1. Logic Programming and Prolog: A First Glimpse

We shall learn to write computer programs in a non-procedural way. The primary objective of this is to open our mind to a new way of looking at computing.

Procedural programming basically views the computer as a device characterized by a large uniform store of memory cells and a processing unit which is able to access the memory cells and to perform - in a strictly sequential way - some arithmetic and logical operations on their contents. A program for such a *von Neumann* machine is a sequence of instructions to perform such operations, and an additional set of control instructions (If, While, Goto, ...) which can affect the next instruction to be executed, possibly depending on the content of some memory cell. Procedural programs describe in detail, **how** a given set of variables (= memory cells) should be manipulated in order to get a solution of the given problem.

On the other hand, *declarative (or definitional or non-procedural) programming* expresses the *knowledge* about the problem and formulates the *objective* (or the *goal*). From a complete description of the *facts* and *rules* of a problem situation, the computer should then be able to find the solution using logical deduction. Ideally, a declarative program specifies **what** is to be solved and **not how** it should be done.

A simple example may explain the difference of the two programming paradigms: the threading of a maze. The problem is to find a way from a given starting point to a given end point. In a *procedural program* the maze is typically represented as a data structure, and an algorithm specifies step-by-step how to try the different possible paths. In a *declarative program*, on the other hand, the maze and the possible moves define "the rules of the game". The computer is told to "play the game", according to these rules, and with the objective of finding a path from the starting point to the end point.

There are several different approaches to declarative programming. The two most important ones are *logic programming* and *functional programming*. Typical *functional programming languages* are Lisp and (in some sense) Mathematica.

The main idea behind *logic programming* is that *deduction* (i.e. the construction of logical consequences) can be viewed as a form of *computation*, and that the declarative statement of the form

P if Q and R and S

can also be interpreted procedurally as:

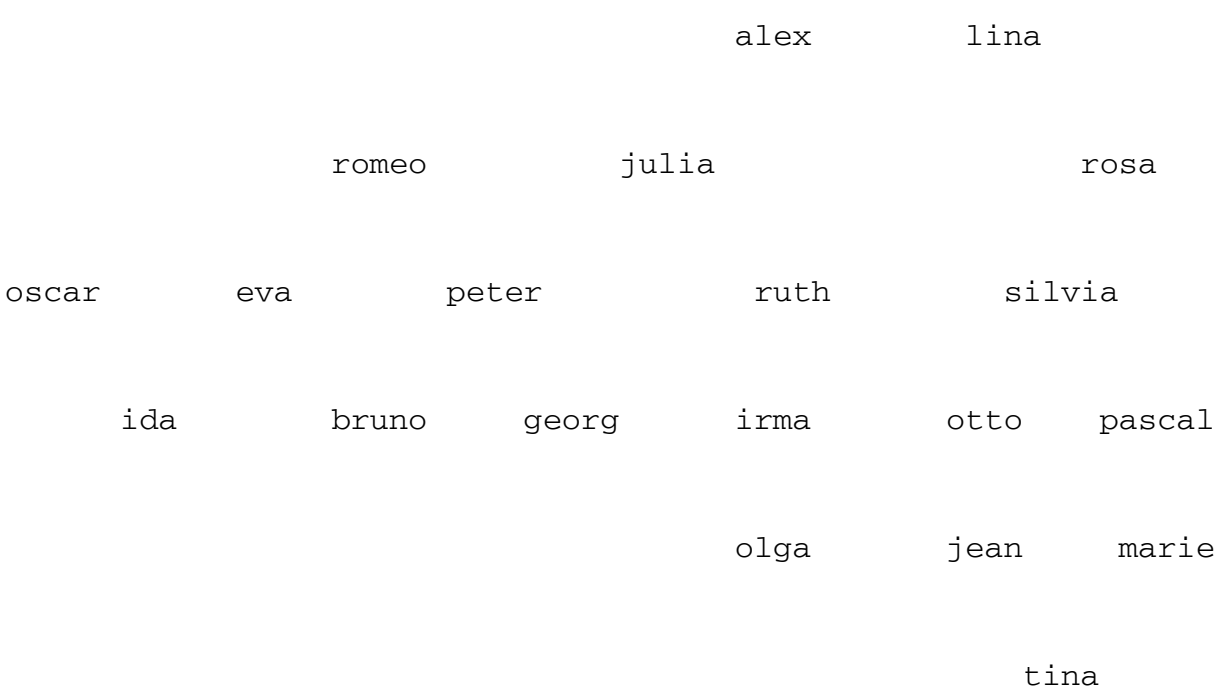
To solve P, solve Q and R and S.

We shall start with some simple examples of logic (and Prolog) programs and return to some more theoretical considerations later.

First Steps: Facts, Queries, and Rules

A *logic database* is comprised of a set of *facts* and *rules*. Facts can define relations just as in relational databases.

We start with a hypothetical family which we define using two relations between persons, *parent-child* and *husband-wife*, and the attribute *male*. We shall then declare rules for other relationships.




```
% parent(X,Y) means: X is the
% father or the mother of Y

parent(alex,julia).
parent(alex,rosa).
parent(lina,julia).
parent(lina,rosa).
parent(romeo,peter).
parent(julia,peter).
parent(rosa,silvia).
parent(oscar,ida).
parent(eva,ida).
parent(eva,bruno).
parent(peter,bruno).
parent(peter,georg).
parent(peter,irma).
parent(ruth,georg).
parent(ruth,irma).
parent(silvia,otto).
parent(silvia,pascal).
parent(irma,olga).
parent(irma,jean).
parent(otto,olga).
parent(otto,jean).
parent(jean,tina).
parent(marie,tina).
```

```
% male(X) means:
% X is a man

male(alex).
male(romeo).
male(oscar).
male(peter).
male(bruno).
male(georg).
male(otto).
male(pascal).
male(jean).

% husband(X,Y) means:
% X is the husband of Y

husband(alex,lina).
husband(romeo,julia).
husband(oscar,eva).
husband(peter,ruth).
husband(otto,irma).
husband(jean,marie).
```

The above family database is stored - one *fact* per line - in a source-text file named `family.pl`. It can be imported (consulted) into the Prolog interpreter with the following command:

```
?-
```

We can then ask questions (*queries*) about the family, like the following:

```
?- parent(eva,bruno).


?- parent(georg,otto).


?- parent(silvia,X).




?- parent(Y,ida).



?- husband(U,V).



?- husband(_,irma).


?- husband(M,_).
```

Note that *facts* and *queries* are syntactically identical: they are both *terms*. They are distinguished by their context only.

Queries can always be interpreted as *goals*: Instead of asking "Is there an X such that *parent(silvia,X)* is true?" we can also put it imperatively: "Try to find an X such that you can prove *parent(silvia,X)*!". Therefore, we shall use the terms "query" and "goal" interchangeably.

The following is called a *conjunctive goal*:

```
?- parent(eva,X), male(X).
```

In logic programs (and also in goals), the comma (**,**) denotes logical **and**. In the above conjunctive goal, X is a *shared variable*. The query reads: "Is there an X such that both *parent(eva,X)* **and** *male(X)*?". Evidently, this is the case for X=*bruno*, the son of *eva*. Let us consider some other conjunctive goals:

```
?- parent(X,Y), parent(Y,peter).
```

```
?- parent(W,Z), husband(otto,Z).
```

```
?- parent(X,Y), male(X).
```

The last query appparently returns all pairs of persons (X,Y) for which X is the father of Y. This brings us to the third and most important statement class in logic programming, the *rules*. A rule is a statement of the form:

A :- *B1, B2, ... Bn.*

where n ≥ 0. *A* is the *head* of the rule, and the *Bi*'s are its *body*.

A rule expressing the *father* relationship is:

Similarly, the *grandfather* relationship can be described as:

Rules can be viewed in two different ways:

(1) Procedural reading: To answer the query *"Is X the father of Y?"*, answer the conjunctive query *"Is X a parent of Y and is X male?"*.

(2) Logical implication: X is the father of Y **if** X is a parent of Y and X is male.

The second interpretation explains why we shall read the symbol **:-** as **if**.

Let us develop some more rules:

```
% brother(X,Y) :- X is the brother of Y.


% uncle(X,Y) :- X is the uncle of Y.


% female(X) :- X is a female person.


% sister(X,Y) :- X is the sister of Y.


% has_son(X) :- the person X has a son.


% married(X,Y) :- X and Y are married (to each other).




% brother_in_law(X,Y) :- X is the brother-in-law of Y.
```

Facts and rules are collectively called *Horn clauses*, or *clauses*, for short. It should be noted that a fact is just a special case of a rule, the case where n = 0. If a relation, such as *parent* or *brother_in_law*, is declared with more than one clause, these clauses are to be considered as *alternatives*.

The following relations are even more interesting and some of them give an idea of more advanced concepts of logic programming.

```
% ancestor(X,Y) :- X is an ancestor of Y.
```

```
% relatives(X,Y) :- X and Y are relatives.
```

```
% ancestors(As,X) :- As is the set of all ancestors of X.
```

```
% descendants(Ds,X) :- Ds is the set of all descendants of X
```

Exercises

(1.1) Develop rules for the following relations and attributes: *mother, niece, female_cousin, mother_in_law, daughter_in_law, is_grandmother, has_both_parents* and implement them as Prolog programs. Experiment with different forms of queries.

(1.2) Modify the rules for *brother* and *sister* so that they only recognize *full siblings*, i.e. those that have the same (known) mother and father.

(1.3) The relation *teaching_plan* is intended to describe the teaching situation at our college. Some typical facts in this relation could be: *teaching_plan(a3,bregy,english), teaching_plan(i3t,hett,theoinfo),* or *teaching_plan(i2w,businger,mathematics).* A second relation, *division*, describes which classes belong to which division, e.g. *division(a3,auto), division(i3w,info).* Write queries with the following meanings: (a) Does *businger* teach at the *auto* division? (b) Who teaches *mathematics* at the *info* division. (c) What division does *bregy* teach *english* at? (d) What subjects are taught at the *electro* division? (e) Who teaches in more than one division? (f) Who should take English lessons?

Although Prolog has not been invented primarily as a programming language for doing arithmetics, it can be used to do mathematics just like any other programming language. The next example, taken from recreational mathematics, should give a first impression on how to achieve this. Details shall be treated later in the course.

Exercise (1.4) Try to solve the following puzzle.

Summary

- Logic programming consists of defining *relations* and querying about relations.
- A program consists of *facts* and *rules*, which we collectively call *clauses.* A set of facts and rules about the same relation is called a *predicate*.
- Querying about relations, by means of *goals*, resembles querying a database. Prolog's answer to a query consists of a set of objects that satisfy the question (query).
- In Prolog, to establish whether an object satisfies a query is often a complicated process that involves logical inference and exploring among alternatives. All this is done automatically by the Prolog system and is, in principle, hidden from the user.
- Two types of interpretation (or meaning) of Prolog programs are distinguished: *declarative* and *procedural*.
- The following concepts have been discussed: Clause, fact, rule, query, goal, head and body of a rule, recursive rule, predicate, variable, a goal succeeds or fails.
- Special built-in predicates allow for arithmetic calculations, with a notation similar to the one of well-established procedural programming languages (e.g. Pascal). Arithmetic assignment is achieved by the *is* predicate.

2. Syntax and Meaning of Prolog Programs

The *syntax* of a programming language defines whether a given sequence of symbols is a legal sentence in the language. Prolog programs are built from *terms*. Each term is written as a sequence of characters. We distinguish between four categories of characters:

upper-case letters: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

lower-case letters: a b c d e f g h i j k l m n o p q r s t u v w x y z

digits: 0 1 2 3 4 5 6 7 8 9

sign characters: + - * / \ ^ < > ~ : . ? @ # $ &

There are actually more sign characters than shown in the fourth row, but others have special meanings and shall be introduced later.

Prolog is *case-sensitive*, i.e *zebra, Zebra, zeBra*, and *ZEBRA* are four different names.

As already mentioned, almost everything in Prolog is a *term*: We already know facts, rules, and queries.

A *term* is either a *constant*, a *variable*, or a *compound term*. We can represent this as a syntax diagram as follows:

Constants are thought of naming *specific objects* or *specific relationships*. A *constant* is either an *atom* or a *number*.

We already know names for objects and relationships, such as `alex` or `parent`. Some special symbols, such as `:-` or `\=` are also atoms. There are two kinds of atoms: those made of letters and digits, and those made up from signs. The first kind must begin with a *lower-case letter*. Those atoms made from signs normally are made up from signs *only*. Sometimes it may be desirable to have an atom beginning with a captal letter or a digit. For this case, an atom may be enclosed in *single quotes:* then the atom may have any character in its name. Finally, the *underscore character* may be placed in the middle of an atom to improve readability. The following are examples of valid atoms:

```
alex   nil   =   'Spirit-of-Biel-Bienne'   \+   spirit_3
```

And the following are *not* valid atoms:

```
239   IS_Biel   _alpha   c++
```

Numbers are the second kind of constants. Integers and floating point numbers are supported. Examples:

```
15   -2398   85.753   -81.92E-7
```

Characters are stored as small integers: their ASCII code. There is some controversy about *character strings* in Prolog. Some implementations (e.g. SWI, Turbo) have strings as a separate data type, others (e.g. Aquarius) treat strings as *lists of characters* only. The ISO standard draft (1993) does not mention strings.

Variables are the second kind of terms in Prolog. Variables begin with a capital letter or an underscore sign. A variable stands for some object that we may not be able to name. Examples of valid variables are:

```
X   Y   Answer   Spirit_3   _8a
```

Sometimes one needs to use a variable, but its name will never be used. In these cases we can use the *anonymous variable*, i.e the single underscore character. For example, in our family problem, if we want to ask the system whether person *eva* is married, we would write:

```
?- married(eva,_).
```

Several anonymous variables in the same clause need not be given consistent interpretations. This is a characteristic peculiar to the anonymous variable.

Compound terms or *structures* are the third kind of terms in Prolog. A compound term can be described by the following syntax diagram:

The *functor* is an atom which can be considered as the *name* of the structure. The *components* of the structure are themselves *terms*, enclosed in round brackets and separated by commas. The number of components is sometimes called the *arity* of the compound term. Example:

```
vehicle('Spirit_3', driver('Balmer'), speed(129), rank(R))
```

Operators are sometimes a convenient way of writing functors. For example, arithmetic operations are commonly written as operators rather than functors. When we write the arithmetic expression *x + y * z* we call the plus sign and the multiply sign *operators*. If we had to write the expression *x + y * z* in the normal way for structures, it would look like this: +(x,*(y,z)).

It is important to note that the operators do not cause any arithmetic to be carried out. So, in Prolog, 3 + 4 does not mean the same thing as 7. The term 3 + 4 is just another way of writing +(3,4), which is a data structure. In order to evaluate such a term numerically we have to use the `is` predicate, as we saw in the ginger bread example.

Operators are characterized by three things: *position*, *precedence* and *associativity*.

Position: prefix, infix, or postfix.

Precedence: an integer that determines the order in which operators group their operands.

Associativity: left or right

Data types. We have seen that constants can have different data types: atoms, integers, floating point numbers (strings). However, Prolog is *not* a typed language. Variables are not declared to be of a certain type. A variable can stand for *any* term, be it simple or compound. Type conversions between integers and floating point numbers are performed automatically when needed.

As we shall see later, variables may be *bound* (or *instantiated*) to atoms or compound terms during the execution of a Prolog program. There are several built-in predicates which allow for the verification of the type of a term at run-time. Examples:

```
var(T)  integer(T)  float(T)  atom(T)  atomic(T) ...
```

Exercises

(2.1) Find out which of the following sentences are legal Prolog terms and explain them:

```
a  Beta  \+  delta(X,chi(Y,gamma))  sigma(2,5.6)  Spirit('3')

3*8.45+a   +(7,3)   .(alfa,X)    X \== Y
```

(2.2) Using your user's manual and by experiment, find out how your Prolog interpreter treats character strings. Try the following goals and explain what happens:

```
?- S = "Hello world".

?- A = 'Hello world'.
```

(2.3) Try to find out what operators are built-in in your Prolog interpreter. What are their position, precedence and associativity. Are there operators having more than one meaning?

(2.4) Experiment with your Prolog interpreter and explain what happens. Try the following goals:

```
?- X = *(3,+(4,5)).

?- X = +(3,*(4,5)).

?- X = *(3,*(4,5)).

?- X = *(*(3,4),5).
```

The Meaning of a Logic Program

In logic programming, the basic idea can be summarized in the two metaphorical equations:

program =

computation =

A logic program is a *set of axioms*, or rules, defining relationships between objects. A computation is a *deduction* of consequences of the program. The set of all consequences of the given axioms is the *meaning* of the program.

Example: In our family program, we have defined facts for the *parent* and *male* relationships, and we have set up rules for the *brother* and *uncle* relationship. Using these rules, it is possible to deduct the fact that *bruno* is the uncle of *olga.* Therefore the goal

```
?-
```

is within the meaning of our program. On the other hand, the goal

```
?-
```

cannot be found as being a consequence of the program; therefore it is not within its meaning.

We say that a logic program P is *correct* with respect to some intended meaning M if the meaning of the program M(P) is a subset of M, i.e. any consequence of P is in M. For example, in the context of our family problem, the following logic program for the relationship *brother-in-law* is correct (but not complete, see below):

Note that every person X for which the goal

```
?-
```

succeeds, is indeed the brother-in-law of the respective person Y.

We say that a logic program P is c*omplete* with respect to some intended meaning M if M is a subset of the meaning of the program M(P), i.e. the program can "produce" all intended consequences (but perhaps more than that). For example, the following program for the *brother* relationship is a complete (but not correct) logic program:

In a *complete* and *correct* program P the meaning of the program M(P) is equal to the intended meaning M.

The art of logic programming is to construct concise and elegant programs that have the desired meaning, i.e. correct *and* complete programs.

We have said that the meaning of a logic program is the set of all its consequences. Logical consequences are obtained by applying *deduction rules*. We shall discuss three deduction rules:

(1) identitity

(2) generalisation

(3) instantiation

The first and most simple deduction rule is

Identity:

This is, of course, almost trivial. In the context of a logic program it says for instance that, if there is a fact

in the program then the goal (or query)

```
?-
```

will succeed.

The second and third rule are more complicated. First, we need to define some additional notions. Remember that *terms* are the single data structure in logic programs. We already know their inductive definition:

> A term is either a *constant* or a *variable* or a *compound term*. A compound term comprises a *functor* and one or more *arguments* which are terms themselves.

Terms not containing any variables are called *ground*.

Examples: `alfa(beta,gamma(delta))`

`alfa(X,beta)`

A *variable* in a logic program stands for an unspecified entity. A *substitution* S is a finite (possibly empty) set of pairs of the form $X_i = t_i$, where X_i is a variable and t_i is a term. X_i must be different from X_j for $i \neq j$, and may *not occur* in any t_j .

Example: S =

Substitutions can be *applied* to terms. The result of applying a substitution S to a term T, denoted by TS, is the term obtained from T by replacing every occurrence of X_i by t_i for every pair $X_i = t_i$ in S.

Example: Let T be the term `parent(X,peter)` and S = `{X = eva, Y = alex}`. Then TS is the term

A term A is called an *instance* of the term B iff there is a substitution S such that A = BS.

Example: `parent(eva,peter)` is an instance of `parent(eva,Z)`, the substitution is

Let us now recall what the meaning of a query is, if it contains variables. Consider, for example, the following goal:

```
?-
```

This reads as follows: "Is there a person X such that parent(X,pascal) can be deduced from the program?" We say that the variables in a query are *existentially quantified*. Of course, there is such a person, X = silvia, since we have the fact parent(silvia,pascal) in our program. This is the second deduction rule:

Generalisation:

Remember: `parent(silvia,pascal)` is an instance of `parent(X,pascal).`

Variables can also be used in facts and rules. If they appear in facts or in the head of a rule they are *universally quantified*:

father(X,Y) :- ... is a rule which says that *for any X and Y* , X is the father of Y, etc....

relatives(X,X). expresses the fact that relatives is a reflexive relationship for *any X*

knows(X,Y). could stand for the fact that *anybody* knows *everybody*

Of course, from the fact knows(X,Y) we can deduce the fact knows(peter,silvia). This is the third deduction rule:

Instantiation:

Unification: One of the most important issues of logic programming is the concept of *unification*. Informally, unification is the "attempt" to make two terms *equal*. According to this, the functor '=' is used in Prolog to perform unification.

Let us start with an example. Consider the following two terms:

t_1 =

t_2 =

Is it possible that t_1 means the same thing as t_2 , i.e.

Yes, we need the substitution

Therefore t_1 can be made equal to t_2 , we say that t_1 and t_2 are *unifiable*.

Unification can be defined mathematically as follows:

> A substitution U is a *unifier* of two terms t_1 and t_2 iff the instance t_1U is identical to t_2U. Terms are said to be *unifiable* if there exists a unifier for them. Otherwise they are *not unifiable*.

A unifier is a *most general unifier MGU* of terms if any other unifier U of these terms is an instance of it. A most general unifier always exists for terms if they are unifiable. The most general unifier MGU of terms can be found using a non-deterministic algorithm, called "Herbrand algorithm" , which is presented on the next page (source: ISO/IEC Prolog Committee Draft Standard, March 1993). However, in practice, unifying terms and finding an MGU is not a very difficult task, as we shall see in the following examples:

(1) t_1 = `plan(i3,F,X).`

t_2 = `plan(Y,Z,prof(W)).`

A unifier U would be

However, this is not the most general unifier. Instead

is a most general unifier.

(2) Try to unify the following term pairs, and if this is possible note the MGU.

(2a) `race(darwin,adelaide,dist(D)) = race(X,Y,dist(3000))`

(2b) `race(X,Y,dist(D)) = race(darwin,Z,W)`

(2c) `pilot(spirit3,name(Z)) = pilot(X,balmer)`

(2d) `drives(X,road(darwin,Y)) = drives(balmer,road(W,Z))`

(2e) `add(0,X,X) = add(Y,3,Z)`

(2f) `mult(1,X,X) = mult(5,8,Y)`

An Abstract Interpreter for Logic Programs

We have alread said that a logic *program* is a set of axioms, and a *computation* is a constructive proof of a goal statement. The computation progresses via *goal reduction*. In order to explain and understand the details of this process we shall use an abstract interpreter for logic programs. It is a non-deterministic algorithm working with a *resolvent*, i.e. a *conjunction of goals* to be proved.

Input: a logic program P, and a goal G

Output: GT if this was the instance of G deduced from P, or *failure*.

Algorithm: Initialize the resolvent to be G, the input goal.
While the resolvent is not empty do
(1) Choose a goal A from the resolvent and a *fresh** *copy* of a clause A' :- B_1,B_2, ...B_n. , n ≥ 0, from P, such that A and A' unify with most general unifier T (exit if no such goal and clause exist).
(2) Remove A from the resolvent and add B_1,B_2, ...B_n at its place.
(3) Apply T to the resolvent and to G.
If the resolvent is empty output G, else output *failure*.

* *fresh copy* means: rewrite the clause with variables not yet used.

An example computation follows on the next page.

There are two *choices* in our abstract interpreter: choosing the goal to reduce and choosing the clause to effect the reduction. These must be resolved in any realization of the computation model. The nature of the choices is fundamentally different.

The choice of *goal* to reduce is arbitrary; it does not matter which is chosen for the computation to succeed. If there is a successful computation by choosing a given goal, then there is a successfull computation by chosing any other goal.

The choice of the *clause* to effect the reduction is, in general, non-deterministic. Not every choice will lead to a successful computation.

For some computations, for example, the computation of finding a solution for the goal `?- father(oscar,X)` there is *always only one* clause from the program which can reduce each goal. Such a computation is called *deterministic*.

The alternative choices that can be made by the abstract interpreter, when trying to prove a goal, implicitely define a *search tree*. In order to be sure not to "loose" a solution, the interpreter should search the whole tree. As we all know, there are different strategies to traverse a tree.

(1) *Breadth-first search* (or level-order tree traversal) explores the search tree "in parallel" and guarantees that, if there is a finite proof of the goal (i.e. a finite successful path in the search tree), it will be found.

(2) *Depth-first search*. In contrast to the breadth-first strategy, the depth-first search does not guarantee finding a proof even if one exists, since the search tree may have infinite paths. A depth-first search of the tree might get lost in an infinite path, never finding a finite successful path, even if one exists.

In technical terms, the breadth-first strategy defines a *complete* proof procedure for logic programs, whereas the depth-first strategy is *incomplete*.

Example computationon the abstract interpreter: Given our family program, let us follow the search for an *uncle* of *olga*. Remember the clauses for the *brother* and *uncle* relationship:

```
brother(X,Y) :-  parent(P,X), parent(P,Y), male(X), X \= Y.

uncle(X,Y) :- brother(X,Z), parent(Z,Y).
```

The Execution Model of Prolog

In spite of its incompleteness, *depth-first* search is the strategy incorporated in *Prolog*, for practical reasons. The following defines Prolog's execution mechanism (or *inference engine*):

> **Prolog's execution mechanism is obtained from the abstract interpreter by choosing the *leftmost* goal in the resolvent, instead of an arbitrary one, and by replacing the nondeterministic choice of a clause in the program by *sequential search* for a unifiable clause and *backtracking*.**

In other words, Prolog adopts a *stack scheduling policy*. It maintains the resolvent as a stack, pops the top goal for reduction, and pushes the derived goals on the resolvent stack.

The main difference between logic programs and Prolog programs is that the order of the clauses and the order of the goals within the body of a rule are very important in Prolog programs. Remember that the Prolog interpreter chooses the *clauses from top to bottom*, and reduces the *goals from left to right*. The effect of good (or bad) decisions about rule order and goal order can be immense. There can be orders of magnitude of difference in efficiency in the performance of Prolog programs. In extreme, correct logic programs will fail to give solutions due to nontermination.

Exercises:

(2.5) Try to find out the influence of *rule order* and *goal order* by experimenting with the following predicates. Use different *flow patterns*.

(a)
```
father1(X,Y) :- parent(X,Y), male(X).
```

(b) What happens in (a) if the facts of `parent/2` are reordered?

(c)
```
ancestor1(X,Y) :- parent(X,Y).
ancestor1(X,Y) :- parent(X,P), ancestor1(P,Y).
```

(d)
```
ancestor2(X,Y) :- parent(X,P), ancestor2(P,Y).
ancestor2(X,Y) :- parent(X,Y).
```

(e)
```
father2(X,Y) :- male(X), parent(X,Y).
```

(f)
```
ancestor3(X,Y) :- parent(X,Y).
ancestor3(X,Y) :- parent(P,Y), ancestor3(X,P).
```

(g)
```
ancestor4(X,Y) :- parent(X,Y).
ancestor4(X,Y) :- ancestor4(X,P), parent(P,Y).
```

(h)
```
ancestor5(X,Y) :- parent(X,Y).
ancestor5(X,Y) :- ancestor5(X,Z), ancestor5(Z,Y).
```

(2.6) Revise the predicates `brother/2` and `uncle/2` and determine the optimal goal order for the following flow patterns (represented by some typical goals). Hint: use time/1.

(a) `(o,i)` `?- uncle(X,olga).`

(b) `(i,o)` `?- uncle(peter,Y).`

(c) `(o,o)` `?- uncle(X,Y).`

Summary

- In Prolog, almost everything is a *term*: *atoms* and *numbers*, *variables* and *structures* (compound terms).
- *Compound terms* are constructed by means of *functors*. A functor is defined by its *name* and *arity*. For some predicates *operators* are a convenient way of writing functors.
- The lexical scope of variables is *one* clause. Thus the same variable name in two clauses means two different variables.
- Constants can have different *data types*: atoms, integers, floating point numbers, characters, (strings). However, variables are not bound to a particular data type.
- In logic programming, the program is a *set of axioms*. A computation is a *constructive proof* of a statement. The set of all consequences of the given axioms is the *meaning* of the program. It is found by applying *deduction rules*: identity, generalisation, and instantiation.
- Variables that appear in a fact or in the head of a rule are *universally quantified* (for all X ...). All others are *existentially quantified* (there is an X, such ...).
- *Unification* is the operation that takes two terms and tries to make them identical by instantiating the variables in both terms. It is the single, very powerful *matching* operation that in effect performs a number of tasks for which other programming languages use different operations: comparisons (recursive!), assignment statements, and parameter passing.
- The execution mechanism of Prolog, its *inference engine*, is based on a *goal-driven* algorithm known as *abstract interpreter*. It maintains a stack of goals as *resolvent* and applies a *sequential search* for unifiable clauses, and *backtracking*.
- The declarative meaning of a logic program does not depend on the order of the clauses and the order of goals in clauses. However, the procedural meaning does depend on the order of clauses and goals. It can affect the efficiency of the program; on unsuitable order may even lead to infinite recursive calls.

3. Arithmetics

Prolog is mainly a language for symbolic computation where the need for numerical calculation is relatively modest. Accordingly, the means for numerical computing in Prolog are rather simple. Some of the predefined operators can be used for basic arithmetic operations: addition (+), subtraction(-), multiplication (*), division (/ and //), modulo (mod), and power (^). Some mathematical functions, like sqrt(), trigonometric functions, random() and some others are usually predefined. See the manual of your Prolog system for details.

Precedence and associativity of the arithmetic operators are defined in such a way that the usual rules for the evaluation of arithmetic expressions apply. Therefore you write down arithmetic expressions just like in other programming languages (e.g. Pascal). There is only one little problem. The following query is a naive attempt to request arithmetic computation:

```
?- X = 1 + 2 * 3.
```

Prolog will quietly answer:

```
X =
```

The reason for that is simple: the expression 1 + 2 * 3 merely denotes a term +(1,*(2,3)) and the '=' forces Prolog to unify this term with the variable X. There is nothing in the above goal to actually activate the multiplication and addition operations. In order to force an evaluation we need to use the `is` operator:

```
?- X is 1 + 2 * 3.
```

There are other operators that force evaluation as well: `>, <, >=, =<, =:=, =\=`

Some practical examples

1. Population statistics of some European countries (1985)

```
% pop(C,P) :- the population of   % area(C,A) :- the area of the
% the country C is 1000 * P       %    country C is A square km

pop('B', 9858).                   area('B', 30518).
pop('D', 77655).                  area('D', 357050).
pop('DK', 5112).                  area('DK', 43069).
pop('F', 55138).                  area('F', 543965).
pop('GR', 9935).                  area('GR', 131957).
pop('GB', 56648).                 area('GB', 244046).
pop('IRL', 3537).                 area('IRL', 70283).
pop('I', 57128).                  area('I', 301225).
pop('L', 366).                    area('L', 2586).
pop('NL', 14484).                 area('NL', 41785).
pop('P', 10229).                  area('P', 92082).
pop('E', 37409).                  area('E', 497477).
pop('A', 7558).                   area('A', 83849).
pop('S', 8358).                   area('S', 449964).
pop('SF', 4910).                  area('SF', 338145).
pop('CH', 6374).                  area('CH', 41288).


% density(C,D) :- the population density of the country C
%    is D inhabitants per square kilometer
```

We can ask the following questions:

(a) What is the population density of Switzerland?

```
?-
```

(b) Are there countries with a population density less than 50 persons per square kilometer?

```
?-
```

(c) What countries are smaller than Switzerland?

```
?-
```

Let us write one more predicate:

```
% bigger_than(A) :- there is a country bigger than A square km
```

How can we find the biggest country?

```
?-
```

2. Let us compute the number of *completely balanced binary trees* having N nodes. A binary tree is completely balanced if for any node the following property holds: the number of nodes in its left subtree and the number of nodes in its right subtree differ by not more than 1.

```
% ncbbt(N,T) :- T is the number of completely
%    balanced binary trees having N nodes
```

3. Factorial and greatest common divisor:

```
% fact(N,F) :- F is N!
```

```
% gcd(X,Y,G) :- G is the greatest common divisor of X and Y
```

Some Prolog systems (e.g. SWI-Prolog) allow for the *registration* of arithmetic functions:

```
?- arithmetic_function(gcd/2).
?- G is gcd(24,36).
G = 12
```

4. Fibonacci numbers.

```
% fib(N,F) :- F is the N'th Fibonacci number.
```

There is a problem with this solution: it is terribly inefficient, because intermediate results are calculated over and over again. Let's have a look at a much more efficient solution as we would program it in Java (we suppose n > 0 is guaranteed):

```
int fibo(int n) {
    if (n == 1) return 1;
    int k = 2, fk = 1, fkm1 = 1, hold;
    // Assertion: k<=n and fk=f(k) and fkm1=f(k-1)
    while (k < n) {
        // Assertion: k<n and fk=f(k) and fkm1=f(k-1)
        hold = fk + fkm1;
        k = k + 1;
        fkm1 = fk;
        fk = hold;
        // Assertion: k<=n and fk=f(k) and fkm1=f(k-1)
    }
    // Assertion: k=n and fk=f(k)
    return fk;
}
```

The above algorithm can also be programmed without a while-loop in a purely recursive way:

```
int fibo(int n) {
    if (n == 1) return 1;
    if (n > 1) return fibo(n,2,1,1);
    return 0; // never happens; compiler!
}

int fibo(int n, int k, int fk, int fkm1) {
    // Assertion: k<=n and fk=f(k) and fkm1=f(k-1)
    if (k == n) return fk;
    if (k < n) return fibo(n, k+1, fk+fkm1, fk);
    return 0; // never happens; compiler!
}
```

How can we implement the idea in Prolog?

```
% fibo(N,F) :- F is the N'th Fibonacci number
% Iterative solution
```

```
% fibo(N,F,K,FK,FKm1) :- F = fib(N), we know
% that FK = fib(K) and FKm1 = fib(K-1). (i,o,i,i,i)
```

Exercises

(3.1) The predicate fact/2 as developed in the course is not tail-recursive. Try to apply the idea we discussed for the fibo problem in order to rewrite fact/2 in such a way that it becomes an *iterative* (tail-recursive) program.

(3.2) In a *(height-) balanced binary tree* the following property holds for every node: the depth of its left subtree and the depth of its right subtree differ by not more than 1. How many different balanced binary trees of depth 5 are there? Write a predicate `nbbt(D,N)` to calculate the number N of distinct balanced binary trees of depth D.

(3.3) Write a predicate `prime(P)` that succeeds if and only if P is a prime number.

(3.4) One of the operations often used in cryptology is modular exponentiation: (A^B) mod M, where A, B are non-negative integers, M is a positive integer. Write a predicate `power_mod(A,B,M,P)` that computes (A^B) mod M and returns the result in P. Try to keep intermediate results as small as possible: they should never exceed M*M. Can your program handle to query `?- power_mod(9876,2345678,1357,P)` ?

(3.5) The harmonic series $S_n = 1 + 1/2 + 1/3 + 1/4 + 1/5 + ... + 1/n$, n = 1, 2, 3, ... is known to be divergent, i.e. S_n exceeds every given limit M, if only n is choosen sufficiently large. Write a Prolog program that calculates the smallest n such that $S_n > M$, for a given limit M.

(3.6) Egyptian camel problem: Find four positive integers, A, B, C, and D that satisfy the equation 1/A + 1/B + 1/C + 1/D = 1. Hints: (1) Without loss of generality we can assume that A <= B <= C <= D. (2) Use the predefined predicate between/3 to generate integer values. (3) The original camel puzzle was A = 2, B = 3, C = 9.

Summary

- Built-in predicates are used to do arithmetics.
- Arithmetic operations have to be explicitely requested with the predefined predicate *is*. There are built-in predicates associated with the predefined operators +, -, *, /, //, mod, and others.
- At the time when arithmetic evaluation is carried out, all arguments must *already be instantiated* to numbers.
- The values of arithmetic expressions can be compared by operators such as <, >, =<, etc. These operators force the evaluation of their arguments.
- *Iterative* algorithms can be programmed in Prolog using *accumulator varibales* and *tail-recursive* auxiliary predicates.

4. Lists and Trees

The *list* is a simple data structure widely used in non-numerical programming. In Prolog it is probably the most frequently used data structure. Graphically, a list is a *linear sequence* of items.

In chapter 1 we have defined the descendants/2 predicate, which returns the set of descendants of a given person, as in

```
?- descendants(Ds,silvia).

Ds = [jean,olga,otto,pascal,tina]
```

The result of this query is a list of atoms in Prolog notation. Other examples of Prolog lists are:

```
[2,3,5,7,11,13]

[olga, 17, book('Bratko','Prolog', N), 35.4, a]
```

The last example shows that the items of a list do not have to be of the same type.

A list containing no elements is called *empty*. The empty list is a special Prolog atom: `[]`.

If a list is not empty it can be viewed as consisting of two things:

(1) the first item, called the *head* of the list

(2) the remaining part of the list, called the *tail*.

For example, in the list `[jean,olga,otto,pascal,tina]`, the head of the list is the atom `jean`, its tail is the list `[olga,otto,pascal,tina]`

It is important to note that head and tail of a list have different quality. In general, the head can be any Prolog object (constant, variable, compound term); the tail is always a list.

In order to allow the programmer to separate head and tail of a list, Prolog provides a syntactical extension, the vertical bar (|). Consider, for example, the following goals:

```
?- [Head|Tail] = [jean,olga,otto,pascal,tina].

Head = jean
Tail = [olga,otto,pascal,tina]

?- [Head|Tail] = [4711].

Head = 4711
Tail = []

?- [Head|Tail] = [].
No
```

In fact, the vertical bar notation is more general: we can list any number of elements, separated with commas, followed by '|' and the list of remaining items. For example:

```
?- [A,B,C|Tail] = [jean,olga,otto,pascal,tina].

A = jean
B = olga
C = otto
Tail = [pascal,tina]
```

Let us now develop some predicates to manipulate lists. Whenever dealing with lists, remember their recursive definition:

> **A list is either empty or it consist of a first item (head) and a remainder (tail) which is itself a list.**

```
% element(X,L) :- X is an element of the list L
%    Note: See the predefined predicate member/2
```

```
% prefix(P,L) :- P is a prefix list of the list L
%    Example: [2,3,5] is a prefix list of [2,3,5,7,11]
```

```
% suffix(S,L) :- S is a suffix list of the list L
%    Example: [5,7,11] is a suffix list of [2,3,5,7,11]
```

```
% partlist(U,L) :- U is a consecutive part of the list L
%    Example: [5,7] is a partlist of [2,3,5,7,11]
```

```
% conc(X,Y,Z) :- the list Z is the result of concatenating
%    the list X and the list Y
%    Note: See the predefined predicate append/2
```

```
% del(X,L1,L2) :- the list L2 is obtained by deleting
%    the element X from the list L1
%    Note: See the predefined predicates delete/3 and select/3
```

We can use the del/3 predicate to elegantly define another predicate that inserts elements into a list:

```
% ins(X,L1,L2) :- L2 is the result of inserting X into the
%     the list L1
```

Using del/2 or ins/2, it is fairly easy to write a predicate that permutes the elements of a list:

```
% permu1(L1,L2) :- L2 is a permutation of the list L1
```

```
% permu2(L1,L2) :- L2 is a permutation of the list L1
```

For the next predicates we assume that the elements of the lists are numbers.

```
% sum_list(L,S) :- S is the sum of the numbers in the list L
```

The above predicate sum_list/2 is not tail-recursive. An alternative is the following:

```
% sum_list(L,S) :- S is the sum of the numbers in the list L


% sum_list(L,S,A) :- S is A plus the sum of the numbers in L
```

```
% scalar_product(X,Y,P) :- P is the dot-product of the vectors
%    X and Y. Vectors are lists in Prolog.
```

We have emphasized the fact that a non-empty list is composed of two things, a head and a tail. We can therefore conceive a non-empty list as a compound term having two arguments: head and tail. Prolog uses the special functor `./2` to denote this sort of term. Thus, the bracket notation is nothing else than a convenient way of writing compound terms with this functor. Example:

`[alfa,beta,gamma]` is the same as:

Exercises

(4.1) A probability vector is a list of non-negative real values whose sum equals 1. Write a predicate `probability_vector/1` that checks whether a given list is a probability vector. Allow for a certain numerical inaccuracy due to rounding errors. Try to find a solution which fails as soon as possible if the list does not satisfy the rules for a probability vector.

(4.2) Write a predicate `maximum/2` to find the greatest number in a list. What is a sensible value for the maximum of an empty list?

(4.3) Write a predicate `monotonic/1` to test whether a given list is a (non-strictly) monotonic sequence of numbers. We define: an empty list is not monotonic.

(4.4) Develop a predicate `range/3` to generate the list of all integers of a given interval, in ascending order. Example: `range(3,7,L)` should unify `L` with the list `[3,4,5,6,7]`. `range(X,Y,L)` should fail if `X > Y`. Do not use the predefined predicate `between/3`.

(4.5) Write a predicate `rev/2` to reverse the order of the element of a list. Example: `rev([2,3,5,7],L)` should unify `L` with `[7,5,3,2]`. Use the predefined predicate `time/1` to compare your solution with a predefined predicate `reverse/2` (if these predefined predicates are available).

(4.6) A palindrome is a list that reads the same forward and backward, e.g. [1,3,7,11,7,3,1] or [o,t,t,o]. Write a predicate `palindrome/1` to check whether or not a list is a palindrome.

(4.7) Try to find out what *exactly* the following predicate does:

```
sqr(X,Y) :- number(X), Y is X*X.
sqr([],[]).
sqr([X|Xs],[Y|Ys]) :- sqr(X,Y), sqr(Xs,Ys).
```

Example goal: `?- sqr([1,[2,3]],M).`

(4.8) Define a predicate `subsets/2` that generates all subsets of a given set. Example: `subsets([a,b,c],S)` should unify `S` (via backtracking) with `[a,b,c]`, `[a,b]`, `[a,c]`, `[a]`, `[b,c]`, `[b]`, `[c]`, and `[]`.

(4.9) Data compression through run length coding. A list of items, e.g. [2,2,2,1,1,7,7,7,7,5,5,5] shall be encoded into a list of codes, e.g. [c(2,3),c(1,2),c(7,4),c(5,3)], where each element c(X,N) represents a "run" of N consecutive items X. Write two predicates, `encode/2` to translate a list of items into a code list, and `decode/2` to perform the inverse translation.

(4.10) Lions club. In a zoological garden there is a row of cages where wild animals can be locked up. However, if two lions are confined in adjacent cages they start roaring at each other and get very aggressive. Therefore one has decided to never put two lions next to each other, but to let at least one empty cage between them. If there are n cages in a row, how many ways are there to lock up lions? Write a predicate `lions/2` that generate lists of 1's and 0's denoting lions and empty cages respectively. A description of the predicate could be:

```
% lions(N,L) :- L is a list of N 1's and 0's, denoting lions
%    and empty cages, respectively. Between two lions there
%    is at least one empty cage.
% (integer, list)  (i,i), nondeterm (i,o)
```

A solution of `lions(7,L)` would be (among many others) `L = [0,1,0,0,1,0,1]`. The predicate `lions/2` should generate all solutions by backtracking.

How many solutions are there for a given N?

Binary Trees

> **A binary tree is either empty or it consists of a root, a left subtree, and a right subtree. Both subtrees are themselves binary trees.**

This very useful definition can be immediately translated into Prolog terms:

- An empty tree is denoted by the atom `nil`.
- A non-empty tree is written as a term `t(X,L,R)`, with `X` representing the root and `L` and `R` representing the left and right subtree, respectively.

Example:

T = t(a,t(b,nil,nil),t(c,t(d,nil,nil),nil))

Using this notation, we can implement many algorithms on binary trees as comparatively simple Prolog predicates. For the moment we assume that the binary trees are instantiated when the following predicates are invoked. In chapter 6 we are going to discuss other flow patterns as well.

```
%  nodes(T,N) :- the binary tree T has N nodes
% (tree,integer);  (i,*)
```

```
% depth(T,D) :- the binary tree T has depth D.
% (tree,integer); (i,*)
```

```
% leaves(T,L) :- the binary tree T has L leaves.
% (tree,integer); (i,*)
```

```
% cbal(T,N) :- T is a completely balanced tree with N nodes
% (tree,integer); (i,*)
```

```
almost_equal(X,Y) :-
```

```
% preorder(T,L) :- L is the preorder sequence of T
% (tree,list); (i,i),(i,o)
```

We conclude this section with a discussion of a *binary search tree* which is sometimes also called a *binary dictionary*. Such a data structure can be used if there is an ordering relation between the data stored in the nodes.

We say that a non-empty tree t(X,Left,Right) is ordered from left to right if:

(1) all the nodes in the left subtree, Left, are less than X; and

(2) all the nodes in the right subtree, Right, are greater than X; and

(3) both subtrees are also ordered.

We assume that there are no duplicate data elements. Furthermore, we introduce comparison operators @< and @> to denote 'less than' and 'greater than', with respect to a standard order of terms.

Suppose now that T is a binary dictionary, and we want to find out, whether a data element X is stored in one of the nodes of T:

```
% in(X,T) :- X is in the binary dictionary T
```

How can we insert a new data item into an existing binary dictionary? The following predicate does the job:

```
% add(X,T1,T2) :- the binary dictionary T2 is obtained by
% adding the item X to the binary dictionary T1
```

Deleting a node from a binary tree is a more serious problem which we leave as a challenging exercise.

General Trees

A *general tree*, or simply *tree*, consists of a distiguished node, called the *root*, and an ordered set of subtrees which are trees themselves.

A *forest* is an ordered set of trees.

Note that with this set of definitions a tree cannot be empty, but a forest can.

In Prolog, we can represent a an ordered set as a list, and therefore the natural representation of a tree is a term t(Root,Forest), where Forest is a list that is either empty or contains subtrees, which in turn are terms of the type t(Root,Forest).

Example:

```
t(a,[t(b,[t(c,[]),t(d,[]),t(e,[])]),
     t(f,[t(g,[])]),
     t(h,[t(j,[]),t(k,[])])
    ])
```

Evidently, the above notation is not very easy to trace for a human reader. A much more convenient notation for the above tree would be:

We shall return to this notation in a moment. But first, let us write some simple predicates:

```
% nodes(T,N) :- the general tree T has N nodes
% (tree,integer) (i,*)
```

The degree of a general tree is defined as the maximum number of subtrees that a node of the tree has. For instance, the tree of our above example has degree 3.

```
% degree(T,D) :- the general tree T has degree D
% (tree,integer) (i,*)
```

We shall now develop a predicate which prints a tree in the human-friendly string notation mentioned at the beginning of this section. The starting point is the syntax of the string representation.

```
% print_tree(T) :- print the tree T in human-friendly form
```

It should be noted that the above predicate `print_tree/1` performs a depth-first order tree traversal.

Exercises

(4.11) Define a predicate `is_tree/1` to check whether or not a given (instantiated) term is a binary tree, according to our definition.

(4.12) Write a predicate `hbal/2` with the following meaning:

```
% hbal(T,D) :- T is a height-balanced tree of depth D
% (binary_tree,integer); (i,*)
```

(4.13) Define predicates `inorder/2` and `postorder/2` with the following meaning:

```
% inorder(T,L) :- L is the inorder sequence of T
% (binary_tree,list) (i,*)

% postorder(T,L) :- L is the postorder sequence of T
% (binary_tree,list); (i,*)
```

(4.14) Modify the predicate `in/2` by adding a third argument `Path`, so that `Path` is a list representing the path between the root of the binary dictionary and the item X.

(4.15) Develop a predicate `balanced_tree/2` to create a completely balanced binary dictionary with a given number N of nodes. The nodes should contain the integer numbers 1 to N as data items.

(4.16) Define a predicate `del/3` to delete a given node (holding a data item X) from a binary dictionary. Hint: A predicate `delmin/3` which deletes the minimal element from a binary dictionary could be helpful in certain situations.

(4.17) Write a predicate `depth/2` to determine the depth of a general tree.

(4.18) Develop a predicate `bottom_up/2` that traverses a general tree in bottom-up order and puts the contents of the nodes in a list.

Summary

- The *list* is a frequently used data structure. It is either *empty* or consists of a *head* and a *tail* which is a list as well. Prolog provides a special notation for lists.
- Common operations on lists are: membership, prefix, suffix, partlist, concatenation, adding and removing an element, permutation.
- A *binary tree* is either empty or consists of a root node and two subtrees which are binary trees as well. It is elegantly represented in Prolog using a special atom (nil) and a term t(Root,Left,Right). Several tree traversal methods can be programmed as recursive predicates in a most succinct way.
- A *general tree* consists of a root node and a (possibly empty) ordered set of subtrees which are general trees themselves. An ordered set of trees is called a *forest*. In Prolog, a general tree can be represented as a term t(Root,Forest), where Forest is either [] or a list of terms representing general trees.

5. Programming Techniques

Generate-and-Test

Many combinatorial problems can be successfully solved using a method commonly known as *generate-and-test*. One process, or routine, *generates* the set of of candidate solutions to the problem, and another process, or routine, *tests* the candidates, trying to find one, or all, of the candidates which actually solve the problem.

Typically, generate-and-test programs are easier to construct than programs that compute the solution directly, but they are also less efficient. A standard technique for optimizing generate-and-test programs is to try to "push" the tester inside the generator, as "deep" as possible. We shall develop this concept in this chapter.

In Prolog, it is easy to write generate-and-test programs. Such programs typically have the following main structure:

The following combinatorial problem is typical for many others, which can be successfully solved using the *generate-and-test* paradigm.

Problem: The postal service of Utopia sells four different types of stamps: 1, 4, 12, and 21 UCs (Utopian Cents). Utopian law does not allow to put more than five stamps on a letter. How can a given postage amount A be composed?

Examples: 37 = 21 + 12 + 4 or 28 = 21 + 4 + 1 + 1 + 1

For some amounts (e.g. A = 28) there are several different solutions, for others there is no solution (e.g. A = 73).

Try A = 32.

We are going to develop an elegant solution in a step-wise approach. For all versions, we suppose the following description of the main predicate:

```
% post(A,Ss) :- Ss is a list of not more than 5 stamps
% with a total value of A  (+,?)
```

We assume a predicate `stamp/1` with the following facts:

```
stamp(21).
stamp(12).
stamp(4).
stamp(1).
```

In a first version, we solve the problem with a pure generate-and-test concept:

```
% stamplist(L,N) :- L is a list of not more than N stamps
% (?,+)
```

```
% sum(L,S) :- S is the sum of the numbers in the list L.

sum(L,S) :- sumlist(L,S,0).

sum([],S,S).
sum([X|Xs],S,A) :- B is A + X, sum(Xs,S,B).
```

The above solution is correct. However, we get all permutations of the stamp list. How can we avoid this? One possibility is to take care that the stamps in the list are sorted descendingly. This can be achieved as follows (version 2):

```
% stamplist(L,N,M) :- L is a list of not more than N
% stamps. The stamp values in L are in (non-strictly)
% descending order and the first stamp in L has a value
% not greater than M. (?,+,+)
```

This solution correctly avoids permutations. However, our program is still rather inefficient because many solution candidates, i.e. stamp lists, with a sum much greater than the goal amount A are generated in the process. For instance, in finding a solution for A = 32, lists like [21,21,21,21,21] are generated and then rejected by the tester `sum/2`. In order to improve

this, we can push the tester `sum/2` into the generator `stamplist`. It turns out that we can even eliminate the tester completely. This is our version 3:

```
% stamplist(L,N,M,A) :- L is a list of not more than N
% stamps. The stamp values in L are in (non-strictly)
% descending order, the first stamp in L has a value
% not greater than M and the total value of the stamps
% in L is A. (?,+,+,+)
```

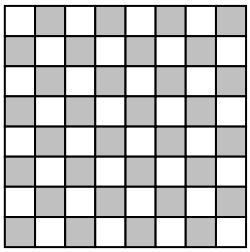
The efficiency improvement can be demonstrated when we compare the number of inferences reported for the goal:

```
?- time(post(32,Stamps)).      Version 1:

                               Version 2:

                               Version 3:
```

Another famous problem in computer science is the **Eight Queens Problem**: The objective is to place eight queens on a chessboard so that no two queens are attacking each other; i.e., no two queens are in the same row, the same column, or on the same diagonal. We generalize this original problem by allowing for an arbitrary dimension N of the chessboard.



We represent the positions of the queens as a list of numbers 1..N. Example: [1,5,8,6,3,7,2,4] means that the queen in the first column is in row 1, the queen in the second column is in row 5, etc. By using the permutations of the numbers 1..N we guarantee that no two queens are in the same row. The only test that remains to be made is the diagonal test. A queen placed at column X and row Y occupies two diagonals: one of them, with number C = X-Y, goes from bottom-left to top-right, the other one, numbered D = X+Y, goes from top-left to bottom-right.

The first version is a simple generate-and-test solution.

```
% queens(N,Qs) :- Qs is a solution of the N-queens problem
```

The following predicates, which have been introduced earlier in the course, are used as the solution candidate generator:

```
% range(A,B,L) :- L is the list of numbers A..B

range(A,A,[A]).
range(A,B,[A|L]) :- A < B, A1 is A+1, range(A1,B,L).

% permu(Qs,Ys) :- Ys is a permutation of the list Qs

permu([],[]).
permu(Qs,[Y|Ys]) :- del(Y,Qs,Rs), permu(Rs,Ys).

del(X,[X|Xs],Xs).
del(X,[Y|Ys],[Y|Zs]) :- del(X,Ys,Zs).
```

The tester is the following predicate:

```
% test(Qs) :- the list Qs represent non-attacking queens
```

```
% test(Qs,X,Cs,Ds) :- the queens in Qs, representing
% columns X to N, are not in conflict with other queens
% (in columns 1 to X-1) which occupy the diagonals
% Cs and Ds (+,+,+,+)
```

Our program is correct; all the 92 solutions to the Eight Queens Problem are found in less than a second on a modern PC. However, as in the previous example, many hopeless solution candidates are completely generated and then rejected by the tester. We can do much better by pushing the tester inside the generator, as follows:

```
% queens(N,Qs) :- Qs is a solution of the N-queens problem
```

The combined generator/tester is the following predicate:

```
% permu_test(Ys,Qs,X,Cs,Ds) :- Qs is a permutation of the
% numbers in the list Ys, representing queens in columns X
% to N, which are not in conflict with other queens (in
% columns 1 to X-1) which occupy the diagonals Cs and Ds
% (+,?,+,+,+)
```

Compare the number of inferences reported for the goal:

```
?- time((queens(8,Qs), write(Qs), nl, fail)).
```

Version 1:

Version 2:

Tracing Prolog Programs

There are several predefined predicates which enable the Prolog programmer to watch his or her program as it runs. One of them, `trace/0`, turns on exhaustive tracing, which means that the program is halted several times during the execution of every goal (and subgoal). The *four-port-tracing model* of Prolog can best be visualized with boxes that symbolize predicates.

Calling a goal means entering the CALL port of the corresponding predicate box. If the goal can be satisfied, the box is exited through the EXIT port. Later on, it can happen that another goal "further down the line" fails, which causes the program control to return to the predicate box from its "rear" entrance (which is *not* a port in the four-port tracing model). If there is no (more) way to satisfy the goal the box is exited through the FAIL port.

Let us now consider the internal structure of the predicate box. As an example, we take the well-known `brother_in_law/2` predicate:

```
% brother_in_law(X,Y) :- X is the brother-in-law of Y.

brother_in_law(X,Y) :- brother(X,P), married(P,Y).
brother_in_law(X,Y) :- husband(X,W), sister(W,Y).
```

According to the two rules of the predicate `brother_in_law/2`, its box is subdivided into two main sub-boxes, each of them containing the predicate boxes of the respective right-hand sides. A REDO port connects the two sub-boxes. Furthermore, there is a "switchbox" at the right end of the main predicate box. The position of the "switch" is memorized and used to find the right way back, if program control should return through the "rear entrance". Finally, there is a small box at the bottom which really isn't important, except for the coherent interpretation of tracing outputs. It only connects the REDO port of the last sub-box with the FAIL port of the main box.

Each predicate box within a sub-box is, of course, again subdivided internally, according to this scheme. The effect is that the boxes are recursively nested just like Russian "Babouschka" dolls. For example, the very first sub-box of the `brother_in_law/2` predicate, `brother/2`, has an internal structure, according to the single rule of this predicate:

```
brother(X,Y) :- parent(P,X), parent(P,Y), male(X), X \= Y.
```

We shall now trace the execution of the following goal:

```
?- leash(-all), trace, brother_in_law(romeo,rosa).

Call:  (  6) brother_in_law(romeo, rosa)
Call:  (  7) brother(romeo, L344)
Call:  (  8) parent(L392, romeo)
Fail:  (  8) parent(L392, romeo)
Redo:  (  7) brother(romeo, L344)
Fail:  (  7) brother(romeo, L344)
Redo:  (  6) brother_in_law(romeo, rosa)
Call:  (  7) husband(romeo, L344)
Exit:  (  7) husband(romeo, julia)
Call:  (  7) sister(julia, rosa)
Call:  (  8) parent(L392, julia)
Exit:  (  8) parent(alex, julia)
Call:  (  8) parent(alex, rosa)
Exit:  (  8) parent(alex, rosa)
Call:  (  8) female(julia)
Call:  (  9) male(julia)
Fail:  (  9) male(julia)
Exit:  (  8) female(julia)
Call:  (  8) julia \= rosa
Exit:  (  8) julia \= rosa
Exit:  (  7) sister(julia, rosa)
Exit:  (  6) brother_in_law(romeo, rosa)

Yes
```

It should be mentioned that the recursive box structure we used to explain tracing, is in fact established *dynamically* during the execution of the program. It is a picture for the dynamics of the *stack*, which is used, just like in other programming languages, to store procedure calls (= goals). In general, the maximum depth of the recursion depends on the particular goal and cannot be predicted at the time when the program is written. The actual depth of the recursion is indicated in the trace listing.

Controlling Backtracking

Prolog provides a system predicate, called *cut*, for affecting the procedural behaviour of programs. The cut, denoted ***!***, can be used to prevent Prolog from following fruitless computation paths that the programmer knows could not produce solutions.

The use of cut is controversial. Many of its uses can only be interpreted procedurally, in contrast to the declarative style of programming which makes the real power of Prolog. However, used sparingly, the cut can considerably improve the efficiency of programs, without compromising their clarity.

We start with a program that separates a list:

```
% split(Ls,Es,Os) :- the list Ls of integers is split
%    into two lists, Es and Os, containing even and odd
%    numbers, respectively.
% (list,list,list)  (i,o,o)
```

Splitting a list of integer numbers Ls into two lists according to the specification of the `split/3` predicate is a *deterministic* operation. Only one of the three clauses applies for each goal (and sub-goal) in a given computation. The programmer knows that if the second clause can be successfully applied, the third is sure to fail. However, in general, the Prolog system cannot find out whether or not two clauses are mutually exclusive. Therefore it must leave *choice points* in order to allow for later REDO calls. Technically, this means that recursive predicate calls must be kept on a *stack.* This can make the program execution inefficient, both in terms of runtime and memory usage.

We can help the system by indicating that clauses are mutually exclusive, i.e. the computation is *determinsitic*. In our example, this is done with a *cut (!)* in the second clauses of the `split/3` predicate. To show the difference, we experiment with the following goals:

```
?- split([2,3,4,5,7,0,11,12],Es,Os).

?- time(split([2,3,4,5,7,0,11,12],Es,Os)).

?- time((split([2,3,4,5,7,0,11,12],Es,Os), fail)).

?- space(split([2,3,4,5,7,0,11,12],Es,Os)).
```

```
?- split(L,[2,4,8,0,12],[1,5,3,7]).
```

The exact way how the cut works can best be explained in the box notation we know from the previous section. Consider the following Prolog program:

```
a(1).
a(2).
b(1).
c(1).
c(2).
d(1).
d(2).

g([0,0,0,0]).
g([A,B,C,D]) :- a(A), b(B), !, c(C), d(D).
g([7,7,7,7]).
```

Try to satisfy the following goals (use also `trace` and `space/2`):

```
?- g(X).

?- g([X,7,X,X]).
```

Operationally, the cut is handled as follows: *The goal, **!**, succeeds and commits Prolog to all the choices made since the parent goal (in our example: g) was unified with the head of the clause the cut occurs in.*

Arithmetic calculations are very often purely deterministic, and therefore the cut is often used in the arithmetic predicates. As an example, let us reconsider the factorial predicate:

We have said that the cut can be used to *express determinism* (as we did in the examples `split/3` and `fact/2`). Textbooks on Prolog programming have impressed a name for cuts which are used in this way:

Typically, *green cuts* are *added* to correct logic programs, and nothing else is changed.

There is, however, a more dangerous use of the cut:

In our `split/3` program, why shouldn't we omit the test goal `odd(X)` from the third clause? If a number isn't even, it has got to be odd!

But, what happens, if we do omit the test and call the goal:

```
?- split([1,2,3,alfa,5,6,7],Es,Os)
```

We might argue that this was not originally intended. However, the behaviour of the program has changed in a way that the specification of the `split/3` predicate is no longer valid.

A more serious example is the following predicate:

```
% minimum(X,Y,M) :- M is the minimum of X and Y
```

```
?-
```

It should be noted that the predicate `minimum/3` gives correct results when the flow pattern (i,i,o) is used. But, for the flow pattern (i,i,i) the program is no longer correct!

As already pointed out, red cuts are a very dangerous thing. They may completely destroy the declarative clarity of a logic program. For this reason they should be avoided as much as possible by the "normal" programmer.

There are a few situations in which red cuts are needed. The most prominent one is the *cut-fail* combination, as used in the definition of *negation-as-failure*. In order to avoid a collision with the predefined `not/1` predicate we define our own negation predicate:

```
% neg(G) :- the goal G cannot be proven
```

The potential danger of *negation-as-failure* can be demonstrated with a simple example:

```
unmarried_student(X) :- neg(married(X)), student(X).

student(romulus).

married(remus).


?- unmarried_student(X).
```

It should be noted that the built-in predicate `not/1` (or `\+/1`) is defined in a similar way and gives the same incorrect result as our private negation predicate `neg/1`.

As a general rule, in order to prevent this type of errors, care should be taken that there are *no uninstantiated variables* inside a negated goal. In our example, the problem can be solved with the clause:

```
unmarried_student(X) :- student(X), neg(married(X)).
```

Our example shows, that a cut, even "hidden" in another predicate, can have an effect on the correct goal order in a clause.

Our last case shows, how the cut can be used to turn Prolog into an ordinary procedural programming language:

```
% if_then_else(C,T,E) :- if C then T else E
```

We conclude this section with three rules of programming style:

- Don't use cuts.
- Don't use cuts, except when you are sure you need them.
- Don't use cuts, except perhaps *green ones*, to improve space and/or runtime efficiency. Green cuts *express determinism* and are *added to correct logic programs.*

The Prolog Program Database

Prolog offers the possibility that a running program modifies itself. Facts and rules can be added and removed while a program is running. This feature is controversial. Some people argue that programs and data should be strictly separated. On the other hand, some problems of *artificial intelligence* seem to require this feature.

Clauses can be added to a program using one of the `assert/1` predicates:

A typical way of using assert/1 is to store derived solutions for later use. This technique is sometimes called *caching*.

Let us reconsider the primitive version of the Fibonacci predicate:

```
% fib(N,F) :- F is the N'th Fibonacci number

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

We already know that this solution is inefficient, because intermediate results are calculated and thrown away. If they were memorized, the program could be much more efficient. In the following version of the program, we store all the known Fibonacci numbers in a database predicate `fibo/2`:

```
% fibo(N,F) :- F is the known N'th Fibonacci number
```

```
% fib(N,F) :- F is the N'th Fibonacci number
```

We can demonstrate the effect of clause assertion with the following goal sequence:

```
?- listing(fibo).

?- fib(20,F).

?- listing(fibo).
```

Clauses can be removed from the program database with the `retract/1` and `retractall/1` predicates. In our example, the predicate `fibo/2` can be reinitialized to its original state with the following predicate:

```
init_fibo :-
   retractall(fibo(_,_)),
   assert(fibo(0,1)),
   assert(fibo(1,1)).
```

In the Fibonacci example, the program stores new facts, i.e. *ground terms* during execution. However, self-modifying Prolog programs can be much more sophisticated, but also more risky! An example is the following solution of the famous problem of the Towers of Hanoi.

```
% hanoi(N,A,B,C,Ms) :- Ms is the sequence of moves required
%    to move N discs from peg A to peg B using peg C as an
%    intermediary according to the rules of the Towers of
%    Hanoi puzzle.
```

The way `hanoi/5` works can best be understood by studying the following goal sequence:

```
% ?- listing(hanoi).

% ?- time(do_hanoi(5,[a,b,c],Ms)).

% ?- listing(hanoi).

% ?- time(do_hanoi(5,[1,2,3],Ms)).
```

It should be mentioned, once again, that many computer scientists think that self-modifying programs are a dangerous and archane concept - *dirty tricks* - that should be avoided.

Loops and Extra-Logical Predicates

There is a class of predicates in Prolog that lie outside the logic programming model, and are called *extra-logical* predicates. These predicates achieve a side effect in the course of being satisfied as a logical goal. Input/output predicates are the most prominent group of extra-logical predicates.

In general, Prolog implementations provide a large variety of different forms of I/O predicates. We are not going to discuss the technical subtleties here. Consult the manual of your Prolog implementation for details.

The basic predicate for input is `read/1`. The goal `read(X)` reads a *term* from the current input stream, usually from the terminal, and unifies it with `X`. `read(X)` succeeds if this unification is successful, otherwise it fails. Normally, X is a variable argument; then read(X) always succeeds (provided there is no syntax error in the term read from the input stream).

The basic predicate for output is `write/1`. The goal `write(X)` prints the *term* X on the current output stream, usually the terminal (screen). `write(X)` always succeeds.

Neither `read(X)` nor `write(X)` nor any other I/O predicate give alternative solutions on backtracking; i.e. they normally succeed *once*, but never more than once.

The following is a first simple example for basic I/O:

```
print_father :-
   write('Whose father are you looking for:'),
   read(Child),
   find_father(Father,Child),
   writeln(['The father of ', Child, ' is ', Father,'.']).

find_father(F,C) :- father(F,C), !.
find_father(unknown,_).

writeln([X|Xs]) :- write(X), writeln(Xs).
writeln([]) :- nl.
```

Prolog allows for I/O redirection using *streams*. See the descriptions of the predefined predicates `see, tell, seen, told, seeing, telling,` and maybe others.

Furthermore, there are predicates for *simple character I/O*, for more sophisticated term I/O, and, depending on the implementation, for other forms of *formated I/O* and *file I/O* and for interaction with the underlying *operating system*.

Our example `print_father/0`, above, uses a useful utility predicate `writeln/1`, analogous to the Pascal command. (Actually, in SWI-Prolog, we could have used `writef/2`, instead.)

`writeln/1` shows a first form of program loop: *tail-recursion.*

We can construct interactive loops based on the idea of tail-recursion. The technique is well-established in procedural programming, under the name *'look-ahead read'* The following example shows the method. Suppose, in the above example, we want to give the user the opportunity to find the father of more than one person.

```
print_fathers :-
   write('Fathers. Type done to stop the loop.'), nl,
   get_person(P), print_father(P).

print_father(done) :- !.
print_father(C) :-
   find_father(F,C),
   writef('The father of %w is %w.\n',[C,F]), !,
   get_person(P), print_father(P).

get_person(P) :-
   nl, write('Whose father are you looking for:'), read(P).
```

The general scheme for tail-recursion based loops can be formulated generically as follows:

Due to the cuts in the above `do_/1` predicate, the program is iterative and deterministic. It can be run efficiently on a system with tail recursion optimization, always using the same small amount of stack space. Note that our generic predicate `get_/1` must provide the atom `done_` at the end of the loop.

Tail-recursion based loops are, in some sense, the equivalent of *while loops* in other programming languages.

In Prolog, there is another method to program loops, which is frequently used in situations, where multiple solutions are found using backtracking. These loops are called *failure-driven loops*, and they correspond, more or less, to *repeat loops* in other programming languages. Failure-driven loops are useful only when used in conjunction with extra-logical predicates with side effects, e.g. I/O predicates.

As an example, let us study a program to print the (known) grandparents of a given person.

```
print_grand_parents :-
   write('Whose grand-parents are you looking for:'),
   read(Person),
   print_grant_parents(Person).

print_grant_parents(P) :-
   grand_parent(G,P),
   writef('%w is a grand-parent of %w.\n',[G,P]),
   fail.
print_grant_parents(P) :-
   writef('%w has no (more) known grand-parents.\n',[P]).
```

The repetition in failure-driven loops is due to a predicate, in our example `grand_parent/2`, which produces several solutions using backtracking.

There is a funny predicate, `repeat/0`, which does nothing else than producing an unlimited number of alternative solutions by backtracking. It is defined as follows:

`repeat/0` must be used with a goal that plays the role of an 'until' statement in other programming languages. The following example demonstrates the technique:

```
do_squares :-
   write('Squares. Enter 0 to stop the loop.'), nl,
   repeat,
      write('x   ='), read(X),
      Y is X*X,
      write('x^2 = '), write(Y), nl,
   X = 0, !.
```

In some Prolog systems, failure-driven interactive loops are more efficiently implemented than tail-recursion based loops. This may be important for cases like mouse event responses and the like.

We conclude this discussion about loops with a construct similar to a *for-loop* in other programming languages. The following predicate generates all integers in a given interval by backtracking. In many implementations, the predicate is predefined.

```
% between(Low,High,Value) :- Low and High are integers,
%    High >= Low. If Value is bound to an integer,
%    Low <= Value <= High. When Value is a variable it is
%    successively bound to all integers from Low to High.
```

Using this predicate, we can write a program that prints all prime numbers of a given interval:

```
% primes(L,H) :- print the prime numbers in L..H
```

We have shown that failure-driven loops can be used to collect alternative solutions created by backtracking. We can, for instance, write the solutions to the terminal (or a file). However, with the techniques we have seen so far, we cannot put the solutions in a list or another data

structure. The built-in predicates `bagof/3`, `setof/3`, and `findall/3` serve this purpose. Their use is simple. Follow the manual of your implementation.

If, for some reason, it is not possible to use one of these built-in predicates, the *assert/retract* mechanism can be used to save results in a "global" fashion. The method can be explained with the example already introduced in chapter 3. There, we had defined the following predicates, in order to find the biggest country:

```
% bigger_than(A) :- there is a country bigger than A sq-km

bigger_than(A) :- area(_,B), B > A.

% biggest(C,A) :- C is the biggest country, its area is A

biggest(C,A) :- area(C,A), \+ bigger_than(A).
```

The disadvantage of this solution is that, in general, `area/2` is called many times for the same country. If `area/2` was not a simple database predicate, but a complicated calculation, the predicate `biggest/2` would be very inefficient. Note that `area/2` generates multiple results by backtracking, and we need to compare the results in order to compute the maximum. The following is a solution based on a global database predicate `leader/2`:

```
% biggest(C,A) :- C is the biggest country, its area is A

biggest(_,_) :-
   init_leader,
   area(C,A),
      update_leader(C,A),
   fail.
biggest(C,A) :- leader(C,A).

init_leader :-
   retractall(leader(_,_)),
   area(C,A), !,
   assert(leader(C,A)).

update_leader(C,A) :-
   leader(_,M),
   (A =< M, !
   ;
   retract(leader(_,_)),
   assert(leader(C,A))).
```

In addition to the *assert/retract* method, some Prolog systems offer other mechanisms to store "global" variables. Check your user's manual for details.

Exercises

(5.1) This is a simple variant of the *Knapsack Problem*, a famous combinatorial problem. Given a set S of *positive* integer numbers and a number A. Find the subset U of S such that A is the sum of the numbers in U. Try to solve the problem with a generate-and-test method. Find a way to "push" the tester inside the generator in order to make the solution more efficient. The solution should be a predicate, defined as follows:

```
% knapsack(S,A,U) :- U is a subset of S with the sum A
% (integer-list,integer,integer-list) (i,i,o)
```

Try, for example, the following goal:

```
?- knapsack([45,34,21,76,43,23,47,121,59,93],481,U)
```

(5.2) Based on the Utopia program developed in the course, write a tail-recusive loop predicate, specified as follows:

```
% utopia(Lower,Upper) :- print the decomposition(s) of
%    all integer values between Lower and Upper limit.
```

The output on the screen should look like this:

```
?- utopia(15,20).
15 = [1,1,1,12]
16 = [4,12] = [1,1,1,1,12] = [4,4,4,4]
17 = [1,4,12] = [1,4,4,4,4]
18 = [1,1,4,12]
19 = [1,1,1,4,12]
20 = [4,4,12] = [4,4,4,4,4]
```

(5.3) Write the predicate specified as follows:

```
% no_utopia(Lower,Upper,Ls) :- Ls is the list of all
%    amounts between Lower and Upper limit which do not
%    have a legal decomposition
```

Example:

```
?- no_utopia(50,80,Ls).
Ls = [72,73,74,77,80]
```

Hint: If you use the `setof/3` predicate then your program is a one-liner!

(5.4) *Processor scheduling*. Suppose that P identical processors can process jobs waiting in a job queue.Whenever a processor is finished with its job, it takes the next job from the queue, until the queue gets empty. We assume that the jobs are independent from each other, i.e. their order doesn't matter for the result of the computation. However, their order *does* matter for the time S until the last job is done. For example, if Js=[25,20,11,17,33,14,8,12,15] is the list of job execution times for a system with P=3 parallel processors, then an ideal job sequence is [33,25,20,17,15,11,14,12,8], with S=52. Write a predicte to find out an ideal job sequence:

```
% schedule(Js,P,Is,S) :- given the job execution times Js
%    and the number of processors P, Is is an ideal job
%    sequence and S is the time when the last job finishes.
% (number-list,integer,number-list,number) (i,i,o,o)
```

Try a pure generate-and-test method first. Then, preclude permutations which lead to a total time larger than what has been found as optimum so far. This is not an easy exercise.

Summary

- Many combinatorial problems can be solved using *generate-and-test* methods. Shifting the tester process "inside" the generator process is a standard optimization technique.

- Prolog programs can be debugged usig a *four-port tracing* model. Program control is monitored at four ports: CALL, EXIT, REDO, and FAIL.

- Backtracking is controlled with a system predicate, called the *cut*, denoted ***!***. The goal **!** *succeeds* and *commits* Prolog to all the choices made since the parent goal was unified with the clause the cut occurs in.

- *Green cuts* express determinism, while *red cuts* are used to omit explicit conditions. Good programming style forbids the use of red cuts in ordinary programs. Green cuts can be used to improve stack space and/or runtime efficiency.

- *Negation-as-failure* is the implementation of the *not* predicate in Prolog. The not predicate can never be used to instantiate a variable.

- In Prolog it is possible that a running program modifies itself. The predefined predicates *assert* and *retract* are used for this purpose.

- Input/output to the terminal and/or to a file system is achieved by *extra-logical* predicates. Terms, even very complicated compound terms, can be written and read at once. I/O stream redirection makes it easy to perform input/output from and to files.

- Loops can be programmed in Prolog in two ways: *tail-recursion* based or *failure-driven*. The former is the equivalent of *while loops* with look-ahead read in other programming languages, while the latter correspond to *repeat-until* loops. A predicate *between/3* helps in implementing *for-loops* as failure-driven loops.

6. Advanced Topics

This chapter introduces two advanced programming techniques: *incomplete data structures* and, a particularly useful special case thereof, *difference-lists*.

Incomplete Data Structures

Let us start with a problem from recreational mathematics:

Problem: Thee friends came first, second and third in a programming competition. Each of them had a different first name, liked a different sport, and had a different nationality. Michael likes basketball, and did better than the American. Simon, the Swiss, did better than the tennis player. The cricket player came first. Who is the Australian? What sport does Richard play?

We are going to represent each friend as a term

The scores table is a list of friends, the head being the winner of the competition.

We can define some auxiliary predicates:

```
fname(fr(Name,_,_),Name).

sport(fr(_,Sport,_),Sport).

natio(fr(_,_,Natio),Natio).

first(X,[X|_]).

better(X,Y,[X,Y,_]).
better(X,Y,[X,_,Y]).
better(X,Y,[_,X,Y]).

element(X,[X|_]).
element(X,[_|Xs]) :- element(X,Xs).
```

It is now easy to translate the problem description into a Prolog program:

```
friends :-
   fname(A,'Michael'), sport(A,'basketball'),
   better(A,B,Fs),
   natio(B,'American'),
   fname(C,'Simon'), natio(C,'Swiss'),
   better(C,D,Fs),
   sport(D,'tennis'),
   sport(E,'cricket'), first(E,Fs),
   element(X,Fs), natio(X,'Australian'), fname(X,XName),
   element(Y,Fs), fname(Y,'Richard'), sport(Y,YSport),
   write(XName), write(' is the Australian.'), nl,
   write('Richard plays '), write(YSport), write('.'), nl.
```

The above predicate is a good example for an almost pure declarative program. But, what exactly happens when it is executed by a Prolog interpreter? We can use program tracing to understand the predicate in a step-by-step manner.

Two important things happen right at the beginning of the program:

```
fname(A,'Michael')

better(A,B,Fs)
```

The above goals create terms *(data structures)* that are only *partially instantiated*. The goals further down the program fill in more and more information, until there are only two variables left in the scores table: a nationality and a first name. These are then instantiated by the "queries" *Who is the Australian?* and *What sport does Richard play?*

This technique of *partially instantiated data structures* or *incomplete data structures*, can be used to solve many of the classical problems of *artificial intelligence*.

Managing a Pool of Variables

An interesting variant of the well-known `element/2` (or `member/2`) predicate is the following:

```
el(X,[X|_]) :- !.
el(X,[_|Xs]) :- el(X,Xs).
```

Find out what the following goal does:

```
?- el(alfa,L), el(bravo,L), el(charlie,L), el(bravo,L).

L =
```

Apparently, the list `L` in the above example is *open-ended*, i.e. we can add further elements to it using the `el/2` predicate. This behaviour can be exploited to manage a pool of Prolog-like variables. We define a variable, in our sense, as a term

```
v(Name,Value)                Example: v(alfa,25)
```

where `Name` is an atom, the name of the variable, and `Value` denotes an integer, at least in our current example. Observe what happens, when the following conjunctive goal is called:

```
?- el(v(alfa,25),E), el(v(beta,B),E), el(v(gamma,33),E),
   el(v(alfa,A),E), el(v(beta,C),E), el(v(beta,20),E).

E =
```

The example shows, how we can "define" a variable for the "environment" E whether or not an actual value is already known. It also demonstrates how we can "assign" a value to a variable defined earlier, and how we can "look up" the value of a such a variable. However, there is a problem. Consider the following goal:

```
?- el(v(alfa,25),E), el(v(alfa,A),E), el(v(alfa,33),E).
```

The result is not exactly what we had intended. Alfa should not be allowed to enter twice into the environment E. We can solve this problem easily with the following predicate:

```
bind0(Name,Value,Env) :- el(v(Name,V),E), !, V = Value.
```

Now, consider the goal:

```
?- bind0(alfa,25,E), bind0(alfa,A,E), bind0(alfa,33,E).
```

We now suppose that, for some reason, it is not allowed, that the same value is assigned to more than one variable. This is, of course, not the normal behaviour of Prolog-like variables. But, it is appropriate for some mathematical puzzles, where different symbols stand for different values. We shall further assume that the values that can be assigned to the variables are single digit integer numbers. The additional properties can be guaranteed with the following very powerful predicate:

```
bind(Name,Val,Env) :-
   el(v(Name,V),Env), nonvar(V), !, V = Val.
bind(Name,Val,Env) :-
   between(0,9,Val), el(v(N,Val),Env), N = Name.
```

We can test the `bind/3` predicate with the following queries:

```
% ?- bind(a,1,E), bind(b,2,E), bind(a,A,E).
% ?- bind(a,1,E), bind(b,2,E), bind(a,1,E).
% ?- bind(a,1,E), bind(b,2,E), bind(a,3,E).
% ?- bind(a,1,E), bind(b,2,E), bind(c,5,E).
% ?- bind(a,1,E), bind(b,2,E), bind(c,2,E).
% ?- bind(a,1,E), bind(b,2,E), bind(c,X,E), X < 7.
```

The following program is a demonstration for the use of the `bind/3` predicate. It solves a class of mathematical puzzles, known as cryptarithmetic. Example: SEND+MORE=MONEY.

The main program is the following predicate:

```
% puzzle(S1,S2,S3) :- solve a mathematical puzzle of the
%    class S1 + S2 = S3, where Si denote strings of
%    characters which must be replaced by digits.

puzzle(S1,S2,S3) :-
   str_rev_char_list(S1,L1),
   str_rev_char_list(S2,L2),
   str_rev_char_list(S3,L3),
   help(S1,S2,S3,E),
   plus_list(L1,L2,0,L3,E),
   write_solution(S1,S2,S3,E),
   fail.
puzzle(_,_,_).
```

Note the failure-driven loop we use in order to find all solutions, if there is more than one.

As the addition has to be performed from right to left, and lists are easily accessible from left to right, we first reverse the order of the characters in the given strings. This is done by the predicate `str_rev_char_list/2` which uses two predefined predicates.

```
% str_rev_char_list(S,L) :- L is the list of the characters
%   (ASCII codes) of the string S in reverse order.

str_rev_char_list(S,L) :-
   string_to_list(S,L1), reverse(L1,L).
```

The heart of the program is the predicate `plus_list/5`, which actually "does" the addition:

```
% plus_list(L1,L2,Carry,L3,Env) :- assign digits to the
%    characters in the lists L1, L2, and L3, such that a
%    correct (left-right mirrored) addition scheme results.
%    Store the assignments in Env. Carry (0 or 1) must be
%    added to the leftmost position.

plus_list([],[],0,[],_) :- !.
plus_list([],[],1,[Ch],E) :- !, bind(Ch,1,E).
plus_list([],[Y|Ys],C_in,[Z|Zs],E) :- !,
   bind(Y,YVal,E),
   plus(0,YVal,C_in,ZVal,C_out),
   bind(Z,ZVal,E),
   plus_list([],Ys,C_out,Zs,E).
plus_list([X|Xs],[],C_in,[Z|Zs],E) :- !,
   bind(X,XVal,E),
   plus(XVal,0,C_in,ZVal,C_out),
   bind(Z,ZVal,E),
   plus_list(Xs,[],C_out,Zs,E).
plus_list([X|Xs],[Y|Ys],C_in,[Z|Zs],E) :-
   bind(X,XVal,E), bind(Y,YVal,E),
   plus(XVal,YVal,C_in,ZVal,C_out),
   bind(Z,ZVal,E),
   plus_list(Xs,Ys,C_out,Zs,E).

plus(A,B,C_in,S,C_out) :- Sum is A + B + C_in,
   S is Sum mod 10, C_out is Sum // 10.
```

Finally, an auxiliary predicate `help/4` makes the program more efficient if the result string S3 is longer than the operand strings S1 and S2.

```
% help(S1,S2,S3,E) binds the most significant digit of the
%    result string S3 to 1 if both operands S1 and S2 are
%    shorter than the result S3. The predicate is used for
%    performance improvement only.

help(S1,S2,S3,E) :-
   atom_length(S1,L1),
   atom_length(S2,L2),
   atom_length(S3,L3),
   L3 > L1, L3 > L2, !,
   name(S3,[C|_]), bind(C,1,E).
help(_,_,_,_).
```

The remainder of the program deals with output formatting, and is not discussed here.

Difference-lists

In chapter 4 we have developed a predicate which concatenates two lists:

```
% conc(X,Y,Z) :- the list Z is the result of concatenating
%    the list X and the list Y

conc([],Ys,Ys).
conc([X|Xs],Ys,[X|Zs]) :- conc(Xs,Ys,Zs).
```

Although elegant, the predicate `conc/3` is not very efficient. Consider, for example, the goal

```
?- conc([1,2,3,4],[5,6,7],L)
```

The first list ([1,2,3,4]) must be completely traversed, element by element. This is the same situation as in procedural programming languages, where lists are represented as records, linked by pointers:

When concatenation is a frequent operation it is a common practice in procedural languages to use a *tail-pointer* in addition to the usual *anchor pointer*.

In Prolog, evidently, there are no explicit pointers. However, in some sense, the idea of a tail-pointer has been adapted.The concept is known under the name *difference-lists*.

Consider the following lists:

```
F =  [2,  3,  5,  7,  11,  13,  15,  17]

G =                  [11,  13,  15,  17]

H =                              [15,  17]
```

We define the following *sequences* as *difference-lists:*

```
A:  2 - 3 - 5 - 7              A =

B:  11 - 13                    B =
```

It is now almost trivial that the concatenation of the sequences A and B becomes

```
C: 2 - 3 - 5 - 7 - 11 - 13     C =
```

The given concatenation example works fine because `G` occurs both in A and B in an appropriate way. We define:

Two difference lists `R-S` and `U-V` are called *compatible* if and only if `S = U`.

For compatible difference-list there is a simple rule for concatenation:

```
% conca(A,B,C) :- C is the concatenation of the two
%    compatible difference-lists A and B
```

Of course, one could argue that the above predicate is not very useful because it is only applicable for *compatible* difference-lists. However, the art of Prolog programming is to choose the instantiation pattern in such a way that the difference-list *become* compatible during the concatenation. We shall demonstrate this below.

Let us first develop a predicate that performs the transformation of ordinary lists into difference-list, and vice versa.

```
% list_dl(L,D) :- D is the difference list representation
%    of the list L.
```

We observe the execution of the following program:

```
?- list_dl([1,2,3],D1),

list_dl([4,5],D2),

conca(D1,D2,D3),

D3 = L3 - [].
```

Unfortunately, the above definition for `list_dl/2` doesn't work for certain instantiation patterns. For example, the goal

```
?- list_dl(L,[a,b,c|R]-R).
```

should return `L = [a,b,c]`. Instead of doing this, the SWI-Prolog system crashes, because the interpreter falls into an infinite instantiation loop. The reason is that the unification algorithm apparently does not perform the so-called *occurs check*. You can check whether your Prolog system does execute the occurs check by calling the following goal:

```
?- X = alfa(beta,X).
```

If the system returns 'No' then it does the occurs check. Otherwise you will probably need to restart the Prolog system (or even reboot your computer).

The following is a version of the `list_dl/2` predicate that works even for the above mentioned instantiation pattern:

```
% list_dl(L,D) :- D is the difference-list representation
%    of the list L. A better solution

list_dl(L0,T-T) :- L0 == [], !.                 % for (i,*)
list_dl([],T1-T2) :- T1 == T2, !.               % for (o,i)
list_dl([X|Xs],[X|Zs]-T) :- list_dl(Xs,Zs-T).   % for both
```

Most interestingly, and ironicly, the clever `conca/3` predicate is almost never explicitely used in practice. In most applications, the concatenation is done implicitely - hidden in the code of another predicate.

A simple example is the solution of the Hanoi puzzle:

```
% hanoi(N,A,B,C,Ms) :- Ms is the sequence of moves required
%    to move N discs from peg A to peg B using peg C as an
%    intermediary according to the rules of the Towers of
%    Hanoi puzzle.
```

A typical situation where difference-lists are useful, is the implementation of a FIFO queue. Such a data structure is needed, for example, if a tree has to be traversed in *breadth-first order* (or *level-order).*

Let us suppose we represent binary trees in the way defined in chapte 4, i.e. as terms t(X,L,R), where X stands for a value stored in a node, and L and R represents left and right subtree, respectively. A program that traverses a binary tree in level-order and prints the sequence of nodes to the screen, is the following:

```
% levelorder(BinTree) :- print the nodes of BinTree in
%    level-order sequence.
```

Another occasion where difference-lists are extremely useful is *string parsing*. Let us return to our string representation of multiway trees, explained in chapter 4 (see pages 34ff). Suppose we have the following string:

```
a{b{c,d,e},f{g},h{j,k}}
```

How can we construct the corresponding term `t(Root,Forest)` that represents the tree, according to our convention? In order to develop a solution we need again the syntax diagrams (see page 35):

We first transform the string (actually a Prolog atom) into a list of characters (ASCII values, and then call a predicate `p_tree/2` that parses a tree, according to the above syntax diagram:

```
% tree(S,T) :- T is the tree represented by the atom S

tree(S,T) :- name(S,L), p_tree(L-[],T).

p_tree(L1-L6,t(X,[T1|Ts])) :-
   p_root(L1-L2,X),
   p_symbol(L2-L3,'{'), !,
   p_tree(L3-L4,T1),
   p_forest(L4-L5,Ts),
   p_symbol(L5-L6,'}').
p_tree(L1-L2,t(X,[])) :-
   p_root(L1-L2,X).

p_forest(L1-L4,[T1|Ts]) :-
   p_symbol(L1-L2,','), !,
   p_tree(L2-L3,T1),
   p_forest(L3-L4,Ts).
p_forest(L-L,[]).

p_root([C|L]-L,X) :- is_letter(C), name(X,[C]).

is_letter(C) :- between(97,122,C), !.   % a - z
is_letter(C) :- between(65,90,C).       % A - Z

p_symbol([C|L]-L,X) :- name(X,[C]).
```

Note: The built-in predicate `name(A,L)` transforms an atom A into a list L of ASCII values or vice versa. Example: `?- name(alfa,[97,108,102,97]). -> Yes`.

A Tiny German-to-French Translator

We conclude this chapter with a more elaborate program to demonstrate how difference-lists can be used in *natural language processing*. It is our objective to write a program to translate very simple sentences of a particular type (in perfect tense) from German to French.

Here are some examples:

German: `Der Hund hat die Katze gebissen.`

French: `Le chien a mordu le chat.`

German: `Du hast die Zeitungen nicht gelesen.`

French: `Tu n' as pas lu les journaux.`

Evidently, there are a number of difficult problems with these translations. One of them is the *order of the sentence parts* (subject, auxiliary verb, object) which is different in the two languages.

In Prolog, we can use difference-lists in order to solve this problem quite elegantly. The following is the heart of the translator program:

```
translate(GL,FS) :-
   maplist(decapitalize,GL,GLD),
   t_phrase(GLD-['.'],FL-['.']),
   polish(FL,[F1|Fs]),
   capitalize(F1,F2),
   concat_atom([F2|Fs],FS).


t_phrase(G0-G4,F0-F4) :-
   t_subj(G0-G1,F0-F1,PersonNumber),
   t_auxv(G1-G2,F1-F2,PersonNumber),
   t_acc_obj(G2-G3,F3-F4),
   t_part(G3-G4,F2-F3), !.
t_phrase(G0-G5,F0-F6) :-
   t_subj(G0-G1,F0-F1,PersonNumber),
   t_auxv(G1-G2,F2-F3,PersonNumber),
   t_acc_obj(G2-G3,F5-F6),
   t_neg(G3-G4,F1-F2,F3-F4),
   t_part(G4-G5,F4-F5), !.
```

The following is the complete Prolog code of the translator program.

```
% German to french translator fragment    werner.hett@hta-bi.bfh.ch)
% 1992 (Turbo-Prolog)  7-Dec-1993 (adapted to SWI-Prolog)
% 13-Feb-2002 completely rewritten for SWI-Prolog

% The translator can translate simple sentences in perfect tense,
% e.g. "Der Hund hat die Katze gebissen.", "Martin hat die Zeitungen
% nicht gelesen.", or "Die Kinder haben Pferde gesehen."

% Start the translator by calling the predicate go.
% Stop with an empty line.

% The translator is a good example for the use of difference-lists.

translate(GL,FS) :-               % GL is the list of the german words
   maplist(decapitalize,GL,GLD),  % make all words lower-case
   t_phrase(GLD-['.'],FL-['.']),  % do the translation
   polish(FL,[F1|Fs]),            % insert apostrophes, e.g. j'ai ...
   capitalize(F1,F2),             % make first french word upper-case
   concat_atom([F2|Fs],FS).       % convert word list into a string

t_phrase(G0-G4,F0-F4) :-
   t_subj(G0-G1,F0-F1,PersonNumber),
   t_auxv(G1-G2,F1-F2,PersonNumber),
   t_acc_obj(G2-G3,F3-F4),
   t_part(G3-G4,F2-F3), !.
t_phrase(G0-G5,F0-F6) :-
   t_subj(G0-G1,F0-F1,PersonNumber),
   t_auxv(G1-G2,F2-F3,PersonNumber),
   t_acc_obj(G2-G3,F5-F6),
   t_neg(G3-G4,F1-F2,F3-F4),
   t_part(G4-G5,F4-F5), !.

t_subj(['ich'|GR]-GR,['je'|FR]-FR,1) :- !.
t_subj(['du'|GR]-GR,['tu'|FR]-FR,2) :- !.
t_subj(['er'|GR]-GR,['il'|FR]-FR,3) :- !.
t_subj(['sie'|GR]-GR,['elle'|FR]-FR,3).
t_subj(['wir'|GR]-GR,['nous'|FR]-FR,4) :- !.
t_subj(['ihr'|GR]-GR,['vous'|FR]-FR,5) :- !.
t_subj(['sie'|GR]-GR,['ils'|FR]-FR,6) :- !.
t_subj(G0-G2,F0-F2,3) :-
   g_art(G0-G1,Det,singular,GenderG),
   t_noun(G1-G2,F1-F2,singular,GenderG,GenderF),
   f_art(F0-F1,Det,singular,GenderF), !.
t_subj(G0-G2,F0-F2,6) :-
   g_art(G0-G1,Det,plural,GenderG),
   t_noun(G1-G2,F1-F2,plural,GenderG,GenderF),
   f_art(F0-F1,Det,plural,GenderF), !.
t_subj([X|GR]-GR,[X|FR]-FR,3).  % proper names (not translated)

g_art(['der'|GR]-GR,det,singular,masc).
g_art(['die'|GR]-GR,det,singular,fem).
g_art(['das'|GR]-GR,det,singular,neut).
g_art(['den'|GR]-GR,det,singular,masc).
g_art(['ein'|GR]-GR,nondet,singular,masc).
g_art(['ein'|GR]-GR,nondet,singular,neut).
g_art(['eine'|GR]-GR,nondet,singular,fem).
```

```
g_art(['einen'|GR]-GR,nondet,singular,masc).
g_art(['die'|GR]-GR,det,plural,masc).
g_art(['die'|GR]-GR,det,plural,fem).
g_art(['die'|GR]-GR,det,plural,neut).
g_art(G-G,nondet,plural,masc).
g_art(G-G,nondet,plural,fem).
g_art(G-G,nondet,plural,neut).

f_art(['le'|FR]-FR,det,singular,masc).
f_art(['la'|FR]-FR,det,singular,fem).
f_art(['les'|FR]-FR,det,plural,_).
f_art(['un'|FR]-FR,nondet,singular,masc).
f_art(['une'|FR]-FR,nondet,singular,fem).
f_art(['des'|FR]-FR,nondet,plural,_).

t_noun([G|GR]-GR,[F|FR]-FR,singular,GenderG,GenderF) :-
   dict_noun(G,_,GenderG,F,_,GenderF), !.
t_noun([G|GR]-GR,[F|FR]-FR,plural,GenderG,GenderF) :-
   dict_noun(_,G,GenderG,_,F,GenderF), !.

t_auxv(['habe'|GR]-GR,['ai'|FR]-FR,1).
t_auxv(['hast'|GR]-GR,['as'|FR]-FR,2).
t_auxv(['hat'|GR]-GR,['a'|FR]-FR,3).
t_auxv(['haben'|GR]-GR,['avons'|FR]-FR,4).
t_auxv(['habt'|GR]-GR,['avez'|FR]-FR,5).
t_auxv(['haben'|GR]-GR,['ont'|FR]-FR,6).

t_acc_obj(G0-G2,F0-F2) :-
   g_art(G0-G1,Det,Number,GenderG),
   t_noun(G1-G2,F1-F2,Number,GenderG,GenderF),
   f_art(F0-F1,Det,Number,GenderF), !.
t_acc_obj([X|GR]-GR,[X|FR]-FR).  % proper names (not translated)

t_neg(['nicht'|GR]-GR,['ne'|FR1]-FR1,['pas'|FR2]-FR2).

t_part([G|GR]-GR,[F|FR]-FR) :- dict_verb(G,F), !.

% The following are some auxiliary predicates

% capitalize(W1,W2) :- make the first letter upper-case
capitalize(W1,W2) :-
   atom_chars(W1,[X|Xs]), char_type(Y,to_upper(X)),
   atom_chars(W2,[Y|Xs]).

% decapitalize(W1,W2) :- make the first letter lower-case
decapitalize(W1,W2) :-
   atom_chars(W1,[X|Xs]), char_type(Y,to_lower(X)),
   atom_chars(W2,[Y|Xs]).

% polish(FL1,FL2) :- insert apostrophes in cases like j'ai
polish([X,'.'],[X,'.']) :- !.
polish([X1,X2|Xs],[X1M|Ys]) :-
   atom_chars(X1,X1L), last(C1,X1L), a_or_e(C1),
   atom_chars(X2,[C2|_]), vowel(C2), !,
   trailing_quote(X1L,X1ML),
   atom_chars(X1M,X1ML),
   polish([X2|Xs],Ys).
polish([X|Xs],[X,' '|Ys]) :- polish(Xs,Ys).
```

```
a_or_e('a').
a_or_e('e').

vowel('a').
vowel('e').
vowel('i').
vowel('o').
vowel('u').
vowel('y').
vowel('h').  % not always correct !

trailing_quote([_],['\'']) :- !.
trailing_quote([X|Xs],[X|Ys]) :- trailing_quote(Xs,Ys).

% The following is a very primitive german to french dictionary

dict_noun('hund','hunde',masc,'chien','chiens',masc).
dict_noun('katze','katzen',fem,'chat','chats',masc).
dict_noun('pferd','pferde',neut,'cheval','chevaux',masc).
dict_noun('affe','affen',masc,'singe','singes',masc).
dict_noun('kind','kinder',neut,'enfant','enfants',masc).
dict_noun('frau','frauen',fem,'femme','femmes',fem).
dict_noun('freund','freunde',masc,'ami','amis',masc).
dict_noun('zeitung','zeitungen',fem,'journal','journaux',masc).
dict_noun('brot','brote',neut,'pain','pains',masc).
dict_noun('wasser','wasser',neut,'eau','eaux',fem).
dict_noun('wein','weine',masc,'vin','vins',masc).
dict_noun('velo','velos',neut,'bicyclette','bicyclettes',fem).
dict_noun('auto','autos',neut,'bagnole','bagnoles',fem).
dict_noun('blume','blumen',fem,'fleur','fleurs',fem).

dict_verb('gelesen','lu').
dict_verb('gesehen','vu').
dict_verb('verkauft','vendu').
dict_verb('bestraft','punis').
dict_verb('gekocht','bouilli').
dict_verb('gefahren','conduit').
dict_verb('gemacht','fait').
dict_verb('genommen','pris').
dict_verb('verraten','trahi').
dict_verb('getrunken','bu').
dict_verb('gebissen','mordu').

do_translate([]) :- !.
do_translate(G) :- translate(G,F), !, write('French = '), write(F),
nl.
do_translate(_) :- !,
   write('I\'m sorry, I cannot translate that. Please try again.'),
nl.

go :-
   prompt1('German > '), readln(GS),
   do_translate(GS),
   GS \= [], go.
go.
```

Exercises

(6.1) *Zebra puzzle*: There are five houses, each of a different colour and inhabited by a man of a different nationality, with a different pet, drink, and brand of cigarettes.

1. The Englishman lives in the red house.
2. The Spaniard owns the dog.
3. Coffee is drunk in the green house.
4. The Ukrainian drinks tea.
5. The green house is immediately to the right of the ivory house.
6. The Winston smoker owns snails.
7. Kools are smoked in the yellow house.
8. Milk is drunk in the middle house.
9. The Norwegian lives in the first house on the left.
10. The Chesterfields smoker lives in the house next to the man with the fox.
11. Kools are smoked in the house next to the house where the horse is kept.
12. The Lucky Strike smoker drinks orange juice.
13. The Japanese smokes Parliaments.
14. The Norwegian lives next to the blue house.

Who owns the zebra? Who drinks water?

(6.2) Write a predicate `level_order/2` that constructs the level-order sequence of the nodes of a binary tree. Use the difference-list technique.

(6.3) Write a predicate `length_dl/2` with the following specification:

```
% length_dl(L1-L2,N) :- the difference list L1-L2 contains
%    N elements. The predicate is applicable for the
%    following instantiation patterns:
%
% ?- length_dl([a,b,c]-[],N).       ->  N = 3
% ?- length_dl([a,b,c|T]-T,N).      ->  N = 3, T = _
% ?- length_dl([a,b,c|T]-[c|T],N).  ->  N = 2, T = _
% ?- length_dl(S-T,3).        -> S = [_,_,_|T], T = _
```

(6.4) *Hamming's problem*. This is a classical programming problem: Determine all natural numbers, up to a given limit, that are multiples of 2, 3 and 5 *only*. In other words: Compute the natural numbers $2^q3^r5^s$ with $q,r,s \in N_0$, in ascending order. There is an elegant solution to the problem which makes use of three fifo queues, one for the multiples of 2, another one for the multiples of 3, and the third one for the multiples of 5. A step in the algorithm consists of removing the (globally) smallest number from the front of (some of) the queues and appending its multiples to all three queues. Use difference-lists to implement the fifo queues.

(6.5) Write a predicate `breadth_first/1` that prints out the breadth-first order sequence of the nodes of a given multiway tree. See chapter 4 for details of the term representing a tree.

(6.6) Let us suppose that the objects stored in the nodes of a binary tree are single lower-case letters. We can then represent the tree as a string, as shown in the following example: a(b,c(d,e(f(,g),))). Use syntax diagrams to develop a predicate that constructs the tree from its string representation.

(6.7) Find out what the following program does and how it works. Try to predict the behaviour of each predicate as exactly as possible, and then experiment with your Prolog interpreter.

```
% in(X,T) :- ???

in(X,t(X,_,_)) :- !.
in(X,t(Y,Left,_)) :- X @< Y, !, in(X,Left).
in(X,t(Y,_,Right)) :- X @> Y, in(X,Right).

% sh(T) :- ???

sh(T) :- var(T), !, write('_ ').
sh(t(X,Left,Right)) :-
   write(X), write(' '), sh(Left), sh(Right).

% do(T) :- ???

do(T) :- in(2,T),in(5,T),in(7,T),in(3,T),in(7,T),sh(T).
```

Suppose a data structure T has been constructed with a sequence of `in/2` calls. Write a predicate that prints all nonvar elements of T in ascending order.

Summary

- *Incomplete data structures* can be very useful for problems in which relations between objects are given (or determined) in an irregular, complicated pattern. Examples are some types of scheduling problems, where many different and complicated boundary conditions must be met.
- The idea of incomplete data structures lead to the powerful programming technique of *difference-lists*. This techique uses lists of elements, with the last element being a variable; i.e. a (yet) uninstantiated remainder list.
- Difference-lists can be used to implement *FIFO queues*.
- Another important application of difference-lists is string *parsing*.

Some Good Books

William F. Clocksin and Christopher S. Mellish, Programming in Prolog, Second Edition, Springer-Verlg, Heidelberg, 1984, ISBN 3-540-15011-0. (Prolog Standard)

Ivan Bratko, Prolog Programming for Artificial Intelligence, Second Edition, Addison-Wesley, Wokingham, England, 1993, ISBN 0-201-41606-9.

Leon Sterling and Ehud Shapiro, The Art of Prolog, Advanced Programming Techniques, The MIT Press, Cambridge, Mass., 1986, ISBN 0-262-69105-1.