretty([a, b, c, d], R).
2. ?- reverse_fwd_pretty([1, [2], [3|[4]]], R).
3. ?- reverse_fwd([a, b, c, d], [1, 2], R).

4.2.3 *Minimum* – Determine the minimum from a list

A first, natural solution for determining the minimum element of a list is to traverse the list element by element and keep, at each step, the minimum element so far. When the list becomes empty, the partial minimum becomes the global minimum. This corresponds to a forward recursion strategy:

```
minimum([], M, M).
minimum([H|T], MP, M):-H<MP, !, minimum(T, H, M).
minimum([H|T], MP, M):-minimum(T, MP, M).
```

The last two clauses of the predicate traverse the list: the second clause covers the case when the partial minimum has to be updated (a new partial minimum has been found), while in the last one the minimum is passed forward unchanged. The first clause represents the termination condition: the list becomes empty, so the partial minimum is unified with the (until then) free variable representing the result.

When querying this predicate, one must initialize MP. The most natural solution is to initialize it to the first element of the list:

```
minimum_pretty([H|T], R):-minimum([H|T], H, R).
```

Exercise 4.7: Study the execution of the following queries:

1. ?- minimum_pretty([1, 2, 3, 4], M).
2. ?- minimum_pretty([3, 2, 6, 1, 4, 1, 5], M).

Redo the queries, repeating the question. How many answers does each query have? Which is the order of solutions? (if it applies)

We can approach the minimum problem using backward recursion:

```
minimum_bwd([H], H).
minimum_bwd([H|T], M):-minimum_bwd(T, M), H>=M.
minimum_bwd([H|T], H):-minimum_bwd(T, M), H<M.
```

The difference is that the minimum update is performed as the recursive calls return (in clauses 2-3 the update is performed after the recursive calls). Therefore, the minimum is initialized at the bottom where recursion stops (in clause 1). There is no need for the third argument (required by forward recursion).

Exercise 4.8: Study (using trace) the execution of the following queries:

1. ?- minimum_bwd([1, 2, 3, 4], M).
2. ?- minimum_bwd([4, 3, 2, 1], M).
3. ?- minimum_bwd([3, 2, 6, 1, 4, 1, 5], M).
4. ?- minimum_bwd([], M).

Redo the queries, repeating the question. How many answers does each query have? Which is the order of solutions? (if it applies)

The specification of the minimum_bwd predicate can be improved if we consider the following observations:

- the two sub-goals (H<M and H>=M) in clauses 2 and 3 are complementary
- since the update of the minimum is performed as recursion returns, there is no point decomposing the list again when sub-goal 2 in clause 2 fails; it is sufficient to update the minimum up to that point:

```
minimum_bwd([H], H).
```

```
minimum_bwd([H|T], M):-minimum_bwd(T, M), H>=M, !.
minimum_bwd([H|T], H).
```

- now, if we analyze clauses 1 and 3 we see that the two can be combined into a single clause, which must be placed after the current second clause (*Why?*):

```
minimum_bwd([H|T], M):-minimum_bwd(T, M), H>=M, !.
minimum_bwd([H|T], H).
```

Exercise 4.9: Study (using trace) the execution of the following queries. Can you tell the difference between the improved implementation and the original implementation of the predicate?

1. ?- minimum_bwd([1, 2, 3, 4], M).
2. ?- minimum_bwd([4, 3, 2, 1], M).
3. ?- minimum_bwd([3, 2, 6, 1, 4, 1, 5], M).
4. ?- minimum_bwd([], M).

4.3. Operations on Sets

Given two lists with no duplicate elements, computes the list which contains all elements appearing at least in one of them.

```
union([ ],L,L).
union([H|T],L2,R) :- member(H,L2),!,union(T,L2,R).
union([H|T],L,[H|R]):-union(T,L,R).
```

Exercise 4.10: Trace the execution of the predicate for the following queries:

1. ?-union([1,2,3],[4,5,6],R).
2. ?-union([1,2,5],[2,3],R).
3. ?-union(L1,[2,3,4],[1,2,3,4,5]).
4. ?-union([2,2,3],[2,3,5],R).
5. ?-union(L1,L2,R).

Exercise 4.10. ***Set intersection****: Given two lists representing sets, give the elements occurring in both of the lists in a third list. Trace the execution of the predicate for the following queries:*

1. ?-inters([1,2,3],[4,5,6],R).
2. ?-inters([1,2,5],[2,3],R).
3. ?-inters(L1,[1,2,3,4,5],[2,3,4]).

Exercise 4.11 ***Set difference****: Given two lists with unique elements create a list containing all the elements appearing in the first, but not the second. Check the predicate by executing the following queries:*

1. ? – set_diff([1,2,3,4,7,8], [2,3,4,5],R).
2. ? – set_diff([1,2,3], [1,2,3,4,5],R).
3. ? – set_diff(L, [1,2,3],[4,5]).

4.4 Quiz exercises

4.4.1 Write a predicate which finds and deletes the minimum element in a list.

4.4.2 Write a predicate which reverses the elements of a list from the Kth element onward (suppose K is smaller than the length of the list).

4.4.3 Write a predicate which finds and deletes the maximum element from a list.

4.5 Problems

4.5.1 Write a predicate which performs *RLE* (*Run-length encoding*) on the elements of a list, i.e. pack ***consecutive*** duplicates of an element in *[element, no_occurences]* packs.

```
?- rle_encode([1, 1, 1, 2, 3, 3, 1, 1], R).
R = [[1, 3], [2, 1], [3, 2], [1, 2]]  ? ;
no
```

4.5.2 Write a predicate which rotates a list K positions to the right.

```
?- rotate_right([1, 2, 3, 4, 5, 6], 2, R).
R = [5, 6, 1, 2, 3, 4]  ? ;
no
```

4.5.3 (**) Extract K random elements from a list L, in a new list, R. *Hint: use random(MaxVal) function.*

```
?- rnd_select([a, b, c, d, e, f, g, h], 3, R).
 R = [e, d, a]  ? ;
no
```

5. Sorting Methods

Sorting represents one of the most common problems in programming languages. As we will see further in this chapter, various sorting algorithms have a very elegant specification in Prolog.

5.1 Direct sorting methods

Direct sorting methods are the simplest, from the algorithmic point of view, sorting methods. They don't employ any specialized programming technique, but use simple techniques starting from the specifications.

5.1.1 Permutation sort

A possible approach to the sorting problem is to find the ordered permutation of a list. This natural strategy can be easily specified in Prolog:

```
perm_sort(L, R):-perm(L,R), is_ordered(R), !.
```

The predicate generates a permutation of the list L, then checks to see if it is ordered. If R is not ordered, the sub-query to is_ordered(R) will fail. The execution will backtrack with a new resolution for perm(L, R), resulting in a new permutation. This process continues until the ordered permutation is found. Of course, this approach, as natural as it is, is very inefficient from the algorithmic point of view. We have included it here due to its simplicity.

For generating the permutations of a list, we employ the following observation:

$n! = (n-1)! * n$

Thus, in order to obtain all the permutations of a list with n elements one has to extract randomly one element from the list (randomly = ensure that each element will be extracted at some point), obtain the permutations of the remaining n-1 elements, and then add the extracted element to the result.

The predicate which generates the permutations of a list:

```
perm(L, [H|R]):-append(A, [H|T], L), append(A, T, L1), perm(L1, R).
perm([], []).
```

The two calls to append in clause one of the predicate are complementary: the first call extracts an element H from the list L randomly, while the second recreates the list without H.

Exercise 5.1: Study the execution of the following queries (repeating the question):

1. ?- append(A, [H|T], [1, 2, 3]), append(A, T, R).
2. ?- perm([1, 2, 3], L).

Exercise 5.2: Reverse the order of the clauses of predicate append and redo the queries in *exercise 5.1*.

The predicate is_ordered is straightforward:

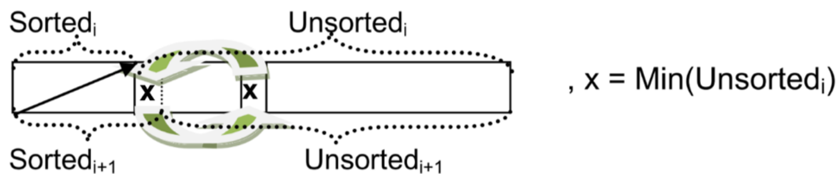
```
is_ordered([_]).
is_ordered([H1, H2|T]):-H1 =< H2, is_ordered([H2|T]).
```

Exercise 5.3: Study the execution of the following queries (repeating the question):

1. ?- is_ordered([1, 2, 4, 4, 5]).
2. ?- is_ordered([1, 2, 4, 2, 5]).
3. ?- perm_sort([1, 4, 2, 3, 5], R).

5.1.2 Selection sort

Selection sort works by selecting, at each step, the minimum (or maximum) element from the unsorted part of the list, and add it to the already sorted part, in the appropriate position:



The predicate which performs selection sort:

```
sel_sort(L, [M|R]):- min(L, M), delete(M, L, L1), sel_sort(L1, R).
sel_sort([], []).
```

The auxiliary predicates employed here have already been discussed earlier in this book. For a quick reference, go to chapter 3, sub-sections 1.5 and 1.6. This version of the selection sort predicate builds the solution as recursion returns. Therefore, the result variable holds the sorted part of the list, and the input list holds the unsorted part, which changes with each recursive call. The sorted part is initialized to [] when the recursive calls stop, and it grows as each recursive call returns, by adding the current minimum in front of the list.

Exercise 5.4: Modify the first clause of the predicate sel_sort such that it outputs intermediate results, and study the execution of the following queries:

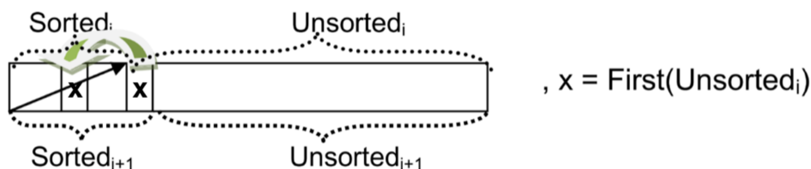
1. ?- sel_sort([3, 2, 4, 1], R).
2. ?- sel_sort([3, 1, 5, 2, 4, 3], R).

Hint: to output the value of a variable onscreen use the function write(). Consult the environment user manual for more information.

Exercise 5.5: Write a predicate which finds the minimum element from a list and deletes it (combines the functionality of min and delete).

5.1.3 Insertion sort

Insertion sort, as its name implies, inserts each element from the unsorted part in the appropriate position in the sorted part. All the elements (in the sorted part) which are greater than the current element considered (x) are shifted to the right in order to make room for x:



The predicate which performs insertion sort can be written as:

```
ins_sort([H|T], R):- ins_sort(T, R1), insert_ord(H, R1, R).
ins_sort([], []).

insert_ord(X, [H|T], [H|R]):-X>H, !, insert_ord(X, T, R).
insert_ord(X, T, [X|T]).
```

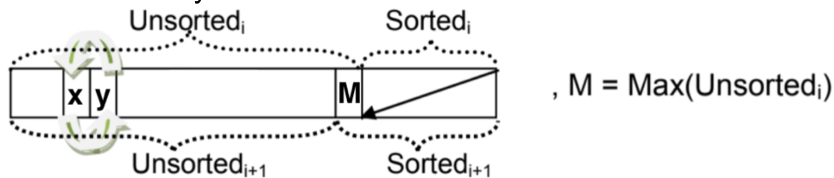
This is a backward recursive approach: first we obtain the result of the recursive call (R_1), then compute the result on the current level by inserting the current element in its right position in R_1 (predicate insert_ord).

Exercise 5.6: Modify the first clause of the predicate ins_sort such that it outputs intermediate results, and study the execution of the following queries:

1. ?- insert_ord(3, [], R).
2. ?- insert_ord(3, [1, 2, 4, 5], R).
3. ?- insert_ord(3, [1, 3, 3, 4], R).
4. ?- ins_sort([3, 2, 4, 1], R).
5. ?- ins_sort([3, 1, 5, 2, 4, 3], R).

5.1.4 Bubble sort

Bubble sort is one of the simplest direct sorting methods, but also the least efficient. It performs several passes through the data. In each pass it compares adjacent elements two by two and, if necessary, it swaps them. This approach ensures that, in each pass, at least the maximum element of the unsorted part reaches its final position – at the end of the unsorted part, or, better yet, the beginning of the sorted part. Therefore, in each pass at least the tail of the sequence is already sorted.



A possible Prolog specification for bubble sort:

```
bubble_sort(L, R):-one_pass(L, R1, F), nonvar(F), !, bubble_sort(R1, R).
bubble_sort(L, L).

one_pass([H1, H2|T], [H2|R], F):- H1>H2, !, F = 1, one_pass([H1|T], R, F).
one_pass([H1|T], [H1|R], F):-one_pass(T, R, F).
one_pass([], [] ,_).
```

We have selected the version which performs only as many passes through the data as necessary. When the list becomes sorted, the algorithm stops. This is controlled through a flag, F. Before each new pass, the flag is reset to a free variable. Whenever a swap is performed, F is instantiated to a constant value (1). After the pass we check the flag: if it has been instantiated (no longer a free variable), then at least a swap has been performed, meaning that the list may not be ordered. Thus, a new call to bubble_sort is required. If F remains free after the call to one_pass, then the call to nonvar(F) will fail. This means the list L is sorted, so we need to pass it to the result (second clause of the predicate bubble_sort).

Exercise 5.7: Study the execution of the following queries (outputting the intermediate results of bubble_sort if necessary):

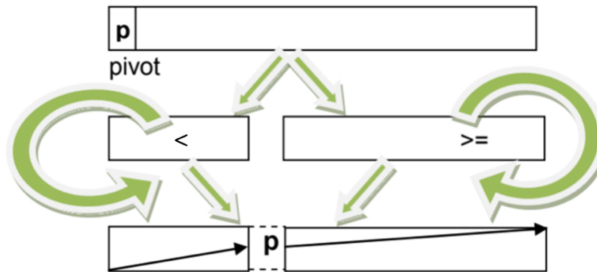
1. ?- one_pass([1, 2, 3, 4], R, F).
2. ?- one_pass([2, 3, 1, 4], R, F).
3. ?- bubble_sort([1, 2, 3, 4], R).
4. ?- bubble_sort([2, 3, 1, 4], R).
5. ?- bubble_sort([2, 3, 3, 1], R).

5.2 Advanced Sorting Methods

We will present two advanced sorting methods, based on the "*divide et impera*" technique: quicksort and merge sort. For the first technique the *impera* part of the algorithm is performed in *O(1)*, while the second performs the *divide* operation in constant time.

5.2.1 Quicksort

Quicksort's divide et impera strategy consists in partitioning the sequence in two, according to a pivot element: a sub-sequence containing the elements smaller than the pivot, and the sub-sequence containing the elements which are larger than or equal to the pivot. Then, apply the same strategy for each of the two sub-sequences. The process stops when the sequence to partition is empty. To compose the result, simply append the sorted sub-sequence of the elements which are smaller than the pivot with the pivot and the sub-sequence of larger elements:



The Prolog predicates which perform quicksort are presented below:

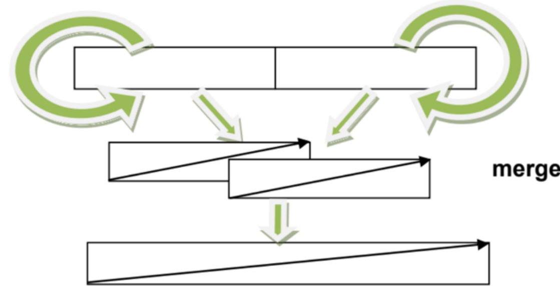
```
quick_sort([H|T], R):-partition(H, T, Sm, Lg), quick_sort(Sm, SmS),
                    quick_sort(Lg, LgS), append(SmS, [H|LgS], R).

quick_sort([], []).

partition(H, [X|T], [X|Sm], Lg):-X<H, !, partition(H, T, Sm, Lg).
partition(H, [X|T], Sm, [X|Lg]):-partition(H, T, Sm, Lg).
partition(_, [], [], []).
```

Exercise 5.8: Study the execution of the following queries (if necessary alter the predicates to allow you to visualize the intermediate results):

1. partition(3, [4, 2, 6, 1, 3], Sm, Lg).
2. quick_sort([3, 2, 5, 1, 4, 3], R).
3. quick_sort([1, 2, 3, 4], R).

5.2.2 Merge sort

Merge sort splits the sequence in two equal parts, applies the procedure recursively on each part to obtain the two sorted sub-sequences, which are then merged at the end:



In Prolog, we can specify the merge sort procedure in the following manner:

```
merge_sort(L, R):-split(L, L1, L2), merge_sort(L1, R1), merge_sort(L2, R2),
                        merge(R1, R2, R).
merge_sort([H], [H]).
merge_sort([], []).

split(L, L1, L2):-length(L, Len), Len>1, K is Len/2, splitK(L, K, L1, L2).

splitK([H|T], K, [H|L1], L2):- K>0, !, K1 is K-1, splitK(T, K1, L1, L2).
splitK(T, _, [], T).

merge([H1|T1], [H2|T2], [H1|R]):-H1<H2, !, merge(T1, [H2|T2], R).
merge([H1|T1], [H2|T2], [H2|R]):-merge([H1|T1], T2, R).
merge([], L, L).
merge(L, [], L).
```

The predicate splitK takes the first K elements in the input list L and adds them to L_1. The rest of L will be put in L_2. Therefore, the predicate split divides the list in two equal parts by calling splitK with argument K equal to the length of the list over 2 (K = length(L)/2). If the length of L is 0 or 1, then the query to split will fail, causing the resolution through clause 1 of merge_sort to fail – this is when the recursion should stop. Therefore, those calls will be matched through clauses 2 or 3 of the predicate merge_sort. The predicate merge performs the merging of two ordered lists. Its specification is straightforward.

Exercise 5.9: Study the execution of the following queries (if necessary alter the predicates to allow you to visualize the intermediate results):

1. ?- split([2, 5, 1, 6, 8, 3], L1, L2).
2. ?- split([2], L1, L2).
3. ?- merge([1, 5, 7], [3, 6, 9], R).
4. ?- merge([1, 1, 2], [1], R).
5. ?- merge([], [3], R).
6. ?- merge_sort([4, 2, 6, 1, 5], R).

5.3 Quiz exercises

5.3.1 For the predicate perm, the two calls to append, extract en element from a list randomly, and recreate the list without the selected element. Write the predicate(s) which perform these operations without using append, then write a new predicate, perm1, which generates the permutations of a list, using the new predicate(s) for extracting/deleting an element from a list.

5.3.2 Write a predicate which performs selection sort by selecting, in each step, the maximum element from the unsorted part, and not the minimum. Analyze its efficiency.

5.3.3 Write a forward recursive predicate which performs insertion sort. Analyze its efficiency in comparison with the backward recursive version.

5.3.4 Implement a predicate which performs bubble sort, using a fixed number of passes through the input sequence.

5.4 Problems

5.4.1 Suppose we have a list of ASCII characters. Sort the list according to their ASCII codes.

```
?- sort_chars([e, t, a, v, f], L).
L = [a, e, f, t, v] ? ;
no
```

Hint: search for char_code in the environment manual.

5.4.2 Suppose we have a list whose elements are lists containing atomic elements. Write a predicate(s) which sorts such a list according to the length of the sub-lists.

```
?- sort_len([[a, b, c], [f], [2, 3, 1, 2], [], [4, 4]], R).
R = [[], [f], [4, 4], [a, b, c], [2, 3, 1, 2]] ? ;
no
```

Hint: the ordering relation should be changed from the normal ordering relation on numbers to the one required here, i.e. on list lengths.

6. Deep lists

The deep list type in prolog represents a recursive structure, where several lists of variable depth are nested one in another. A trivial case of a deep list is a simple (or shallow) list. Formally a deep list can be defined by:

DL = [H|T], where H∈{atom, list, deep list} and T is a deep list

Examples of deep lists include:

- L1 = [1,2,3,[4]].
- L2 = [[1],[2],[3],[4,5]]
- L3 = [[],2,3,4,[5,[6]],[7]].
- L4 = [[[[1]]],1, [1]].
- L5 = [1,[2],[[3]],[[[4]]],[5,[6,[7,[8,[9],10],11],12],13]].
- L6= [alpha, 2,[beta],[gamma,[8]]].

6.1 Simple operations with deep lists

All operations defined for shallow lists can also be used with deep lists including (but not only) the studied member, append and delete predicates. To understand how operations on deep lists work, one can consider a deep list equivalent to a shallow list with different types of elements, but only those on the first level.

Exercise 6.1: Using the list L5 defined above try to give the result of the following queries:

```
? - member( 2 ,L5).
? – member( [2] , L5).
? – member(X, L5).
? – append(L1,R,L2).
? – append(L4,L5,R).
? – delete(1, L4,R).
? – delete(13,L5,R).
```

Execute the queries in Prolog to check your results.

6.2 Advanced operations with deep lists

6.2.1 The atomic predicate

To execute different operations on all elements of a deep list we have to know how to treat these elements according to their type(if they are atoms we process them, if they are lists further decomposition might be needed before processing). To find out if a given element is an atom or a complex structure the built in atomic predicate can be used.

Exercise 6.2: Answer then execute the following queries:

```
? – atomic(apple).
? – atomic(4).
? – atomic(X).
? – atomic( apple(2)).
```

? – atomic([1,2,3]).
? – atomic([]).

6.2.2. The depth of a deep list

The depth represents the maximum nesting level in the case of a deep list. The depth of an atom is defined as 0, and the depth of a shallow list(including the empty list) as 1. When computing the maximum depth we will have three branches:

We arrived at the empty list. The depth is 1.

We have an atomic head, we ignore it, since it doesn't influence the depth. The depth of the list will be equal to the depth of the tail.

We have a list in the head of the deep list. In this case the depth of the list will be either the depth of the tail, or the depth of the head increased by one(think about why).

The predicate corresponding to the description is:

```
depth([],1).
depth([H|T],R):-atomic(H),!,depth(T,R).
depth([H|T],R):- depth(H,R1), depth(T,R2), R3 is R1+1, max(R3,R2,R).
```

Exercise 6.3: Trace the execution of the following queries for the predicate depth, and the lists L1-6 defined as above:

? – depth(L1,R).
? – depth(L2,R).
? – depth(L3,R).
? – depth(L4,4).
? – depth(L5,R).
? – depth(L6,4).

6.2.3. Flattening a deep list

This operation means obtaining an equivalent shallow list from a deep list, containing all the elements, but with nesting level 1. In order to do this, we take only the atomic elements from the source list and place them in the result. We have again 3 main cases:

1. We have to deal with the empty list. Flattening the empty list results in an empty list.
2. If the first element of the list is atomic we put it into the result, and process the rest of the list.
3. If the first element is not atomic, the result will be composed of all the atomic elements(or flattening) of the head, and all the atomic elements(flattening) of the tail.(How do we collect the two results in a single list?)

The solution is:

```
flatten([],[]).
flatten([H|T], [H|R]) :- atomic(H),!, flatten(T,R).
flatten([H|T], R) :- flatten(H,R1), flatten(T,R2), append(R1,R2,R).
```

Exercise 6.4: Trace the execution of the following queries:

? – flatten(L1,R).

```
? – flatten(L2,R).
? – flatten(L3,R).
? – flatten(L,[1,2,3,4]).
? – flatten(L5,R).
? – flatten(L6, [alpha,2,beta,gamma,8]).
```

6.2.4. List heads

Returns all the atomic elements, which are at the head of a shallow list. Several solutions exist, we will present an efficient solution, which uses a flag to determine if we are at the first element of a list.

```
heads3([],[],_).
heads3([H|T],[H|R],1):-atomic(H),!,heads3(T,R,0).
heads3([H|T],R,0):-atomic(H),!,heads3(T,R,0).
heads3([H|T],R,_):-heads3(H,R1,1),heads3(T,R2,0), append(R1,R2,R).
heads_pretty(L,R) :- heads(L, R,1).
```

Exercise 6.5: Trace the execution of the following queries:

```
? – heads(L1,R).
? – heads(L2,R).
? – heads(L3,R).
? – heads(L6,R).
? – heads(L5,R).
? – heads(L,[1,2,3,4,5]).
```

6.2.5 The nested member function

Works similarly to the member function in the case of the shallow lists, considers as member all elements appearing in the list, atomic or not, at any level.

```
member1(H,[H|_]).
member1(X,[H|_]):-member1(X,H).
member1(X,[_|T]):-member1(X,T).
```

Exercise 6.6: Trace the execution of the following queries:

```
? – member1(1,L1).
? – member1(4,L2).
? – member1([5,[6]], L3).
? – member1(X,L4).
? – member1(X,L6).
? – member1(14,L5).
```

Observation: If we want our nested function to find only atomic elements we can use the flattening of the list to obtain a short solution:

```
member2(X,L):- flatten(L,L1), member(X,L1).
```

6.3 Quiz exercises

6.3.1 Define a predicate which computes the number of atomic elements in a deep list.

6.3.2 Define a predicate computing the sum of atomic elements from a deep list.

6.3.3 Define the deterministic version of the member predicate.

6.4 Problems

6.4.1 Define a predicate returning the elements from a deep lists, which are at the end of a shallow list (immediately before a ‘]’).

6.4.2 Write a predicate which replaces an element/list/deep list in a deep list with another expression.

6.4.3 Define a predicate ordering the elements of a deep list by depth (when 2 sublists have the same depth, order them in lexicographic order – after the order of elements).

Hint: L1< L2, if L1 and L2 are lists, or deep lists, and the depth of L1 is smaller than the depth of L2.

L1<L2, if L1 and L2 are lists, or deep lists with equal depth, all the elements up to the k-th are equal, and the k+1-th element of L1 is smaller than the k+1th element of L2.

7. Trees. Operations on trees

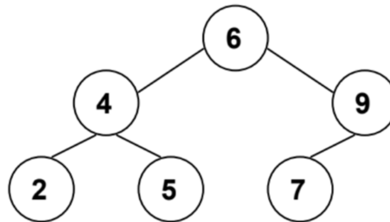
In this lesson you will explore operations on trees. Two types of trees will be addressed: binary search trees and ternary trees, with corresponding operations. In Prolog, trees are modeled as recursive structures. The empty tree is denoted through a constant, which is typically the symbol nil.

7.1 Binary Search Trees

Binary search trees can be represented in Prolog by using a recursive structure with three arguments: the *key* of the root, the *left sub-tree* and the *right sub-tree* – which are structures of the same type. The empty (null) tree is usually represented as the constant nil.

Example 7.1: The binary search tree below can be specified in Prolog using the following structure:

```
t(6, t(4, t(2, nil, nil), t(5, nil, nil)), t(9, t(7, nil, nil), nil)).
```



Hint: In order to avoid writing each time you want to make a query such a long construction for the input tree, you may choose to "save" a few "test" instances as predicate facts in the source file (and the predicate base), e.g.:

```
tree1(t(6, t(4, t(2, nil, nil), t(5, nil, nil)), t(9, t(7, nil, nil), nil))).
tree2(t(8, t(5, nil, t(7, nil, nil)), t(9, nil, t(11, nil, nil)))).
…
```

and instantiate a variable in the query:

```
?- tree1(T), some_useful_predicate(T, …).
```

7.1.1 Tree traversal – preorder, inorder, postorder

Perhaps the simplest operations on trees are the traversal operations. As you already know, there are three possible modes of traversing trees: *inorder*, *preorder* and *postorder*, depending on the order in which the nodes are processed.

The inorder traversal processes the left sub-tree first, then the root node, then the right sub-tree. The predicate is presented below:

```
inorder(t(K,L,R), List):-inorder(L,LL), inorder(R, LR),
                          append(LL, [K|LR],List).
inorder(nil, []).
```

You may observe that, even though the recursive call for the right sub-tree is performed before processing the root node, the correct order of the nodes is maintained when constructing

the output list, in the call to append. So, the nodes on the left sub-tree appear first in the list, then the root node, then the keys in the right sub-tree.

The same observation applies for the preorder and postorder traversals, presented below:

```
preorder(t(K,L,R), List):-preorder(L,LL), preorder(R, LR),
                        append([K|LL], LR, List).
preorder(nil, []).
postorder(t(K,L,R), List):-postorder(L,LL), postorder(R, LR),
                        append(LL, LR,R1), append(R1, [K], List).
postorder(nil, []).
```

Exercise 7.1: Study (by tracing) the execution of the following queries:

```
?- tree1(T), inorder(T, L).
?- tree1(T), preorder(T, L).
?- tree1(T), postorder(T, L).
```

7.1.2 Pretty print

Pretty printing trees in Prolog is very useful for visualizing the correctness of the other predicates on trees. The simplest strategy for pretty printing is to perform an *inorder* traversal of the tree, and print each node at a number of tabs equal to the *depth* at which the node appears in the tree. Also, each node is printed on a separate line. The root of the tree is considered to be at depth 0.

A pretty printing of the tree in *example 7.1* is presented below:

```
        2
    4
        5
6
        7
    9
```

If we study the listing above more closely we observe that the keys on the left are printed first, then the root, then the keys on the right. This suggests that an inorder traversal is suited for obtaining such a pretty print. Therefore, the predicate(s) which output the pretty printing above are:

```
% inorder traversal
        pretty_print(nil, _).
pretty_print(t(K,L,R), D):-D1 is D+1, pretty_print(L, D1), print_key(K, D),
                        pretty_print(R, D1).

% predicate which prints key K at D tabs from the screen left margin and then
% proceeds to a new line
print_key(K, D):-D>0, !, D1 is D-1, write('\t'), print_key(K, D1).
print_key(K, _):-write(K), nl.
```

Hint: nl sends a newline to the standard output stream; equivalent to write('\n').

Exercise 7.2: Study the execution of the following query:

?- tree2(T), pretty_print(T, 0).

7.1.3 Searching for a key

Because of the ordering of the keys in a binary search tree, searching for a given key is very efficient. The algorithm is sketched below:

```
if currentNode = null then return -1; //not found
if searchKey = currentNode.key then return 0; //found
else if searchKey < currentNode.key
        then search(searchKey, currentNode.left); //search left subtree
        else
        search(searchKey, currentNode.right); //search right subtree
```

It is very straightforward to transform the pseudo code above in Prolog specifications. Since we want our predicate to fail in case the key is not found, we can either specify this fact explicitly, by using an explicit fail for when a nil is reached, or implicitly, by not covering the case of reaching a nil. The search_key predicate is presented below:

```
search_key(Key, t(Key, _, _)):-!.
search_key(Key, t(K, L, _)):-Key<K, !, search_key(Key, L).
search_key(Key, t(_, _, R)):-search_key(Key, R).
```

Exercise 7.3: Study the execution of the following queries:

```
?- tree1(T), search_key(5, T).
?- tree1(T), search_key(8, T).
```

7.1.4 Inserting a key

Each new key is inserted as a leaf node in a binary search tree. Before performing the actual insert, we must search for the appropriate position of the new key. If the key is found during the search process, no insertion occurs. When reaching a nil in the search process, we create the new node.

The insert_key predicate is presented below:

```
insert_key(Key, nil, t(Key, nil, nil)):-write('Inserted '), write(Key), nl.
insert_key(Key, t(Key, L, R), t(Key, L, R)):-!, write('Key already in tree\n').
insert_key(Key, t(K, L, R), t(K, NL, R)):-Key<K, !, insert_key(Key, L, NL).
insert_key(Key, t(K, L, R), t(K, L, NR)):- insert_key(Key, R, NR).
```

Exercise 7.4: Study the execution of the following queries:

```
?- tree1(T), pretty_print(T, 0), insert_key(8, T, T1), pretty_print(T1, 0).
?- tree1(T), pretty_print(T, 0), insert_key(5, T, T1), pretty_print(T1, 0).
?- insert_key(7, nil, T1), insert_key(12, T1, T2), insert_key(6, T2, T3), insert_key(9, T3, T4), insert_key(3, T4, T5), insert_key(8, T5, T6), insert_key(3, T6, T7), pretty_print(T7, 0).
```

7.1.5 Deleting a key

The deletion of a key in a binary search tree also requires that the key be initially searched in the tree. Once found, we distinguish among three situations:

- *We have to delete a leaf node*
- *We have to delete a node with one child*
- *We have to delete a node with both children*

The first two cases are rather simple. For the third case we have two alternatives: either replace the node to delete with its predecessor (or successor) – by reestablishing the links correctly, or to "hang" the left sub-tree in the left part of the right sub-tree (or vice-versa).

We have implemented the first alternative in the delete_key predicate, below:

```
delete_key(Key, nil, nil):-write(Key), write(' not in tree\n').
        delete_key(Key, t(Key, L, nil), L):-!. % this clause covers also case for leaf
                                                (L=nil)
delete_key(Key, t(Key, nil, R), R):-!.
delete_key(Key, t(Key, L, R), t(Pred, NL, R)):-!, get_pred(L, Pred, NL).
delete_key(Key, t(K, L, R), t(K, NL, R)):-Key<K, !, delete_key(Key, L, NL).
delete_key(Key, t(K, L, R), t(K, L, NR)):- delete_key(Key, R, NR).
get_pred(t(Pred, L, nil), Pred, L):-!.
get_pred(t(Key, L, R), Pred, t(Key, L, NR)):-get_pred(R, Pred, NR).
```

Exercise 7.5: Study the execution of the following queries:

```
?- tree1(T), pretty_print(T, 0), delete_key(5, T, T1), pretty_print(T1, 0).
?- tree1(T), pretty_print(T, 0), delete_key(9, T, T1), pretty_print(T1, 0).
?- tree1(T), pretty_print(T, 0), delete_key(6, T, T1), pretty_print(T1, 0).
?- tree1(T), pretty_print(T, 0), insert_key(8, T, T1), pretty_print(T1, 0), delete_key(6, T1, T2),
pretty_print(T2, 0), insert_key(6, T2, T3), pretty_print(T3, 0).
```

7.1.6 Height of a binary tree

The height of a binary tree can be computed using the following idea:

- *the height of a nil node is 0*
- *the height of a node other than nil is the maximum between the height of the left sub-tree and the height of the right sub-tree, plus 1 (max{hLeft, hRight} + 1)*

Therefore, the predicate which computes the height of a binary tree can be specified in Prolog as:

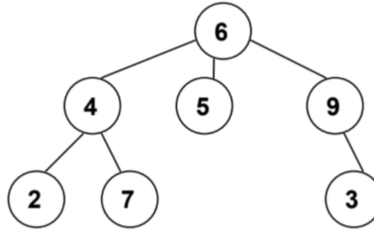
```
% predicate which computes the maximum between 2 numbers
max(A, B, A):-A>B, !.
max(_, B, B).

% predicate which computes the height of a binary  tree
height(nil, 0).
height(t(_, L, R), H):-height(L, H1),
                        height(R, H2),
                        max(H1, H2, H3),
                        H is H3+1.
```

Exercise 7.6: Study the execution of the following queries:

```
?- tree1(T), pretty_print(T, 0), height(T, H).
?- tree1(T), height(T,H), pretty_print(T,0), insert_key(8,T,T1), height(T1,H1), pretty_print(T1,0).
```

7.2 Ternary Trees

In a ternary tree, each node can have up to three children. Although it is not as easy as for binary trees, ordering relations can be established for ternary trees as well. For the sake of simplicity, we will not impose any ordering relation for the keys in a ternary tree. Below you have an example of a ternary tree:



We shall study a few operations for ternary trees. In Prolog, they are represented in the same manner as binary trees – through recursive structures.

7.2.1 Pretty print

Since a node in a ternary tree can have up to three children, we need a different strategy for pretty printing such structures than the one employed for binary trees. A solution would be to print each node at depth tabs from the left screen margin (as before); moreover, a node's sub-trees should appear bellow the node (should be printed after the node is printed), but above the next node at the same depth:

```
6
	4
		2
		7
	5
	9
		3
```

Such a pattern could be achieved through a pre-order traversal of the tree (*Root, Left, Middle, Right*), and printing each key on a line, at *depth* tabs from the left margin.

Exercise 7.7: Implement pretty printing for a ternary tree. Study the execution of one or two queries for your predicate.

7.2.2 Tree traversal

Tree traversal operations can be performed on ternary trees as well. The order of visiting the nodes in each of the tree walks is the following:

- inorder: *Left->**Root**->Middle->Right*
- preorder: ***Root**->Left->Middle->Right*
- postorder: *Left->Middle->Right->**Root***

Exercise 7.8: Implement the tree traversal operations for a ternary tree. Study the execution of different queries for the resulting predicates.

7.2.3 Tree height

The height of a ternary tree can be computed using the same idea from binary trees; the only difference is that you have to consider three branches, instead of two.

Exercise 7.9: Write a predicate which computes the height of a ternary tree. Study the execution of different queries for your predicate.

7.4 Quiz exercises

7.4.1 Alter the predicate for the inorder traversal of a binary search tree such that the keys are printed on the screen instead of collecting them in a list.

7.4.2 Alter the delete_key predicate for deleting a key from a binary search tree, such that when the key is in a node with two children you apply the second solution: “hang” the left sub-tree to the right sub-tree, or vice-versa.

7.4.3 Write a predicate which collects, in a list, all the keys found in leaf nodes of a binary search tree.

7.5 Problems

7.4.1 Write a predicate which computes the diameter of a binary tree (*diam(Root) = max{diam(Left), diam(Right), height(Left)+height(Right)+1}*).

```
?- tree1(T), diameter(T, D).
D =  5 ,
T = (6, t(4, t(2, nil, nil), t(5, nil, nil)), t(9, t(7, nil, nil), nil)) ? ;
no
```

7.4.2 (**) Write a predicate which collects, in a list, all the nodes at the same depth in a ternary tree.

7.4.3 (**) Let us call a binary tree *symmetric* if you can draw a vertical line through the root node and then the right sub-tree is the mirror image of the left sub-tree. Write a predicate symmetric(T) to check whether a given binary tree T is symmetric. We are only interested in the structure, not in the contents of the nodes.
Hint: Write a predicate mirror(T1, T2) first to check whether one tree is the mirror image of another.

```
?- tree1(T), symmetric(T).
        no

?- tree1(T), delete_key(2, T, T1), symmetric(T1).
        T = t(6,t(4,t(2,nil,nil),t(5,nil,nil)),t(9,t(7,nil,nil),nil)),
        T1 = t(6,t(4,nil,t(5,nil,nil)),t(9,t(7,nil,nil),nil)) ? ;
        no
```

8. Incomplete structures – lists and trees

Incomplete structures are special Prolog data elements, with the following particularity: instead of having a constant element at the end – such as [] for lists or nil for trees – they end in a free variable.

Example 8.1: Below you have a few examples of incomplete structures:

a. ?- L = [a, b, c|_].
b. ?- L = [1, 2, 3|T], T = [4, 5|U].
c. ?- T = t(7, t(5, t(3, _, _), _), t(11, _, _)).
d. ?- T = t(7, t(5, t(3, A, B), C), t(11, D, E)), D = t(9, F, G).

Incomplete structures, or partially instantiated structures, offer the possibility of altering the free variable at the end (instantiate it partially), and have the result in the same structure (addition at the end does not require an extra output argument). In order to reach the free variable at the end, incomplete structures are traversed in the same manner as complete structures. However, the clauses which specify the behavior when reaching the end must be explicitly stated (*even in the case of failure!*) and they must be placed in front of all the other predicate clauses – because the free variable at the end will unify with anything. To avoid undesired unifications, those cases must be treated first.

8.1 Incomplete lists

Incomplete lists are a special type of incomplete structures. Instead of ending in [], an incomplete list has a free variable as its tail. *Examples 8.1.a* and *8.1.b* show instances of such structures.

Incomplete lists are traversed in the same way as complete lists, using the [H|T] pattern; the difference comes when we have to process the end of the list – we are no longer dealing with [] at the end of the list, but with a free variable. This has the following implications on the predicates on incomplete lists:

- testing for the end of the list must ***always*** be performed, even for fail situations – and the fail must be explicit
- when testing for the end of the list, you have to check if you have reached the free variable at the end:

  some_predicate(L, …):-var(L), …
- the clause which checks for the end of the list must ***always*** be the first clause of the predicate (*Why?*)
- you may add any list at the end of an incomplete list, without needing a separate output structure (adding at the end of an incomplete list can be performed in the same input structure):

  e.g. ?- L = [1, 2, 3|T], T = [4, 5|U], U=[6|_].

Keeping these observations in mind, in the following we will transform a number of well-known list predicates such that they work on incomplete lists.

8.1.1 Member – member_il

Before writing the new predicate, let us check the behavior of the known member predicate (the deterministic version) on incomplete lists.

Exercise 8.1: Trace the execution of the following queries:

```
?- L = [1, 2, 3|_], member1(3, L).
?- L = [1, 2, 3|_], member1(4, L).
?- L = [1, 2, 3|_], member1(X, L).
```

As you have observed, when the element appears in the list, the predicate member1 behaves correctly; but when the element is not a member of the list, instead of answering no, the predicate adds the element at the end of the incomplete list – incorrect behavior. In order to correct this behavior, we need to add a clause which specifies the explicit fail when the end of the list (the free variable) is reached:

```
% must test explicitly for the end of the list, and fail
member_il(_, L):-var(L), !, fail.
% these 2 clauses are the same as for the member1 predicate
member_il(X, [X|_]):-!.
member_il(X, [_|T]):-member_il(X, T).
```

Exercise 8.2: Trace the execution of the following queries:

```
?- L = [1, 2, 3|_], member_il(3, L).
?- L = [1, 2, 3|_], member_il(4, L).
?- L = [1, 2, 3|_], member_il(X, L).
```

8.1.2 Insert – insert_il

As you may have already observed, adding an element at the end of an incomplete list doesn't require an additional output argument – the addition may be performed in the input structure.

To do that, we need to traverse the input list element by element and when the end of the list is found, simply modify that free variable such that it contains the new element. If the element is already in the list, don't add it:

```
insert_il(X, L):-var(L), !, L=[X|_]. %found end of list, add element
insert_il(X, [X|_]):-!. %found element, stop
insert_il(X, [_|T]):- insert_il(X, T). % traverse input list to reach end/X
```

Exercise 8.3: Trace the execution of the following queries:

```
?- L = [1, 2, 3|_], insert_il(3, L).
?- L = [1, 2, 3|_], insert_il(4, L).
?- L = [1, 2, 3|_], insert_il(X, L).
```

Notice how similar the two predicates are: insert_il and member_il are – the only thing that differs is what to do when reaching the end of the list.

Also, insert_il and member1 are very similar. Actually, if we compare their traces more closely, we find that they provide the same behavior! This means that testing for membership on complete lists is equivalent to inserting a new element in an incomplete list – i.e. we can remove the first clause of the insert_il predicate (*Explain!*).

8.1.3 Delete – delete_il

The predicate for deleting an element from an incomplete list is very similar to its counterpart for complete lists:

```
delete_il(_, L, L):-var(L), !. % reached end, stop
delete_il(X, [X|T], T):-!. % found element, remove it and stop
delete_il(X, [H|T], [H|R]):-delete_il(X, T, R). % search for the element
```

Again, notice how the stopping condition which corresponds to reaching the end of the input list is the first clause, and has the known form. Clauses 2 and 3 are the same as for the delete predicate.

Exercise 8.4: Trace the execution of the following queries:

```
?- L = [1, 2, 3|_], delete_il(2, L, R).
?- L = [1, 2, 3|_], delete_il(4, L, R).
?- L = [1, 2, 3|_], delete_il(X, L, R).
```

8.2 Incomplete binary search trees

Incomplete trees are another special type of incomplete structures – a branch no longer ends in nil, but in an unbound variable. *Examples 8.1.c* and *8.1.d* show instances of such structures.

The observations in section 8.1 for writing predicates on incomplete lists apply in the case of incomplete trees as well. Thus, in the following we shall apply those observations to develop the same predicates we discussed for lists in the previous section: search_it, insert_it, delete_it – for incomplete binary search trees.

8.2.1 Search – search_it

Just like in the case of lists, the predicate which searches for a key in a complete tree does not perform entirely well on incomplete trees – we need to explicitly add a clause for fail situations:

```
search_it(_, T):-var(T), !, fail.
search_it(Key, t(Key, _, _)):-!.
search_it(Key, t(K, L, R)):-Key<K, !, search_it(Key, L).
search_it(Key, t(_, _, R)):-search_it(Key, R).
```

Exercise 8.5: Trace the execution of the following queries:

```
?- T = t(7, t(5, t(3, _, _), t(6, _, _)), t(11, _, _)), search_it(6, T).
?- T = t(7, t(5, t(3, _, _), _), t(11, _, _)), search_it(9, T).
```

8.2.2 Insert – insert_it

Since insertion in a binary search tree is performed at the leaf level (i.e. at the end of the structure), we can insert a new key in an incomplete binary search tree, without needing an extra output argument. If we turn again to the analogy with incomplete lists, we find that here as well the predicate which performs search on the complete structure will act as an insert predicate on the incomplete structure (*Explain!*):

```
insert_it(Key, t(Key, _, _)):-!.
insert_it(Key, t(K, L, R)):-Key<K, !, insert_it(Key, L).
insert_it(Key, t(_, _, R)):- insert_it(Key, R).
```

Exercise 8.6: Trace the execution of the following queries:

```
?- T = t(7, t(5, t(3, _, _), t(6, _, _)), t(11, _, _)), insert_it(6, T).
?- T = t(7, t(5, t(3, _, _), _), t(11, _, _)), insert_it(9, T).
```

8.2.3 Delete – delete_it

Deleting an element from an incomplete binary search tree is very similar with the deletion from a complete binary search tree:

```
delete_it(Key, T, T):-var(T), !, write(Key), write(' not in tree\n').
delete_it(Key, t(Key, L, R), L):-var(R), !.
delete_it(Key, t(Key, L, R), R):-var(L), !.
delete_it(Key, t(Key, L, R), t(Pred, NL, R)):-!, get_pred(L, Pred, NL).
delete_it(Key, t(K, L, R), t(K, NL, R)):-Key<K, !, delete_it(Key, L, NL).
delete_it(Key, t(K, L, R), t(K, L, NR)):- delete_it(Key, R, NR).

get_pred(t(Pred, L, R), Pred, L):-var(R), !.
get_pred(t(Key, L, R), Pred, t(Key, L, NR)):-get_pred(R, Pred, NR).
```

Exercise 8.6: Trace the execution of the following queries:

```
?- T = t(7, t(4, t(3, _, _), t(6, t(5, _, _), _)), t(11, _, _)), delete_it(7, T, R).
?- T = t(7, t(4, t(3, _, _), t(6, t(5, _, _), _)), t(11, _, _)), delete_it(3, T, R).
?- T = t(7, t(4, t(3, _, _), t(6, t(5, _, _), _)), t(11, _, _)), delete_it(6, T, R).
?- T = t(7, t(5, t(3, _, _), _), t(11, _, _)), delete_it(9, T).
?- T = t(7, t(5, t(3, _, _), _), t(11, _, _)), nl, nl, write('Initial: '), write(T), nl, nl, write('Insert 10: '), insert_it(10, T), write(T), nl, nl, write('Delete 7: '), delete_it(7, T, R), write(R), nl, nl, write('Insert 7: '), insert_it(7, R), write(R), nl, nl.
```

8.3 Quiz exercises

8.3.1 Write a predicate which appends two incomplete lists (the result should be an incomplete

list also).

8.3.2 Write a predicate which reverses an incomplete list (the result should be an incomplete list also).

8.3.3 Write a predicate which transforms an incomplete list into a complete list.

8.3.4 Write a predicate which performs a preorder traversal on an incomplete tree, and collects the keys in an incomplete list.

8.3.5 Write a predicate which computes the height of an incomplete binary tree.

8.3.6 Write a predicate which transforms an incomplete tree into a complete tree.

8.4 Problems

8.4.1 Write a predicate which takes as input a deep incomplete list (i.e. any list, at any level, ends in a variable). Write a predicate which flattens such a structure.

```
?- flat_il([[1|_], 2, [3, [4, 5|_]|_]|_], R).
R = [1, 2, 3, 4, 5|_] ? ;
no
```

8.4.2 (**) Write an efficient predicate which computes the diameter of a binary incomplete tree:

diam(Root) = max{diam(Left), diam(Right), height(Left) + height(Right) + 2}).

What is the complexity of your algorithm?

8.4.3 (**) Write a predicate which determines if an incomplete list is sub-list in another incomplete list.

```
?- subl_il([1, 1, 2|_], [1, 2, 3, 1, 1, 3, 1, 1, 1, 2|_]).
yes
?- subl_il([1, 1, 2|_], [1, 2, 3, 1, 1, 3, 1, 1, 1, 3, 2|_]).
no
```

9. Difference Lists. Side Effects

A prolog list is accessed through its head and its tail. The setback of this way of viewing the list is that when we have to access the nth element, we must access all the elements before it first. If, for example, we need to add an element at the end of the list, we must go through all the elements in the list to reach that element.

```
add(X, [H|T], [H|R]) :- add(X, T, R).
add(X, [], [X]).
```

There is an alternative technique of representing lists in prolog that lets us access the end of a list easier. A difference list is represented by two parts, the start of the list S and the end of the list E:

S: [1,2,3,4]
E: [3,4]
S-E: [1,2]
S-E(the difference list) represents the list obtained by removing part E from part S:

There are no advantages when using difference lists like in the previous example, but when combined with the concepts of free variables and unification, difference lists become a powerful tool. For example, list [1,2] can be represented by the difference list [1,2|X]-X, where X is a free variable. We can write the add predicate with difference lists in the following way:

```
add(X,LS,LE,RS,RE):-RS=LS,LE=[X|RE].
```

We test it in Prolog by asking the following query:

```
?- LS=[1,2,3,4|LE],add(5,LS,LE,RS,RE).
LE = [5|RE],
LS = [1,2,3,4,5|RE],
RS = [1,2,3,4,5|RE] ?
yes
```

To better understand the way the add predicate works, we can imagine the list is represented by two “pointers”, one pointing to the start of the list (LS) and the second one to the end of the list (LE), a variable without an assigned value.

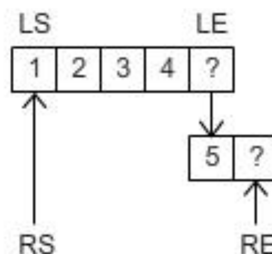


Figure 9.1 – Adding an element at the end of a difference list

The result is also represented by two “pointers”. The resulting list will be the input list with the new element inserted at the end. The beginning of the input and result lists is the same so we can unify the result list start variable with the input list start variable (RS=LS).

The result list must end, just like the input list, in a variable (RE), but we must, somehow, modify the input list to add the new element at the end. Because the end of the input list is a free variable, we can unify it with the list beginning with the new element followed by a new variable, the new end of the list (LE=[X|RE]). After the predicate finished execution we can see that the input list LS and result list RS have the same values, but the end of the input list is no longer a variable (LE=[5|RE]).

9.1 Tree traversal

The tree traversal predicates are used to extract the elements in a tree to a list in a specific order. The computation intensive part of these predicates is not the traversing itself but the combination of the result lists to obtain the final result. Although hidden from us, prolog will go through the same elements of a list many times to form the result list. We can save a lot of work by changing regular lists to difference lists.

The inorder predicate using a regular list to store the result:

```
inorder(t(K,L,R),List):-  inorder(L,ListL),
                           inorder(R,ListR),
                          append1(ListL,[K|ListR],List).
inorder (nil,[]).
```

By executing a query trace on the inorder predicate we can easily observe the amount of work performed by the append predicate. It is also visible that the append predicate will access the same elements in the result list more than once as the intermediary results are appended to obtain the final result:

```
[. . .]
 8     3 Exit: inorder(t(5,nil,nil),[5]) ?
12      3 Call: append1([2],[4,5],_1594) ?
13      4 Call: append1([],[4,5],_10465) ?
13      4 Exit: append1([],[4,5],[4,5]) ?
12      3 Exit: append1([2],[4,5],[2,4,5]) ?
[. . .]
22      2 Call: append1([2,4,5],[6,7,9],_440) ?
23      3 Call: append1([4,5],[6,7,9],_20633) ?
24      4 Call: append1([5],[6,7,9],_21109) ?
25      5 Call: append1([],[6,7,9],_21585) ?
25      5 Exit: append1([],[6,7,9],[6,7,9]) ?
24      4 Exit: append1([5],[6,7,9],[5,6,7,9]) ?
23      3 Exit: append1([4,5],[6,7,9],[4,5,6,7,9]) ?
22      2 Exit: append1([2,4,5],[6,7,9],[2,4,5,6,7,9]) ?
[. . .]
```

We can improve the efficiency of the inorder predicate by rewriting it using difference lists. The inorder_dl predicate has 3 parameters: the tree node it is currently processing, the start of the result list and the end of the result list:

```
/* when we reached the end of the tree we unify the beggining and end of the partial result list – representing an empty list as a difference list */
inorder_dl(nil,L,L).
inorder_dl(t(K,L,R),LS,LE):-
```

```
/* obtain the start and end of the lists for the left and right subtrees */
inorder_dl(L,LSL,LEL),
inorder_dl(R,LSR,LER),
/* the start of the result list is the start of the left subtree list */
LS=LSL,
/* insert the key between the end of the left subtree list and start of the right subtree list */
LEL=[K|LSR],
/* the end of the result list is the end of the right subtree list */
LE=LER.
```

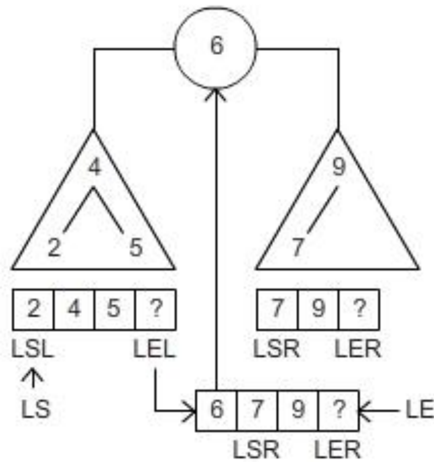


Figure 9.2 – Appending two difference lists

The predicate can be simplified by replacing the explicit unifications with implicit unifications:

```
inorder_dl(nil,L,L).
inorder_dl(t(K,L,R),LS,LE):-inorder_dl(L,LS,[K|LT]), inorder_dl(R,LT,LE).
```

Exercise 9.1: Study the execution of the following queries:

```
?- tree1(T), inorder_dl(T,L,[]).
?- tree1(T), inorder_dl(T,L,_).
```

Exercise 9.2: Implement the preorder_dl *tree traversal predicate using difference lists.*

Exercise 9.3: Implement the postorder_dl *tree traversal predicate using difference lists.*

9.2 Sorting – quicksort

Remember the *quicksort* algorithm (and predicate): the input sequence is divided in two parts – the sequence of elements smaller or equal to the pivot and the sequence of elements larger than the pivot; the procedure is called recursively on each partition, and the resulting sorted sequences are appended together with the pivot to generate the sorted sequence:

```
quicksort([H|T], R):-
        partition(H, T, Sm, Lg),
```

```
        quicksort(Sm, SmS),
        quicksort(Lg, LgS),
        append(SmS, [H|LgS], R).
quicksort([], []).
```

Just as for the inorder predicate, the quicksort predicate will waste a lot of execution time to append the results of the recursive calls. To avoid this we can apply difference lists again:

```
quicksort_dl([H|T],S,E):-
        partition(H,T,Sm,Lg),
        quicksort_dl(Sm,S,[H|L]),
        quicksort_dl(Lg,L,E).
quicksort_dl([],L,L).
```

The partition predicate is the same as the one for the old quicksort predicate, its purpose being to divide the list in two by comparing each element with the pivot. All elements in the list must be accessed for this operation so we cannot improve the performance for the partition predicate.

The quicksort predicate works in the same way as before: divides the list in elements larger and smaller than the pivot element and applies quicksort recursively on the each partition. The difference from the original version is the result list, represented by two elements, the start and the end of the list, and, consequently, the way in which the results of the two recursive calls are put together with the pivot (figure 9.3 below).

Exercise 9.4: Study the execution of the following queries:

```
?- quicksort([4,2,5,1,3],L,[]).
?- quicksort([4,2,5,1,3],L,_).
```

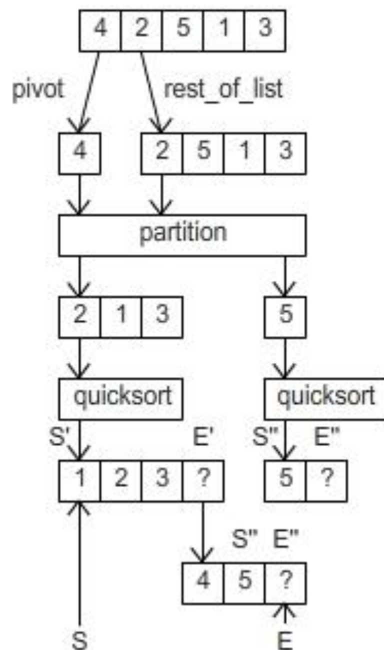


Figure 9.3 – Quicksort with difference lists

9.3 Side Effects

Side effects refer to a series of Prolog predicates which allow the dynamic manipulation of the predicate base:

- assert/1 (or assertz/1) - adds the predicate clause given as argument as last clause
- asserta/1 - adds the predicate clause given as argument as first clause
- retract/1 - tries to unify the argument with the first unifying fact or clause in the database. The matching fact or clause is then removed from the database

Predicates which can be manipulated via assert/retract statements at run-time are called dynamic predicates, as opposed to the static predicates that we have seen so far. Dynamic predicates should be declared as such. However, when the interpreter sees an assert statement on a new predicate, it implicitly declares it as dynamic.

The important aspects you need to know when working with side effects:

- Their **effect is maintained in the presence of backtracking**: i.e. once a clause has been asserted, it remains on the predicate base until it is explicitly retracted, even if the node corresponding to the assert call is deleted from the execution tree (e.g. because of backtracking).
- **assert - always succeeds; doesn't backtrack**
- **retract - may fail; backtracks;** retract respects the *logical update view* - it succeeds for all clauses that match the argument when the predicate was ***called***. *Exercise 9.5:* T*o understand how it works, try to execute the following query*:

```
?- assert(insect(ant)),
assert(insect(bee)),
(retract(insect(I)),
   writeln(I),
    retract(insect(II)),
   fail
   ; true
 ).
```

  You have probably observed that this query will output:

```
ant
bee
true
```

  This is because even if the second call to retract will also delete the clause insect(bee), when backtracking reaches the first retract call, the clause is still present in its logical view - it doesn't see that the clause has been deleted by the second retract call.

Prolog also has a retractall/1 predicate, with the following behavior: it deletes all predicate clauses matching the argument. In some versions of Prolog, retractall may fail if there is nothing to retract. To get around this, you may choose to assert a dummy clause of the right type. In SWI Prolog, however, retractall succeeds even for a call with no matching facts/rules.

Dynamic database manipulation via assert/retract can be used for storing computation results, such that they are not destroyed by backtracking. Therefore, if the same question is asked in the future, the answer is retrieved without having to recompute it. This technique is called memoisation, or caching, and in some applications it can greatly increase efficiency. However, side effects can also be used to change the behaviour of predicates at run-time (meta-programming). This generally leads to dirty, difficult to understand code. In the presence of heavy backtracking, it gets even worse. Therefore, this non-declarative feature of Prolog should be used with caution.

An example of memoisation with side effects is the following predicate which computes the nth number in the *fibonacci sequence* (you have already seen the less efficient version in the second lab session):

```
:-dynamic memo_fib/2

fib(N,F):-memo_fib(N,F),!.
fib(N,F):- N>1,
        N1 is N-1,
        N2 is N-2,
        fib(N1,F1),
        fib(N2,F2),
        F is F1+F2,
        assertz(memo_fib(N,F)).
fib(0,1).
fib(1,1).
```

Exercise 9.6: Consult the predicate specification above, and run the following queries, sequentially:

```
?- listing(memo_fib/2).
?- fib(4,F).
?-listing(memo_fib/2).
?-fib(10,F).
?-listing(memo_fib/2).
?-fib(10,F).
```

What do you notice?

9.3.1 Failure-driven loops

Whenever you want to collect all the answers that you have stored on your predicate base via assert statements, you can use failure driven loops: *force Prolog to backtrack* until there are no more possibilities left. The pattern for a failure driven loop which reads all stored clauses - say for predicate memo_fib/2 above - and prints all the *fibonacci numbers* already computed:

```
print_all:-memo_fib(N,F),
        write(N),
        write(‘ - ‘),
        write(F),
        nl,
```

```
        fail.
print_all.
```

Exercise 9.7: Study the execution of the following queries:

```
?-print_all.
?-retractall(memo_fib(_,_)).
?-print_all.
```

Question 9.1: Can you collect all the values in a list, instead of writing them on the screen? How? Can you do that without modifying the predicate base? (After you answer these questions, search for findall/3 *in the SWI manual).*

Exercise 9.7: What are the answers you should get on the following queries (try to solve them on paper first, then check the answers in the Prolog engine):

```
?- findall(X, append(X,_,[1,2,3,4]),List).
?- findall(lists(X,Y), append(X,Y,[1,2,3,4]), List).
?-findall(X, member(X,[1,2,3]),List).
```

Let's now see an example of using *side effects* to get all the possible answers to a query: let's write a predicate which computes all the permutations of a list, and returns them in a separate list. We want the query on the predicate to behave like this:

```
?- all_perm([1,2,3],L).
L=[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]];
no.
```

Assuming that you already know how to implement the perm/2 predicate - the predicate which generates a permutation of the input list (see the document on *Sorting Methods* if not) - the all_perm/2 predicate specification is:

```
all_perm(L,_):-perm(L,L1),
               assertz(p(L1)),
               fail.
all_perm(_,R):-collect_perms(R).

collect_perms([L1|R]):-retract(p(L1)),
                       !,
                       collect_perms(R).
collect_perms([]).
```

Exercise 9.8: Study the execution of the following queries:

```
?-retractall(p(_)), all_perm([1,2],R).
?-listing(p/1)).
?-retractall(p(_)),all_perm([1,2,3],R).
```

Questions:

9.2: Why do I need a retractall call before calling all_perm/2?

9.3: Why do I need a ! after the retract call in the first clause of collect_perms/1*?*

9.4: What kind of recursion is used on collect_perms/1*? Can you do the collect using the other type of recursion? Which is the order of the permutations in that case?*

9.5: Does collect_perms/1 *destroy the results stored on the predicate base, or does it only read them?*

9.4 Quiz exercises

9.4.1 Write a predicate which transforms an incomplete list into a difference list (and one which makes the opposite transformation).

9.4.2. Write a predicate which transforms a complete list into a difference list (and one which makes the opposite transformation).

9.4.3. Write a predicate which generates a list with all the possible decompositions of a list into 2 lists, without using findall. Example query:

```
?- all_decompositions([1,2,3], List).

List = [ [[], [1,2,3]], [[1], [2,3]], [[1,2], [3]], [[1,2,3], []] ];
no.
```

9.5 Problems

9.5.1 Write a predicate which flattens a deep list using difference lists instead of append.

9.5.2 Write a predicate which collects all even keys in a binary tree, using difference lists.

9.5.3. Write a predicate which collects, from a binary **incomplete** search tree, all keys between K1 and K2, using difference lists.

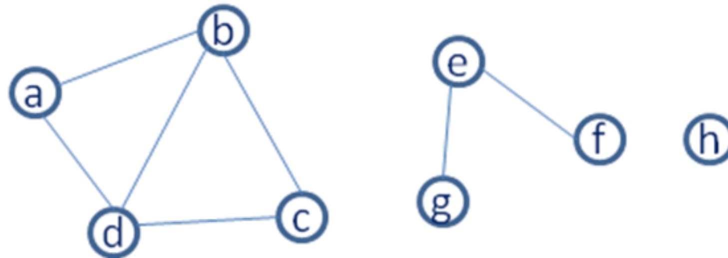
10. Graphs. Paths in Graphs

This session covers graph representation alternatives and several types of graph paths in Prolog.

10.1 Representation

A graph is given by a set of vertices and a set of edges (if the graph is undirected, or arcs, if the graph is directed): *G=(V,E)*

Let us consider an example of an undirected graph:



Several alternatives are available for representing graphs in Prolog, and they can be categorized according to the following criteria:

A. Representation type:
   1. As a collection of edges
   2. As a collection of vertices and associated neighbor list

B. Where you store the graph:
   1. In the main memory, as a data object
   2. In the predicate base, as a set of predicate facts

Consequently, four main representations are possible (other approaches may exist, but for the purpose of the current course these are enough):

- (A1B2) As a **set of edges**, stored as **predicate facts** (*edge-clause* form):
```
edge(a, b).
edge(b, a).
edge(b, c).
edge(c, b).
….
```

In this representation, isolated nodes have to be specified as having an edge between them and nil: edge(f, nil). If your graph is undirected, you may write a predicate such as the following (to avoid having to write the edges in both directions):

```
is_edge(X,Y):- edge(X,Y); edge(Y,X).
```

- (A2B2) As a **set of vertices and associated neighbor lists**, stored as **predicate facts** (*neighbor list-clause* form):
```
neighbor(a,[b,d]).
neighbor(b, [a, c, d]).
neighbor(c, [b, d]).
```

…

- (A2B1) As a **set of vertices and associated neighbor lists**, stored as a **data object** (*neighbor list-list* form):
  ?- Graph = [n(a, [b,d]), n(b, [a,c,d]), n(c, [b,d]), n(d, [a,b,c]), n(e, [f,g]), n(f, [e]), n(g, [e]), n(h, [])].

- (A1B1) As **the set of vertices and the set of edges,** stored as a **data object** (*graph-term* form):
  ?- Graph = graph([a,b,c,d,e,f,g,h], [e(a,b), e(b,a), …]).

The most suitable representation to use is highly dependent on the problem at hand. Therefore, it is convenient to know how to perform conversions between different graph representations. Here, we provide an example conversion from the *neighbor list-clause* form to the *edge-clause* form:

```
neighbor(a, [b, d]).        % an example graph – 1st connected component of the
neighbor(b, [a, c, d]).     % example graph
neighbor(c, [b, d]).

neighb_to_edge:-neighbor(Node,List),
                process(Node,List),
                fail.
neighb_to_edge.

process(Node, [H|T]):- assertz(edge(Node, H)),
                       process(Node, T).
process(_, []).
```

The graph is initially stored in the predicate database. Predicate neighb_to_edge reads one clause of predicate neighbor at a time, and processes the information in each clause separately; process traverses the neighbor list of the current node, and asserts a new fact for predicate edge for each new neighbor of the current node.

10.2 Paths in Graphs

Having covered the issue of graph representation, let us address the graph traversal problem. We shall start with the simple path between two nodes, and progress to the restricted path between two nodes, the optimal path between two nodes and, finally, the Hamiltonian cycle of a graph.

10.2.1 Simple path

We assume the graph is represented in the edge-clause form. A predicate which searches for a path between two nodes in a graph is presented below:

```
% path(Source, Target, Path)

path(X,Y,Path):-path(X,Y,[X],Path).

path(X,Y,PPath, FPath):- is_edge(X,Z),
```

```
                    \+(member(Z, PPath)),
                    path(Z, Y, [Z|PPath], FPath).
path(X,X,PPath, PPath).
```

What type of recursion is used here?

Exercise 10.1: Represent a graph using the edge-clause form and trace the execution of the path predicate on different queries (you may use the example graph, perhaps add some edges to it). What happens when you repeat the question?

10.2.2 Restricted path

Let us now try to write a predicate which searches for a restricted path between two nodes in a graph, i.e. the path must pass through certain nodes, in a certain order (these nodes are specified in a list).

```
% restricted_path(Source, Target, RestrictionsList, Path)
% check_restrictions(RestrictionsList, Path)

restricted_path(X,Y,LR,P):- path(X,Y,P),
                        check_restrictions(LR, P).

check_restrictions([],_):- !.
check_restrictions([H|T], [H|R]):- !, check_restrictions(T,R).
check_restrictions(T, [H|L]):-check_restrictions(T,L).
```

The predicate restricted_path/4 searches for a path between the source and the destination nodes, and then checks if that path satisfies the restrictions specified in LR (i.e. passes through a certain sequence of nodes, specified in LR), using predicate check_restrictions/2, performs the actual check. check_restrictions/2 traverses the restrictions list (the first argument) and the list representing the path (the second argument) simultaneously, so long as their heads coincide (clause 2). When the heads do not match, we advance in the second list only (clause 3). The predicate succeeds when the first list becomes empty (clause 1).

Question: What happens if we move the stopping condition as last clause? Do we need the "!" in the stopping condition?

Exercise 10.2: Trace the execution of the following queries:

1. ?- check_restrictions([2,3], [1,2,3,4]).
2. ?- check_restrictions([1,3], [1,2,3,4]).
3. ?- check_restrictions([1,3], [1,2]).

Exercise 10.3: Trace the execution of several queries for the restricted_path/4 predicate, on your example graph. Which is the order in which you have to specify the nodes in the list of restrictions? Why?

10.2.3 Optimal path

We consider the optimal path between the source and the target node in a graph as the path containing the minimum number of nodes. One approach to find the optimal path is to generate all paths via backtracking and then select the optimal path. Of course, this is extremely

inefficient. Instead, during the backtracking process, we will keep the partial optimal solution using lateral effects (i.e. in the predicate base), and update it whenever a better solution is found:

```
%optimal_path(Source, Target, Path)

:- dynamic sol_part/2.

optimal_path(X,Y,_):-asserta(sol_part([],100)),
                     path(X,Y,[X],1).
optimal_path(_,_,Path):-retract(sol_part(Path,_)).

path(X,X,Path,LPath):-retract(sol_part(_,_)),!,
                     asserta(sol_part(Path,LPath)),
                     fail.
path(X,Y,PPath,LPath):-is_edge(X,Z),
                     \+(member(Z,PPath)),
                     LPath1 is LPath+1,
                     sol_part(_,Lopt),
                     LPath1<Lopt,
                     path(Z,Y,[Z|PPath],LPath1).
```

The predicate path/4 generates, via backtracking, all paths which are better than the current partial solution, and updates the current partial solution whenever a shorter path is found. Once a better solution than the current optimal solution is found, the predicate replaces the old optimal in the predicate base (clause 1) and then continues the search, by launching the backtracking mechanism (using fail).

Exercise 10.3: Trace the execution of several queries for the optimal_path/3 predicate, using the example graph.

!!! When working with assert/retract, make sure you "clean after yourselves", i.e. check that no unwanted clauses remain asserted on your predicate base after the execution of your queries (their effects are not affected by backtracking!).

10.2.4 Hamiltonian Cycle

A Hamiltonian cycle is a closed path in a graph which passes exactly once through all nodes (except for the first node, which is the source and the target of the path). Of course, not all graphs possess such a cycle. The predicate hamilton/3 is provided below:

```
%hamilton(NbNodes, Source, Path)

hamilton(NN, X, Path):- NN1 is NN-1, hamilton_path(NN1,X, X, [X],Path).
```

The predicate hamilton_path/5 is left for you to implement. The predicate should search for a closed path from X, of length NN1 (number of nodes in the graph, minus 1).

Exercise 10.3: Trace the execution of several queries for the hamilton/3 predicate, using the example graph.

10.3 Quiz exercises

10.3.1 Write the predicate(s) which perform the conversion between the *edge-clause* representation (A1B2) to the *neighbor list-list* representation (A2B1).

10.3.2 The restricted_path predicate computes a path between the source and the destination node, and then checks whether the path found contains the nodes in the restriction list. Since predicate path used forward recursion, the order of the nodes must be inversed in both lists – path and restrictions list. Try to motivate why this strategy is not efficient (use trace to see what happens). Write a more efficient predicate which searches for the restricted path between a source and a target node.

10.3.3 Rewrite the optimal_path/3 predicate such that it operates on weighted graphs: attach a weight to each edge on the graph and compute the minimum cost path from a source node to a destination node.

10.4 Problems

10.4.1 Write a predicate cycle(A,P) to find a closed path (cycle) P starting at a given node A in the graph G (use any graph representation for G). The predicate should return all cycles via backtracking.

10.4.2 (**) Write a set of Prolog predicates to solve the Wolf-Goat-Cabbage problem: "A farmer and his goat, wolf, and cabbage are on the North bank of a river. They need to cross to the South bank. They have a boat, with a capacity of two; the farmer is the only one that can row. If the goat and the cabbage are left alone without the farmer, the goat will eat the cabbage. Similarly, if the wolf and the goat are together without the farmer, the goat will be eaten."
Hints:

- you may choose to encode the state space as instances of the configuration of the 4 objects (Farmer, Wolf, Goat, Cabbage), represented either as a list (i.e. [F,W,G,C]), or as a complex structure(e.g. F-W-G-C, or state(F,W,G,C)).
- the initial state would be [n,n,n,n], the final state [s,s,s,s], for the list representation of states (e.g. if Farmer takes Wolf across -> [s,s,n,n] (and the goat eats the cabbage), so this state should not be valid)
- in each transition (or move), the Farmer can change its state (from n to s, or vice-versa) together with at most one other participant (Wolf, Goat, or Cabbage)
- this can be viewed as a path search problem in a graph Enjoy!

11. Graphs Search Algorithms

This session covers graph search algorithms in Prolog: depth-first search, breadth-first search and best-first search.

11.1 Depth-first search

As in the previous session, we shall employ the *edge-clause* form for graph representation. Since *depth-first search* is the mechanism employed by the Prolog engine, all path predicates from the previous session employed a DFS search strategy. Below you have the predicates which implement a DFS search from a source node (it explores the connected component of the source node):

```
%d_search(Start, Path)

d_search(X,_):- df_search(X,_).
d_search(_,L):- collect_v([],L).

df_search(X,L):-
                asserta(vert(X)),
                edge(X,Y),
                \+(vert(Y)),
                df_search(Y,L).

collect_v(L,P):-retract(vert(X)),!, collect_v([X|L],P).
collect_v(L,L).
```

Exercise 11.1: Trace the execution of the d_search/2 predicate on an example graph.

11.2 Breadth-first search

The *BFS* strategy employs a queue to keep track of the expansion order of the nodes. In each step, a new node is read from the queue and is expanded – i.e. all its unvisited neighbors are added to the queue, in turn. Below you may find the specification for the necessary predicates:

```
%do_bfs(Start, Path)

do_bfs(X, Path):-assertz(q(X)), asserta(vert(X)), bfs(Path).

bfs(Path):- q(X), !, expand(X), bfs(Path).
bfs(Path):- assertz(vert(end)), collect_v([],Path).
expand(X):-     edge(X,Y),
                \+(vert(Y)),
                asserta(vert(Y)),
                assertz(q(Y)),
                fail.
expand(X):-retract(q(X)).
```

Exercise 11.2: Trace the execution of the do_bfs/2 predicate on an example graph.

11.3 Best-first search

The *Best-first search* algorithm is an informed greedy search strategy, in that it employs a heuristic to estimate the cost of the path from the current node to the target node. In each step, the algorithm selects the node having the smallest estimated distance to the target node (via the heuristic function).

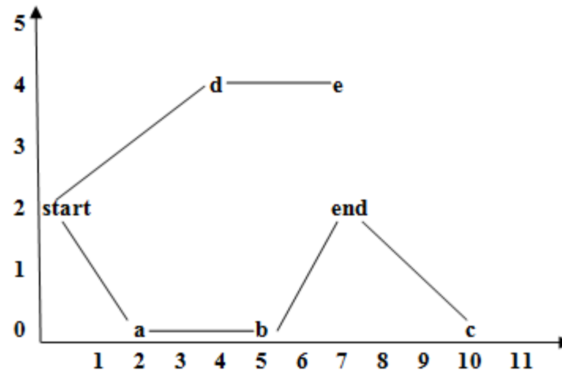


Figure 11.1: An example graph for the best-first search algorithm

The graph above can be represented using a variation of the *neighbor list-clause* form, as:

```
pos_vec(start,0,2,[a,d]).
pos_vec(a,2,0,[start,b]).
pos_vec(b,5,0,[a,c,tinta]).
pos_vec(c,10,0,[b,tinta]).
pos_vec(d,3,4,[start,e]).
pos_vec(e,7,4,[d]).
pos_vec(tinta,7,2,[b,c]).

is_target(end).
```

The end node is specified as being the target node, using a predicate clause. The predicate specifications are presented below:

```
dist(Nod1,Nod2,Dist):-pos_vec(Nod1,X1,Y1,_),pos_vec(Nod2,X2,Y2,_),
                        Dist is (X1-X2)*(X1-X2)+(Y1-Y2)*(Y1-Y2).

order([Nod1|_],[Nod2|_]):- is_target(Target),
                        dist(Nod1,Target,Dist1),
                        dist(Nod2,Target,Dist2),
                        Dist1<Dist2.

best([],[]):-!.
best([[Target|Rest]|_],[Target|Rest]):-is_target (Target),!.
best([[H|T]|Rest],Best):-pos_vec(H,_,_,Vec),
                        expand(Vec,[H|T],Rest,Exp),
                        q(Exp,SortExp,[]),
                        best(SortExp,Best).
```

```
expand([],_,Exp,Exp):-!.
expand([E|R],Cale,Rest,Exp):-\+(member(E,Cale)),!,
                              expand(R,Cale,[[E|Cale]|Rest],Exp).
expand([_|R],Cale,Rest,Exp):-expand(R,Cale,Rest,Exp).

partition(H,[A|X],[A|Y],Z):-
                    order(A,H),!,partition(H,X,Y,Z).
partition(H,[A|X],Y,[A|Z]):-partition(H,X,Y,Z).
partition(_,[],[],[]).

q([H|T],S,R):-
          partition(H,T,A,B),
          q(A,S,[H|Y]),
          q(B,Y,R).
          q([],S,S).
```

Exercise 11.3: Trace the execution of the following query:
?- best([[start]], Best).

11.3 Quiz exercises

11.3.1 Write a predicate which perform DLS – Depth-Limited Search on a graph. Set the depth limit via a predicate (e.g. depth_max(2).).

Bibliography

[1] Bratko, I., “Prolog Programming for Artificial Intelligence”, Addison-Wesley Publishing, 1986
[2] Muresan, T., Potolea, R., Todoran, E. and Suciu, A., “Programare Logica. Indrumator de laborator”, UTPress, 1998
[3] Potolea, R., “Programare Logica”, UTPress, 2007