Advanced Programming Languages

Authors

Tibor Ásványi
Ákos Balaskó
Iván József Balázs
Balázs Csizmazia
Péter Csontos
Szabina Fodor
Attila Góbi
Hajnalka Hegedűs†
Zoltán Horváth
András Juhász
Attila Kispitye
Tamás Kozsik
Lehel István Kovács D.
Richárd Legéndi
Tamás Marcinkovics
Attila Rajmund Nohl
Judit Nyéky-Gaizler
Gábor Páli
Zoltán Porkoláb
Gábor Pécsy
Máté Tejfel
Szabolcs Sergyán
Balázs Zaicsek
Viktória Zsók

Advanced Programming Languages

Editor
Judit Nyéky-Gaizler

The book is supported by the TÁMOP-4.1.2.A/11-1/1 project.

Editor:
Judit Nyéky-Gaizler

Layout editor: Attila Kispitye
Publisher: Eötvös Loránd University
ISBN: 978-963-284-450-3

Contents at a Glance

Table of Contents

Introduction

Programming languages are thought by many to provide as a notation form for program description. This view does not take into account – or does not even know –, how *high level* or *user-centered languages* can aid in managing program complexity. Different languages with their possibilities suggest different programming approaches, so the common practice, which is still used nowadays in many places, is highly dangerous, when programming methodology is taught through particular programming languages, not independently from them – this could only lead to narrow concerning all the programming possibilities.

The goal of programming is to produce a *good quality software product*, so the education of programming must start with the general definition of the task and its solving program [Fó83]. Then based on this principle, the different concrete language tools should be acquainted to the programmers, which support the implementation. However, as it is questionable to teach the methodology through particular concrete programming languages, it also leads to a dead end, if the used programming language is said to be not important for the sake of the methodology. This is – as described by Bertrand Meyer [Mey00] – like "a bird without wings". The idea is inseparable from the possibilities of formulation. It is not a coincidence that in programming no single language has become dominant, nor that always newer programming languages are designed, which support even more the adaptation of different methodological concepts and requirements into practice.

Designers of programming languages must deal with three problems [Hor94]:

- The representation provided by the language must fit the hardware and the software at the same time.
- The language must provide a good nomenclature for the description of algorithms.
- The language must serve as a tool to manage program complexity.

Aspects of software quality

The software is a product, and as for every product, it has – as defined by many ([Mey00], [Hor94] and [LG96]) – different quality characteristics and requirements. One of the most important goals of the programming methodology is to specify a theoretical approach for creating good quality program products. The design and the evaluation of already existing programming languages are definitely influenced by methodological considerations.

Next, characteristics of "good" software will be discussed according to the work of Bertrand Meyer [Mey00]. After that, language features will be examined for supporting the methodology – through numerous programming languages.

Software quality is influenced by many factors. One part of these – such as reliability, speed, or ease of use – are basically perceived by the user of the program. Others – such as how easy it is to reuse some parts of it for a different, but similar problem – affect program developers.

Correctness

Correctness of the program product means that the program solves exactly the problem and fits the desired specification. This is the first and most important criterion, since if a program is not working like it should, other requirements do not really count. The elementary basis for this is the precise and the most complete specification.

Reliability

A program is called *reliable* if it is correct, and abnormal – not described in the specification – circumstances do not lead to catastrophe, but are handled in some "reasonable" way.

This definition shows, that reliability is by far not as a precise notion as correctness. One could say, of course with a more specific specification reliability would mean correctness exactly, but in practice there are always cases which are not covered by specification explicitly. That is why reliability is of high priority for the program product quality.

Maintainability

Maintainability refers to how easy it is to adjust the program product to specification changes.

The users often demand further development, modification, adjustment of the program product to new external conditions. According to some surveys 70 % of program product costs are spent on maintenance, so it is understandable that this requirement significantly affects the quality of the program. (This is

relevant especially if developing big programs and program systems, since for small programs usually no change is too complex.)

To increase maintainability, design simplicity and decentralization (to have independent modules) can be seen as the two most important basic principles.

Reusability

Reusability is the feature of the software products, that they can be partly or as a whole reused in new applications.

This is different to maintainability, since the same specification was modified there, but now the experience should be utilized, that many elements of software systems follow common patterns, and reimplementing already solved problems should be avoided.

This question is particularly important, not only when producing individual program products, but for a global optimization of software development, as the more reusable components are available to help problem solving, the more energy remains to improve other quality characteristics (at the same costs).

Compatibility

Compatibility shows how easy it is to combine the software products with each other. Programs are not developed isolated, so efficiency can go up by orders of magnitude, if ready software can be simply connected to other systems. (Communication between programs is based on some standards, such as, for example, in Unix.)

Other characteristics

From the quality characteristics of the program product, portability, efficiency, user friendliness, testability, clarity etc. are also important to pay attention to.

Portability regards how easy it is to port the program to another machine, configuration or operating system – usually to have it run in different runtime environments.

The *efficiency* of a program is proportional to the running time and used memory size – the faster, or the less memory is used, the more efficient it is. (These requirements often contradict each other, a faster run is often set off by bigger memory requirements, and vice versa.)

The *user friendliness* is very important for the user: this requires data input to be logical and simple, the output of the results must be clearly formatted.

Testability and *clarity* are important for the developers and maintainers of the program, without these the reliability of the program cannot be guaranteed.

Aspects of software design

Some of these requirements – the improvement of correctness and reliability – require primarily the development of specification tools. The easier it is to verify if a piece of program code is really an implementation according to the specification, the easier it will be to developed correct and reliable programs. The main role here have programming language features for specification (type invariant, pre- and postconditions) descriptions – this is supported for example by Eiffel [Mey00], by Ada 2012 [NG+98] etc.

Implementation of another group of requirements – mainly maintainability, reusability and compatibility – can be best supported by designing the programs as independent program units having well defined interconnections. This is the basis of the so called *modular design*. (A module here is not a programming language concept, but a unit of the design.) This question will be handled in more detail in Chapter **??**.

Our goal is to examine the features of different programming languages to support professional programmers in developing reliable software of good quality.

Study of the tools of programming languages

It is a natural question, why it is not enough to know *one* programming language, for what purpose it is good to deal with all the possible features of different programming languages. In the following – primarily based on the work of Robert W. Sebesta [Seb13] – we will try to summarize the advantages coming from this:

Increase of the expressive power

Our thinking and even abstraction skills are strongly influenced by the possibilities of the language used. Only that can be expressed, for which there are words. Likewise during program development and designing the solution, the knowledge of diverse programming language features can help programmers to widen their horizon. This is also true if a particular language must be used, since good principles can be applied in any environments.

Choosing the appropriate programming language

Many programmers have learnt programming through one or two languages. Others know older languages which are now considered obsolete, and they are not familiar with the features of modern languages. This could result in not selecting the most appropriate language if there would be more programming languages as options to choose from for a new task – since they do not know the possibilities the other languages could offer. If these programmers would know

the unique features of the available tools, they could make considerably better decisions.

Better attainment of new tools

Newer and newer programming languages will appear, thus quality programming requires continuous learning. The more the basic elements of the programming languages are known, the easier it will be to learn and keep up with progress.

In our book most examples are in Ada, C/C++ or Java language for certain language constructs, there are only a few chapters (except of course those about logical and functional programming) where these languages are not referenced in almost every paragraph.

Our book is aimed at facilitating primarily, the studies of university and college students to learn about programming languages, and to help the work of IT and computer specialists. Some degree of knowledge of informatics is a prerequisite to fully understand our book: readers must have already solved some programming tasks on some programming languages.

Acknowledgements

The authors wish to thank for the support of TÁMOP tender on developing teaching materials.

We also thank Zoltán Horváth, the dean of the Faculty of Informatics at the Eötvös Loránd University for permitting the usage of the infrastructure of the Faculty of Informatics. Without his kind contribution this work could not have been completed.

We would like to thank the generous assistance of the PhD students who helped us with their feedback to improve this new edition of the book.

In such a voluminous book – despite all the best efforts of the authors and the editor – there could be errors. We would like to ask You, dear reader, if such an error is found, please notify us via email addressed to *proglang@inf.elte.hu*. We also welcome every kind of constructive criticism.

The current version of the whole book can be found as a downloadable pdf at:
`http://nyelvek.inf.elte.hu/APL`

1 Logic programming and Prolog

Povided that our knowledge about a problem is modelled by axioms, we can pose queries in the form of statements to be proved, in order to find objects satisfying these statements to be proved. The process of formal, constructive proof or deduction is some kind of computation. If this computation is controlled by an algorithm, then our statements together with the algorithm form a program, and writing such programs is logic programming (LP). In other words, a logic program is a set of axioms and a statemement to be proved + a machine controlling deduction. It is similar to an automated theorem prover. The main difference is the simplicity of the control component of logic programs: the process of computing is trackable, controlable, predictable. Its termination can be ensured. Its time and space complexity can be calculated: an LP language is a general purpose programming language, and effectivity is a main point.

In this chapter we survey the development of logic programming, its key concepts, and its (up till now) most important realization: the Prolog language. We discuss the programming methodology of Prolog, give examples, consider some extensions, and new trends. We emphasize the practical methods of Prolog programming; how to write valid and effective programs.

1.1 Introduction

The roots of logic programming (LP) reach as far as Hilbert, who proclaimed his program to axiomatize mathematics because the naive set theory had been found burdened with contradictions. Also he proposed to work out the methods of *automated theorem proving*.

The birth of the resolution algorithm [Rob65] is an important milestone of research, because we can extract answers from the run of the algorithm. Therefore it is a tool for constructive proof, and the deduction is a program searching or computing objects with given properties.[1]

Provided an automated theorem prover, the algorithm of proof or computation consists of two components:

1. The logical (declarative) description of the problem;
2. And the control component of the deduction or computation.

Shortly:

Algorithm = Logic + Control.

After some unsuccessful american efforts, in the beginning of the seventies Robert Kowalski realised that a general purpose programming language can be based on these principles [Kow79]. Alain Colmerauer and his colleagues designed and implemented the first LP language with acceptable performance in Marseille, 1972. This was the first version of Prolog. It was implemented as an interpreter.

The practical purpose of the developers was a natural language interface for a data base handler, and this became the first practical application of Prolog.[2]

[1] Familiarity with first order logic, and with resolution algorithm ([Nil82] and [Kow79]) is supposed, although the main flow of this chapter is understandable even without them.

[2] The name Prolog is an abbreviation of the French expession **Pro***grammation en* **log***ique*.

The first effective Prolog compiler was developed by David H. D. Warren and his colleagues at Edinborough in the second half of the seventies. Considering effectivity, it was comparable with best Lisp implementations of the time. It became a *de facto* standard of Prolog, determined the direction of its later development, and indirectly even its ISO standard, which was developed in the middle of nineties. Unfortunately, still there are compatibility problems between the module systems of the major Prolog compilers. And there are the extensions of the language allowed by the standard. Consequently, program code written on one Prolog compiler will not necessarily work on others, except if only the core of the language is used. Whenever we go beyond this, we sign it and we follow SICStus Prolog 4 [C+12], because nowdays this is one of the most popular implementations, and it follows closely the standard. (Our example programs have been tested in SICStus Prolog 4.2.3.)

Up till now, Prolog is the most widely used LP language. Its designers proved a practical sensibility. The compromises they made were theoretically criticized. Nevertheless, Prolog helps us in a good programming style, and in the fast development of programs with just a few errors, producing effective codes. This is the secret of its relative success.

Maybe there is no other language in which good programming style is so basic from the point of view of producing robust and effective code. Therefore, in this short chapter we concentrate on the *basic* features of the Prolog language, and on its *programming methodology*. We prefer this to a general introduction to LP, because we believe that this approach gives more to the newcomer. And we hope, at the end of the chapter we give good pointers to the reader yearning for wider knowledge.

1.2 Logic programs

A logic program is a set of logic *axioms* referring to a model, and a *query* referring to this model. The axioms describe the properties and relations of the objects of this model. For example, given statemens defining who is who's father. These axioms describe the relation `father` with arity two. If we tell who are male, the appropriate axioms describe the relation `male` with arity one (1.2.1).

A subset of axioms describing a relation is called a *predicate*. The run of a program is a constructive proof of a theorem being the consequence of the axioms. In other words, the run of the program computes an answer to a *query* or *goal*. During this computation or proof the predicates work as procedures.

In today's LP languages, any axiom is a *fact* or a *rule*. The axioms and queries (i.e. goals) are called *sentences* (although the queries may not be part of the source program). In some LP languages, the sentences can also be *declarations* referring to the predicates, and *directives* to be performed while loading the program. The declarations modify the run of the predicates, i.e. procedures, while the directives may modify the workspace of the program. The declarations

describe special properties of the predicates they refer to, while the directives are goals containing predicate invocations.

In Prolog, the declarations and the directives have the same form:
$:-goal.$
Each sentence is terminated by a dot, and at least one whitespace character.

The axioms of a logic program are also called *definite clauses*. The *statements* (the axioms and queries together) of a logic program are also called *Horn clauses*, but we will simply use the word *clauses*, because we will speak just about *Horn clauses*, and this abbreviation is quite common in logic programming.

1.2.1 Facts

The simplest logic programs consist of facts only.

The facts are formally atomic formulas. In the next example `father(x,y)` means that x is the father of y, while `male(x)` means that x is a male.

```
father('Abraham','Isaac'). father('Abraham','Ishmael').
father('Abraham','Anon').
father('Isaac','Jacob').   father('Isaac','Esau').

mother('Sarah','Isaac').   mother('Hagar','Ishmael').
mother('Rebeka','Jacob').  mother('Rebeka','Esau').

male('Abraham'). male('Isaac'). male('Ishmael').
male('Jacob').  male('Esau').

female('Sarah').     female('Hagar').  female('Rebeka').
female('Anon').
```

These facts express simple properties and relations of objects. The objects or entities are represented by their names: an identifier starting with a lower-case letter, or any sequence of characters delimited by single quotes or by single back-quotes. The name of a relation follows the same syntax.

Examples for the simplest queries:[3]

```
| ?- father('Abraham','Isaac').
yes
| ?- mother('Sarah','Jacob').
no
```

[3] The `| ?-` is the Prolog prompt: the Prolog environment waits for our query. We can type our commands and queries to this prompt. A command or query is always finished by a dot and `<Enter>`. The response of the Prolog system is `yes`, if it finds that our query as a statement is implied by the program, but its response is `no` if it finds that it is not implied by it. The proof is just one step here: either we find the fact identical to the query or not.

If you want to ask: *Is there some X, whose father is Isaac?* – then you receive two solutions according to the program above. X indicates the unknown entity in the query, i.e. an existentially quantified logical variable:

```
| ?- father('Isaac',X).
X = 'Jacob' ? ; X = 'Esau' ? ;
no
| ?-
```

Note: The names of logic variables are written as identifiers starting with an upper-case letter or underscore character.

The results above come from matching goal `father('Isaac',X)` against the appropriate facts.[4] During the process of matching, the variables of the goal are substituted by the appropriate nonvariables of the fact. Such substitutions represent the solutions of the goals.

Up till now we posed atomic goals, that is atomic queries to the Prolog environment. We can form *compound goals* as conjunctions of atomic ones. An atomic query in a compound goal is called its *subgoal*. For example, let us find Abraham's daughters:

```
| ?- father('Abraham',X), female(X).
X = 'Anon' ? ;
no
```

The solutions of a goal are the common solutions of its subgoals. We can say, a subgoal solved later provides a selection on the solutions of a subgoal solved earlier (*iff* they have common logic variable(s)).[5]

In logic programming, the variables are always logic variables. They stand for unknown objects. They may be universally or existentially quantified. The variables of goals are always existentially quantified. The run of the program produces constructive answers to the query, that is, it computes possible values of its variables.

Warning: Because a logic variable stands for an unknown object, destructive assignment statements like `X := X + 1` do not have stand in pure logic programming. (For example, in `X := X + 1`, which value of X stands for the unknown object?)

1.2.2 Rules

A proper conjunction of subgoals defines a new relation. For example, the conjunction "`father('Abraham',X), female(X)`" refers to the daughters of Abraham.

[4] The question marks refer to the questions of the SICStus Prolog environment, whether we ask for another solution. Our response `;` means that we ask for that. (If we do not need further solution, we press `<Enter>`. Then the system finishes the current conversation with `yes`.) In the example above, the finishing `no` abbreviates *no more solutions*.

[5] The word *iff* abbreviates the expression *if and only if*.

Assigning a name to this new relation, we receive a *rule*. Beside facts, rules form the second class of axioms in LP (logic programming).

```
'Abraham's daughter'(X) :- father('Abraham',X), female(X).
```

This reads: X is Abraham's daughter **if** Abraham is father of `X` **and** `X` is female. (This rule does not contain the information that no one has more fathers.) The logical variables of a rule are *universally* quantified, as it is with the rule above. The general form of a rule:

$A :- B_1, B_2, \ldots, B_n.$ $\quad (n > 0)$
(A, B_1, ..., B_n are atomic formulas.)

The consequence part of a rule (above A) is the *head* of the rule. The condition part of a rule (above $B_1,\ldots,B_n$) is the *body* of the rule.

Note: The facts have only head. A *fact* can be considered a *rule with empty body* or a rule with the condition part `true`. A *proper rule* is a rule which is not a fact.

Even a fact may contain (universally quantified) logic variables. Then it is called a *universal fact*. For example, the next fact says that everybody likes Sarah.

```
likes(_anybody,'Sarah').
```

And the next one says that anything is equal to itself. (Note that this predicate, i.e. $'='/2$ is a standard built-in of Prolog. Notice also that its name can be written between its parameters. This is posible, if infix operator notation (1.9) is defined to the name.)

```
X=X.
```

In such a way, we do not have to repeat the appropriate axiom for each element of the universe of the program. Anyway, the universe of a practical logic program is usually infinite, so we are not able to do this. There is another important difference between a set of facts and the appropriate universal fact. If you ask, who likes Sarah:

```
| ?- likes(Who,'Sarah').
true ? ;
no
```

The answer `true` means that the solution found did not substitute the logical variable `Who`. We may interpret this as a generic answer:

The statement `likes(Who,'Sarah')` is true for each element of the universe, that is, the unsubstituted varible can be substituted by any element "x" of the universe, and the statement `likes(x,'Sarah')` still remains true. It is quite natural that the Prolog machine did not find any other answer, just this generic one.

A *relation* and the *predicate* defining it can be specified by its name and arity, in the form *name/arity*. (The *arity* is the number arguments (i.e. parameters) of a predicate. An *argument* is a position in the program text, where a parameter is written.)

A relation is often defined by more axioms or rules. (Remember that facts can be considered as special rules with empty or `true` body.) In such a case, this relation is the union of the relations defined by the individual rules. The scope of a logical variable is the sentence containing it. It is visible in the whole sentence, because there are neither more local, nor more global variables. (It is visible exacly in its scope, because it cannot be hidden by other variables: there is no hierarchy of scopes, unlike in first-order logic, or in block-structured languages.) For example, in the next program the relation `parent/2` is the union of the relations `mother/2` and `father/2`:

```
parent(X,Y) :- mother(X,Y).
parent(X,Y) :- father(X,Y).

son(X,Y) :- parent(Y,X), male(X).

daughter(X,Y) :- parent(Y,X), female(X).

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

On the other hand, the relation `son/2` is the intersection of `parent/2` and `male/1`, and we receive similarly relation `daughter/2`, too. The last rule differs from the others, because it contains a new logical variable in the rule body. In this way, it has two different, but equivalent readings:

- For each `X`, `Y`, `Z`, `grandparent`(`X`, `Y`), *if* `parent`(`X`, `Z`) *and* `parent`(`Z`, `Y`).
- For each `X`, `Y`, `grandparent`(`X`, `Y`), *if* there exists a `Z` so that `parent`(`X`, `Z`) *and* `parent`(`Z`, `Y`) hold.

1.2.3 Computing the answer

The above readings of rules correspond to the *declarative* meaning of them. This is contrasted by the *procedural* meaning, which corresponds to the run of logic programs, i.e. to the process of the constructive proof of goals. In each case we pose a query. This is the goal to be proved. Now the Prolog system has to prove the subgoals, i.e. it has to eliminate them. The proof is finished when each subgoal has been eliminated (proved), and so just an empty *conjunction of goals* has been remained. The goal may be proved in *top-down*, or in a *bottom-up* manner. There are still other strategies that can be studied in [Kow79]. In each cases, the elementary step is the unification of two atomic formulas. Unification means the calculation and application of the most general unifier substitution

(*mgu*) of the atomic formulas. This is a substitution of the variables of the formulas, making them identical. Not each pair of atomic formulas are unifiable. In order to unify them, it is necessary, but insufficient, that they must refer to the same relation (same name and arity), and they must not contain different constants (or functors) at the same position. For example, the *mgu* of the atomic formulas $p(a, b)$ and $p(X, Y)$ is $\{X = a, Y = b\}$. The *mgu* of $p(a, Y)$ and $p(X, b)$ is also $\{X = a, Y = b\}$. The *mgu* of $p(a, Y)$ and $p(X, Z)$ is $\{X = a, Y = Z\}$, although $\{X = a, Y = b, Z = b\}$ and $\{X = a, Y = a, Z = a\}$ unify them, too. But the first one is more general than the others: $\{X = a, Y = b, Z = b\} = \{X = a, Y = Z\}\{Z = b\}$ and $\{X = a, Y = a, Z = a\} = \{X = a, Y = Z\}\{Z = a\}$. Next, $p(a, b)$ and $p(b, Y)$ do not have *mgu*: a substitution like $\{a = b, Y = b\}$ is NOT possible, because only variables can be substituted. (We do not know anything about the identity of two constants.) Similarly $p(a, b)$ and $p(Y, Y)$ do not have *mgu*, because $\{Y = a, Y = b\}$ is not a substitution: a variable like Y denotes something unknown entity of the universe, but cannot refer to two or more of them. (More details about unification can be found in any introductory material about first-order logic.) Now let us consider the *top-down*, and the *bottom-up* proofs of goals.

1. The *bottom-up* proof means that the conditions of the rules are unified with facts, and so they are eliminated one by one. In this way we receive new facts from the rules, until the facts unify with the subgoals of the original goal. When each subgoals have been eliminated the original goal have been proved. However, using this *bottom-up* method it is hard to direct the proof to the goal. Therefore in practice the *top-down* method is more often used. Here we start from the goal and go to the facts. This approach has been adopted up till now in most of the logic programming systems, for example in Prolog.
2. Shortly, the *top-down* proof consists of *steps of reduction*. One step of reduction means that one of the atomic formulas (i.e. subgoals) of the goal is unified with a *fact* or with a *head of a rule*. (Just like in resolution, the sentences taking part in the unification must not share common variables. This can be ensured by the appropriate renaming of the variables of the rule [i.e. fact or proper rule] taking part in the unification.) If we unify a fact, the appropriate subgoal is eliminated from the goal. If we unify the head of a proper rule, the appropriate subgoal is substituted by the rule body. In both cases, the unifying substitution is applied to the resulting conjunction of subgoals. This is a *step of reduction*. If a sequence of such steps of reduction, i.e. a top-down proof results in an empty conjunction of subgoals, then the original goal has been proved. This is a constructive proof. During the process, substitutions of the logical variables of the clauses have been performed. The results of these substitutions referring to the variables of the original goal provide for us the objects (terms) satisfying the conditions defined by the original goal.

For example, let us suppose that given a goal which is a $Q_1, Q_2, \ldots, Q_n$ conjunction of subgoals. And given a fact A sharing no common variable with this goal. Now, if $Q_1\theta = A\theta$, where θ is the *mgu* of the atomic formulas A and Q_1, then it is enough to prove $(Q_2, \ldots, Q_n)\theta$ in order to prove the original goal. This is the first case of a *step of reduction* referring to the goal to be proved.
The other case goes as follows: Consider our original goal $Q_1, Q_2, \ldots, Q_n$, and rule $A: -B_1, \ldots, B_m$ sharing no common variable with this goal. Let us suppose that θ is the *mgu* of A and Q_1. Then it is enough to prove $(B_1, \ldots, B_m, Q_2, \ldots, Q_n)\theta$ in order to prove the original goal. This is the second case of a *step of reduction* referring to the goal to be proved.
In both cases, if the result of the step of reduction is a C conjunction of subgoals, which we prove by the substitution φ, that is $C\varphi$ is proved to be true, then $(Q_1, Q_2, \ldots, Q_n)\theta\varphi$ is proved, too. This means that the substitution $\theta\varphi$ is a solution of the original query.

There is a theorem that both of the top-down and the bottom-up method of proof is correct and complete, i.e. exactly the goals following from the program can be proved if one of these strategies is used.

However, we will prefer the *top-down* method here, because it is easier to control, and it is preferred by logic programming languages, too. Corresponding to this method, there is a *procedural reading* of rules, i.e. facts and proper rules:

- The fact A means that subgoal A can be directly solved.
- The proper rule $A :- B_1, B_2, \ldots, B_n$ means that subgoal A can be solved by solving the subgoals $B_1, B_2, \ldots, B_n$.

Note that the subgoal, and the fact or rule head are rarely identical in practice. However, they must refer to the same relation (same name and arity). They must be unifiable as well. Let us suppose, that θ is the *mgu* of the atomic formulas A and Q. Then

- the fact A means that subgoal Q can be directly solved by the substitution θ, and
- the rule $A :- B_1, B_2, \ldots, B_n$ means that subgoal Q can be solved by solving the subgoals $B_1\theta, B_2\theta, \ldots, B_n\theta$. Provided that the substitution φ is their common solution, $\theta\varphi$ is a solution of Q.

For example, the rule "`grandparent(X,Y) :- parent(X,Z), parent(Z,Y).`" has the following procedural meaning: *In order to solve goal* `grandparent(X,Y)`*, solve goals* `parent(X,Z)`*, and* `parent(Z,Y)`*.*

1.2.4 Search trees

A subgoal may be unifiable by many facts and/or rule heads of our program. Therefore the possible goal reductions may have many branches forming a tree.

The root of this tree is the original goal, its nodes are conjunctions of subgoals derived through the top-down proof processes, its edges are the steps of different reductions, and its leaves are the empty conjunctions of subgoals, the *solution leaves*, and those nodes, where the subgoal selected for reduction cannot be reduced: these are the *fail leaves*. The solutions of the original goal correspond to the solution leaves. This tree is called *derivation tree*, *proof tree*, *search tree* or *search space*.

In the example in Figure 1.1 the levels of this tree are shown by the indentation. In each step we select the first subgoal for reduction. We try to eliminate it then, unifying it with a fact, or substitute it with the body of an appropriate rule (after unifying the subgoal with the head of the rule). This is the *leftmost subgoal selection strategy*. In the search tree we show only the substitutions referring to the variables of the actual goal. They are called *output substitutions*. They are shown in the form *variable*$<-$*substuting_term*.

For simplicity, if during the unification of a subgoal and a rule head or fact two variables must be unified, we always substitute the variable of the subgoal into the variable of the rule head or fact.

It is clear that in the steps of reduction where we select the first subgoal of the actual goal, we could choose another subgoal, and we would receive different search trees, if we applied different subgoal selection strategies.

The question is this: Whether we would receive different solutions or not? Fortunately, it is true that although the different subgoal selection strategies lead to different search trees, but each search tree containes the same solutions. Unfortunately, the different search trees usually have different sizes, as it can be checked easily by the reader, if he or she selects another subgoal selection strategy in the example in Figure 1.1. (Therefore the effectivity of the computation can be different, if our subgoal selection strategy is changed.)

The leftmost subgoal selection strategy has produced a search tree with minimal size here. However, if we posed the query `grandparent(X,'Jacob')`, the minimal search tree would be produced by a strategy selecting always the *rightmost* subgoal.

Notice that if we pose a query of the kind `grandparent(X,Y)`, after the first step of reduction we receive a goal like `parent(X,Z)`, `parent(Z,Y)`. Next it is better to choose the subgoal containing less variables. This method is often useful, even in other programs, because a subgoal containing less variables often leads to a search tree with fewer branches. But this is just heuristics.

In general, (when we allow recursive rules) it may happen that one search tree of the same query is finite, while the other is infinite. (We will see examples later.) In this case the search may go to an infinite branch, which leads to infinite computation. This means that we should choose a subgoal selection strategy producing finite search trees, if it were possible. Fortunately, if our rules are nonrecursive, the search tree is clearly finite.

```
?- grandparent('Abraham',X).
    parent('Abraham',Z),parent(Z,X).
      mother('Abraham',Z),parent(Z,X). % fails
      father('Abraham',Z),parent(Z,X).
        { Z <- 'Isaac' }
          parent('Isaac',X).
            mother('Isaac',X).      % fails
            father('Isaac',X).
              { X <- 'Jacob' }      % 1. solution
              { X <- 'Esau' }       % 2. solution
        { Z <- 'Ishmael' }
          parent('Ishmael',X).
            mother('Ishmael',X).    % fails
            father('Ishmael',X).    % fails
        { Z <- 'Anon' }
          parent('Anon',X).
            mother('Anon',X).       % fails
            father('Anon',X).       % fails
```

Figure 1.1: search tree of query `grandparent('Abraham',X)`

1.2.5 Recursive rules

Let us suppose that we want to describe the following relation.

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

It is clear that this new relation is a generalisation of the union of the relations `parent`/2, `grandparent`/2, 'great − gandparent'/2, and so on:

```
parent(X,Y) :- mother(X,Y).
parent(X,Y) :- father(X,Y).

grandparent(X,Y) :- parent(X,Z), parent(Z,Y).

'great-grandparent'(X,Y) :- parent(X,Z), grandparent(Z,Y).
```

After all, X is ancestor of Y iff X is parent of Y or X is parent of some ancestor of Y. This means that there are two cases and the second one is recursive. Therefore this may be expressed by a non-recursive and a recursive rule:

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

This is declaratively simple, but the question is, if there are infinite search trees

or not. And how to select the appropriate subgoal selection strategy to ensure the finiteness of the search tree.

For example, let us suppose temporarily, that the next fact (and only the next fact) defines the relation `parent/2`.

```
parent('First','First').
```

Now let us ask about the descendants of First, and look at the rightmost branch of the search tree (applying the leftmost subgoal selection strategy):

```
?- ancestor('First',X).
         parent('First',Z1), ancestor(Z1,X).
           { Z1 <- 'First' }
             ancestor('First',X).
                parent('First',Z2), ancestor(Z2,X).
                  { Z2 <- 'First' }
                    ancestor('First',X).
                        ...
```

Clearly, it is infinite. And it is infinite with any subgoal selection strategy.

Now, let us reconsider the relation `parent/2`. We either consider the original relation or the temporary one, it defines a directed, finite graph. Its edges are given by the relation `parent/2`, and its nodes are the endpoints of the edges. The temporary graph consists of a trivial loop, but the original one is acyclic. Then the relation `ancestor/2` corresponds to nonempty, finite paths in the graph. These paths in the temporary graph are looping, but in the original one they contain no cycle, and their length has an upper limit. (These later properties of the original graph are based on the earthly nature of relation `parent/2`.) Now, let us omit the temporary definition of `parent/2`.

And let us consider the query `ancestor('Abraham',X)`. It is easy to see, that it is useful to apply the leftmost subgoal selection strategy: In this case in the subgoals of the form `parent(X,Y)` the first parameter will be always known. This property makes the search tree more slim. And it is more important, that the search tree will be finite, because in each recursive call we process one additional edge of a directed, finite path starting from node `'Abraham'`.

Now let us consider the query `ancestor(X,'Isaac')`. In order to generate the search tree, it may seem useful to choose the rightmost subgoal selection strategy, because the second parameter of the rightmost subgoal will be always a constant. However, the rightmost subgoal will be always a recursive call, and we generate an infinite search tree. If we prefer the second rule in goal reduction, we find its infinite branch immediately. If we prefer the first rule, first we find the solutions, and then go to the infinite branch. In general, a computation does not know, when it has found the last solution. Therefore this can be troublesome.

However, if we reconsider the query `ancestor(X,'Isaac')`, and we prefer the *leftmost* subgoal selection strategy, then the first `parent(X,Y)` call selects an edge from the graph, and then the searching of the path to the node `'Isaac'` goes on

from its right endpoint. Therefore, the search tree remains finite, although it will be probably fatter, than with the previous query. And it can be seen similarly, that here, the search tree of any query of the form `ancestor(X,Y)` is finite.

One might suggest another subgoal selection strategy now: given a conjunction of subgoals, we should select a subgoal defined by a nonrecursive predicate. This strategy is often useful, but there are some cases, when it does not help:

```
ancestor00(X,Y) :- parent(X,Y).
ancestor00(X,Y) :- ancestor00(X,Z), ancestor00(Z,Y).
```

This definition of relation `ancestor00/2` is logically equivalent with predicate `ancestor/2`. However, a query of the form `ancestor00(X,Y)` always has an infinite search tree, regardless of the actual parameter values, regardless of the subgoal selection strategy.

In addition, the more complex a subgoal selection strategy is, the harder it is to follow its behaviour, to prove the finiteness of the corresponding search tree, and to calculate the effectivity of our logic program.

After all, we can conclude that good formulation is more essential than sophisticated subgoal selection strategy. The formulation of a logic program must match the expectable queries, and ensuring this is easier if the subgoal selection strategy is simple.

1.3 Introduction to the Prolog programming language

Therefore the invertors of the Prolog programming language decided in favour of the leftmost subgoal selection strategy.

The program is performed by the Prolog machine, which is a virtual machine. It may work as an interpreter or emulator, but it may be built into an independent, executable file, too.

The search tree is traversed by backtracking, but its branches are not built up in advance, and the actual branch is always destroyed when Prolog backtracks from it. In such a way, always just one branch of the search tree is stored in the call stack containing the call frames of the predicate invocations and the choice points for the alternative branches. In this way, Prolog tries to minimize the memory needed by the run of the program. The facts and rules are tried in their order.

Prolog serves for **Pro***gramming in* **log***ic*. In reality, Prolog is a very high level, general purpose programming language, and it is not a tool for automated theorem proving. Its logic formulas and its inference machine are too simple for the later purpose. Yes, the Prolog machine is simple enough to be controlled by the programmer, so that he or she can prove the finiteness of the search tree, and calculate the time and space complexity of the program.

The correctness of a Prolog program is always related to goals to be solved. Partial correctness simply means correct formalization. In order to prove the

termination of the program it is enough to prove, that the search trees of the possible goals are finite.

If there is no (directly or indirectly) recursive rule in the program, then the finiteness of the search trees is a trivial statement. If there is a (directly or indirectly) recursive predicate, then we consider the possible goals invoking it, and prove the finiteness of the related search trees: usually we define a terminator function whose value is a nonnegative integer for each node of the tree, and its values are strictly decreasing while going down in the tree. These properties make sure that value of the function at the predicate invocation is an upper bound of the depth of the related search tree.

For example, in the case of the predicate `ancestor/2` the length of the unprocessed path is an appropriate terminator function (1.2.5). Surely, this is decreasing by processing a subgoal of the form `parent(X,Z)`, and it cannot be negative. Therefore, the search tree is finite, and the predicate invocation terminates.

However, in the case of the predicate `ancestor00/2` the search tree is infinite (1.2.5), and there is no terminator function.

A Prolog system nomally offers the user an interactive programming environment, using an internal or external text editor.

It is typical that the Prolog system and a standard text editor (for example Emacs) communicates through an interface supported by both of them, and the Prolog environment starts inside the text editor, benefiting from the services of it, like automatic indentation and text colouring of the source code, highlighting the match of different kinds of brackets, compiling and loading programs, source level debugging, and so on.

The Prolog environment normally starts in a special console window (although it may have a GUI). We type our commands and queries at the Prolog prompt of this console, and we can read the standard output of the Prolog system on it. If we want to make an executable file [C$^+$12], the program must contain at least one directive (a goal to be performed while loading the program). This or these provide a primary control of its execution. A source program may consist of many files. We can variously load the program files into the Prolog environment. If we want to run the program in an interpreted way we can use the `consult(File)` command: we can type it at the Prolog prompt (?−), where `File` must be a Prolog source file. (Traditionally, `.pl` is the de facto standard extension of Prolog sources, and this extension can be omitted.) Any predicate of the program loaded can be queried.

For example, let us suppose that the Prolog predicates defined in this book are in the source file `lpp.pl`.

```
| ?- consult(lpp).
% consulting c:/documents and settings/pl/book/lpp.pl...
% consulted c:/documents and settings/pl/book/lpp.pl
%          in module user, 0 msec 8 bytes
yes
| ?- grandparent('Sarah',GrandChild).
GrandChild = 'Jacob' ? ;
GrandChild = 'Esau' ? ;
no
| ?- ancestor(A,'Jacob').
A = 'Rebeka' ? ;
A = 'Isaac' ? ;
A = 'Sarah' ? ;
A = 'Abraham' ? ;
no
```

Therefore, any predicate of the program loaded can be directly tested. We need no testbed.[6]

1.4 The data structures of a logic program

The data structures of a logic program are called *terms* (like in mathematical logic). According to ISO Prolog, a Prolog term can be a logical variable called `var` (referring to an unknown term) or a partially or properly known term called `nonvar`.

A `nonvar` can be a constant called `atomic`, or a structured term called `compound`.

An `atomic` term can be a name constant called `atom`, or it can be a `number`.

A `number` can be an `integer` or a `float`. (The syntax of numbers will be familiar to C, C++ or Java programmers.)

A name constant or `atom` is syntactically an identifier starting with a lowercase letter, a sequence of characters between quotes ~~or backquotes,~~ special character sequences of the characters $+ - * / \backslash \wedge < > = : . ? @ \# \& \$$ (for example: $=<$, $@>=$, $?-$, $*\$$, etc.), or one of the following extras: ; (called *or*, *else*, or *elsif*), ! (called *cut*), [] (called *nil*), and {} (called *empty*).

The name of a variable is syntactically an identifier starting with an uppercase letter or underscore sign. A variable denotes an unknown entity, like the variables of mathematical equations and formulas. They are very different from the variables of procedural (OOP) languages like Pascal, C, C++, and Java. The aim of the computations is the determination of the possible values of the variables of the queries, like in the case of the mathematical equations. Therefore

[6] In LP languages, like Prolog, any data structure can be described at the source code level, and even the private predicates of a module can be accessed.

it is not reasonable, and it is not possible to write assignment statements. When a logical variable has been subtituted by a `nonvar` term, then it cannot be distinguished from the substituting term, except if the program backtracks before the substitution, when the variable looses its value. This may be strange for a programmer used to procedural programming languages, but after some practice in logic programming it becomes natural. As we work with variable substitutions instead of assignment statements, we get rid of the most dangerous source of programming mistakes.

Summary: A `var` is a logical variable which has not been substituted by a `nonvar` (on the actual branch of the search tree).

It is a tradition that a variable the value of which is important is denoted by an identitifer starting with an upper-case letter, but a variable the value of which is not important is denoted by an identitifer starting with an underscore sign:

```
father(SomeBody) :- father(SomeBody,_Child).
```

If there is a sentence in a source file of a logic program, and a variable of this sentence has just one occurence in this sentence, then the value of this variable is not important, because there is no place where to pass its value.

Note: Therefore, many Prolog environments suppose that an identifier starting with an upper-case letter denotes an important variable. If there is a sentence in the source code with just one occurence of such a variable, then it sends a warning, in order to help us to get rid of typing mistakes.

On the other hand, if we type a query at the Prolog prompt, and it contains a variable starting with an underscore sign, then its value is not automatically printed when the query has been solved:

```
| ?- grandparent(GrandParent,GrandChild), male(GrandParent).
GrandChild = 'Jacob', GrandParent = 'Abraham' ? ;
GrandChild = 'Esau', GrandParent = 'Abraham' ? ;
no
| ?- grandparent(GrandParent,_GrandChild), male(GrandParent).
GrandParent = 'Abraham' ? ;
GrandParent = 'Abraham' ? ;
no
```

There is the *anonymous variable* consisting of just an underscore sign, the occurences of which are different logical variables, even inside a single sentence. Therefore it can denote only unimportant variables. *The scope of any other variable is the sentence containing it.* For example:

```
father_and_son(SomeBody) :-
   father(SomeBody,_), parent(_,SomeBody).
```

In order to store structured information we can use *compound terms* of the form $f(t_1, ..., t_n)$, where f is an `atom`, $t_1, ..., t_n$ are arbitrary terms, and the *function symbol* or *functor* is f/n, which unites the terms $t_1, ..., t_n$ into a single compound term (although this strict form is sometimes relaxed by so called *syntactic sugars*). Therefore the functors are specified by their *name/arity* pairs, like relations and predicates. Any parameter t_i may be `atomic`, `var`, or `compound`, too.

A term which is not `compound`, is called a *simple term*. This means that a *simple term* is `atomic` or `var`.

A *callable term* is a `compound` or an `atom`, because these terms may represent the goals (queries) of a program.

A *ground term* is a term containing no `var`. Recursively, a ground term is an `atomic`, or a `compound` with ground term parameters.

Therefore the compound terms may represent recursive data structures like lists and trees. For example, an empty tree may be represented by the *atom* [], and a nonempty tree by the *compound term* .(*root*, $\mathtt{t_1}$, ... , $\mathtt{t_n}$),
where each t_i denotes a direct subtree.

Now a list is a unary tree. For example, a list of the numbers 1, 2, and 3 is .(1, .(2, .(3, []))). (Syntactic sugars will make the notation more convenient, especially here.)

And the tree .(4, .(2, .(1, [], []), .(3, [], [])), .(5, [], [])) is a binary search tree with depth two.

Surely, we may choose any other notation. The important thing is to apply it consistently. For example, the empty tree may be denoted by the atom `o`. The functor name of the nonempty trees may be the atom `t`, and, especially in binary trees, it may be convenient to put the root in the middle, in the form `t`(*leftSubTree*, *root*, *rightSubTree*).
Using this notation, the previous binary seach tree looks like
t(t(t(o, 1, o), 2, t(o, 3, o)), 4, t(o, 5, o)).

1.5 List handling with recursive logic programs

We have seen, that we can represent lists as unary trees. Now, before discussing the basic list handling predicates, let us see the different kinds of lists and list-like structures.

A *list* may be *proper* or *partial*. A *non-list* is a term which is not a list. A *proper list* may be *empty* or *nonempty*. The standard representation of the *empty list* is the atom [] read as *nil*. The standard constructor of a nonempty list is $'.'/2$. A term .(`X`, `Xs`) is proper list *iff* `Xs` is proper list.[7]

[7] In this chapter, the two-letter identifiers `As, Bs, Cs, ..., Xs, Ys, Zs` usually refer to lists.

According to this, the list [1, 2, 3] can be written as .(1, .(2, .(3, []))). Fortunately, we have three standard syntactic sugars to allow an easier-to-read notation. (The notation convention implied may seem too complicated at the first glance, but it is easy to learn and useful.)

In order to introduce these syntactic sugars, still we need some definitions:

`X == Y` means that the terms `X` and `Y` are *identical*. (Actually, ′ ==′ /2 is a built-in predicate in Prolog with the same meaning.)

A *list-term* is a compound term with the main functor ′.′/2. (Therefore, the notion of list-term is a generalisation of the notion of nonempty list: The list .(1, .(2, .(3, []))) is a list-term, and the non-list .(1, .(2, .(3, non_nil))) is a list-term, too.)

`X` is the (first) head and `Ys` is the (first) tail of the term `T`, *iff* `T == .(X, Ys)`.

`X` is the $(i+1)^{\text{th}}$ head and `Ys` is the $(i+1)^{\text{th}}$ tail of term `T`, *iff* `T == .(A, Bs)`, and `X` is the i^{th} head and `Ys` is the i^{th} tail of term `Bs`, where i is a positive integer number. The i^{th} head of a term is also called its i^{th} element.

Now let us see the standard syntactic sugars allowing the easier-to-read notation.

1. A list-term .(`X`, `Xs`) can be written as [`X`|`Xs`].
2. The front elements of a list-term `T` can be separated by commas: if `X1`, `X2`, ..., `Xn` are the first `n` heads of `T`, and `Ys` is its $\mathtt{n}^{\text{th}}$ tail, then `T` == [`X1`, `X2`, ..., `Xn`|`Ys`].
3. "|[]" may be omitted in [`X1`, `X2`, ..., `Xn`|[]].

In this way

```
| ?- .(X,Xs)==[X|Xs], [X1|[X2|Xs]]==[X1,X2|Xs],
    [X1,X2,X3|[]]==[X1,X2,X3],
     .(1,.(2,.(3,[])))==[1,2,3].
true
```

An element of a list is an arbitrary term. It may be even a variable or a list. For example, [a|[_]] is a proper list, and [[_]|[_]] is also a proper list, but [[_]|a] is not a list, because the atom `a` is not a list.

A term is *partial list,* iff it is a `var` or it is a term of the form [`X`|`Xs`], where `Xs` is a partial list. A `var` is an empty partial list.

Therefore a nonempty partial list always has the form `T` == [`X1`, `X2`, ..., `Xn`|`Vs`], where `Vs` is a `var`. And a term is partial list, *iff* it is not a proper list but after an appropriate substitution it becomes a proper list. Actually, if the `var` at the end of a partial list is substituted by a proper list, then the partial list becomes a proper list. If the `var` at the end of a partial list is substituted by another partial list, then the first partial list still remains a partial list. If the `var` at the end of a partial list is substituted by a non-list, then the partial list becomes a non-list, because a list-term is a non-list, *iff* at its end

there is a `nonvar` different from the empty proper list, that is []. No substitution maps a non-list to a list.

Conventionally, iterative data structures are represented by lists in Prolog, and there are a lot of built-in and library predicates supporting list handling in the different implementations.

At last, a *ground list* is a variable free list, i.e. a list which is a ground term. Therefore the ground lists form a proper subset of proper lists.

For example, let us suppose that Xs is a `var`. Now `Xs` and [1, 2|`Xs`] are *partial lists*, but [`Xs`] and [1, 2, `Xs`] are proper lists of one and three elements. Neither of these four lists is a ground list. However, if `Xs` == 3, then neither `Xs` nor [1, 2|`Xs`] is list, but [`Xs`] and [1, 2, `Xs`] are ground lists.

Consider the next predicate.

```
list([]).
list([_X|Xs]) :- list(Xs).
```

If the goal `list(Ys)` is parameterized by a proper list, it will be successful with no variable substitution. Parameterized by a partial list, this goal will have infinite search tree, and infinite number of solutions. These solutions will be the most general proper list samples of `Ys`. For example:

```
| ?- list(Ys).
Ys = [] ? ;  Ys = [_A] ? ;  Ys = [_A,_B] ? ;
Ys = [_A,_B,_C] ? ; Ys = [_A,_B,_C,_D] ? <Enter>
yes
```

The goal `list(Ys)` parameterized with a non-list will fail.

Now, using the notations above, let us consider some list handling predicates. And we discuss some useful programming methods used in logic programs which handle lists and recursive data structures in general.

We need a definition: The *precondition of a predicate* specifes the set of goals allowed to invoke that predicate. For example, we may prescribe that the first parameter must be a proper list, the second parameter must be positive integer, and so on.

1.5.1 Recursive search

First let us consider the classic predicate `member_`/2.[8] [9]

```
% PreCond: Xs is a proper list.
% member_(X,Xs) :- X is a member of list Xs.
member_(X,[X|_Xs]).
member_(X,[_X|Xs]) :- member_(X,Xs).
```

[8] The character `%` followed by any sequence of characters up to the end of the line is comment.

[9] `member/2` and `append/3` are the built-in counterparts of our `member_/2` and `append_/3` (1.5.2) in many Prolog implementations. Therefore we cannot redefine them.

So the members of a list are its head, and the members of its tail. Invoking this predicate, the appropriate item is found directly, or through recursive calls. Therefore this programming method is called *recursive search.*

Let us consider the precondition of `member_(X,Xs)`: If Xs were a partial list, the search tree of the query would be infinite. If it is a proper list, its length is strictly decreasing throughout the recursion. This guarantees the finiteness of the search tree. For example:

```
| ?- member_(X,[1,2,3]).
X = 1 ? ;   X = 2 ? ;   X = 3 ? ;   no
| ?- member_(2,[1,2,3,X,4]).
true ? ;   X = 2 ? ;   no
```

Notice that also `ancestor/2` applies recursive search.

1.5.2 Step-by-step approximation of the output

The next predicate is the standard predicate for appending or splitting lists. Notice that we have two rules: One of them refers to the case when the first parameter is an empty list, and the other handles nonempty lists in the first argument, recursively, while the length of a proper list parameter is strictly decreasing through the recursion, guaranteeing the finiteness of the search tree. Therefore this is quite a usual way of organizing list handling predicates in Prolog.

```
% PreCond: Xs or XsYs is a proper list,
%          the other two parameters are lists.
% append_(Xs,Ys,XsYs) :-
%   appending lists Xs and Ys we receive list XsYs.
append_([],Ys,Ys).
append_([X|Xs],Ys,[X|Zs]) :- append_(Xs,Ys,Zs).
```

Appending the empty list and any other lists we receive the other list. Appending a nonempty list `[X|Xs]` and another list `Ys`, the head of the connected list is `X`, while the tail of the connected list is the result of appending `Xs` and `Ys`. Notice that in the recursion the length of the first parameter, and the length of the third parameter is also decreasing. Therefore, the precondition above is strong enough: If one of the first and third parameters is a proper list, the search tree is finite. (However, if both of them are partial lists, the search tree is clearly infinite.) The second parameter may be proper or partial list. Notice that if the first parameter is a non-list, the call will surely fail: this case is not handled. And if the second or third parameter is a non-list, we may fail or receive non-list results.

Consider now the search tree of query `append_([1,2],[3,4],Zs)` in Figure 1.2. (We suppose that the LP system renames each variable `V` of a rule to `V`i, where i is the sequence number of the actual step of the goal reductions.)

```
append_([1,2],[3,4],Zs)
  { Zs <- [1|Zs1] }
    append_([2],[3,4],Zs1)
      { Zs1 <- [2|Zs2] }
        append_([],[3,4],Zs2)
          { Zs2 <- [3,4] }          % (solution)

% After all:
Zs=[1|Zs1]=[1|[2|Zs2]]=[1,2|Zs2]=[1,2|[3,4]]=[1,2,3,4]
```

Figure 1.2: Search tree of query `append_([1,2],[3,4],Zs)`

```
| ?- append_([1,2],Ys,Zs).
Zs = [1,2|Ys] ? ;
no
| ?- append_(Xs,Ys,[1,2,3]).
Xs = [], Ys = [1,2,3] ? ;
Xs = [1], Ys = [2,3] ? ;
Xs = [1,2], Ys = [3] ? ;
Xs = [1,2,3], Ys = [] ? ;
no
```

Figure 1.3: Queries of `append_/3`

Notice that the search tree is linear (unary) tree here. Everywhere just the head of one of the rules have unified the goal. On the other hand, the appended proper list has been approximated in more steps, through partial lists. In each steps, we substituted the `var` tail of the actual partial list, computing bigger and bigger part of the output list, until we received the appended proper list as a result.

Here we have built up the result data structure in a top-down manner. First, we built its topmost level, but we left some details undefined. Then these details were refined by variable substitutions in the same manner again and again. Therefore, this way of computing the result is called the *step-by-step approximation of the output.*

Exercise: Try to draw the search tree of both of the queries in Figure 1.3. Observe the process of the approximation of the output.

In the first query the result is a partial list: its front is equal to the first parameter, and its tail is the second parameter, whatever it may be.

In the second query we receive a search tree with branches, and the results (at the leaves of the tree) are the possible cuts of the list (in the third parameter) into two parts.

1.5.3 Accumulator pairs

Accumulators are used to build the result in a bottom-up manner. Their usage is shown through the following example.

```
% PreCond: Xs is a proper list, Ys and Zs are lists.
% rev_app(Xs,Ys,Zs) :-
%     Xs reversed and appended before Ys provides Zs.
rev_app([],Ys,Ys).
rev_app([X|Xs],Ys,Zs) :- rev_app(Xs,[X|Ys],Zs).
```

The empty list reversed and appended before another list provides this other list. The nonempty list [X|Xs] reversed and appended before another list provides Xs reversed and appended before [X|Ys].

The length of the first parameter strictly decreases through the recursion. Therefore, if Xs is a proper list, the search tree of the query rev_app(Xs, Ys, Zs) is finite. If Xs is a partial list, the search tree is clearly infinite. Both of Ys and Zs may be proper or partial list, especially var. Let us consider the search tree of the query rev_app([1, 2], [3, 4], Zs).

```
rev_app([1,2],[3,4],Zs)
  rev_app([2],[1,3,4],Zs)
    rev_app([],[2,1,3,4],Zs)
      { Zs <- [2,1,3,4] }            % solution
```

The resulting data structure is built up in the second argument, in bottom-up manner, through the recursion. Therefore the second argument is an *accumulator*. The result is completed at the bottom of the recursion. In order to return the result we need a third argument, a *tunel*: it is passed through the recursion without change, and it is substituted by the result at its bottom. An accumulator and a tunel argument form an *accumulator pair*, because they are found in the logic programs always in pair.

In practice, we often write a predicate containing more accumulator pairs, and/or parameters calculated by step-by-step approximation.

For example, notice that in the query ? − append_(Xs, Ys, [1, 2, 3]) the first parameter is calculated by step-by-step approximation. The second argument is a tunel, while the third one behaves like a so called negative accumulator: the data structure to be returned through the tunnel is not built up, but appropriately pulled down in it.

1.5.4 The method of generalisation

The next predicate is an example of *generalisation*. In such a case the problem is generalized, solved, and its program is used to handle the original, more specific problem. This method is often used in human problem solving, especially in programming, and even more specially in LP. It is often used when we write a recursive procedure: It is able to solve a more general problem, and it is appropriately parameterized when it is called:

```
% PreCond: Xs is a proper list.
% reverse(Xs,Ys) :- the reverse of Xs is Ys.
reverse(Xs,Ys) :- rev_app(Xs,[],Ys).
```

The reverse of list `Xs` is `Ys`, if `Xs` reversed and appended before the empty list provides Ys.

Therefore, this predicate inherits the condition of finiteness from `rev_app/3`: If the first parameter of goal `reverse`$(\texttt{Xs}, \texttt{Ys})$ is a proper list, then the search tree of this goal is finite. But if `Xs` is a partial list, then this search tree is infinite.

For example, if `Xs` is a `var`, then the invocation `reverse`$([1,2,3], \texttt{Xs})$ calculates its only solution $\texttt{Xs} = [3,2,1]$, and its search tree is finite. Naturally, the query `reverse`$(\texttt{Xs}, [1,2,3])$ has the same, only solution. But its search tree is infinite.

Especially, in the Prolog predicate `rev_app/3`, the fact finishing the recursion precedes the recursive rule. Therefore, we will find the solution of the query `reverse`$(\texttt{Xs}, [1,2,3])$. Nonetheless if we instruct the Prolog environment to find another solution, it always backtracks, and tries to unify longer and longer lists of `var`s with the second parameter of the goal, but fails.

Let the reader explain the following behaviour now.

```
| ?- reverse([X,Y,Z],[1,2,3]).
X = 3, Y = 2, Z = 1 ? ;
no
```

The four methods discussed above are used while handling recursive data structures in LP, especially lists and trees.

1.6 The Prolog machine

Full Prolog contains meta-logical and extra-logical constructs, too. Up till now we have omitted these extensions and concentrated on *pure Prolog*, which is the core of the language: it is based on pure mathematical logic. In this section we provide a detailed explanation of the abstract interpreter performing pure Prolog programs. However, we remain mainly at the abstract level, and explain just a few implementation details. We try to help Prolog programmers rather than Prolog implementors.

A pure Prolog program is a set of the Prolog programmer's predicates, and a query or goal to be answered or solved: the predicates manifest our knowledge in the topic. Each predicate is a set of rules at the declarative level, and a sequence of rules at the procedural level. Each rule is a fact or a proper rule. No rule body or query contains any subgoals referring to a built-in predicate.

1.6.1 Executing pure Prolog programs

Executing a program, the Prolog machine performs preorder traversal of the search tree (1.2.4) of the query. During the traversal just the actual branch of the tree is stored. This branch is the path from the root to the actual node. A node of this branch is labelled by the appropriate goal, by the predicate its first subgoal refers to, and by the untried rules of this predicate in their original order. The edges of the actual branch represent the goal reduction steps leading to the actual goal. An edge is labelled by the rule used in this step of reduction, and by the variables of the parent goal substituted in the connected unification. Notice that we do not have to remember the unifying substitutions. See the algorithm of the execution of Prolog programs coming here. We use the word *matching* instead of *unifying*. This will be explained in (1.6.2).

1. In the beginning, the root of the search tree is labelled only by the original query (goal). Now, the root is the actual node, and only this root is stored. (Each time the node furthest from the root on the actual branch is the actual node.)
2. If the actual node is labelled by the empty conjunction of subgoals, we have found a solution: this is the actual substitution of the variables of the original query. In this case we print the solution, and ask the user, whether he or she asks for another solution.
   If so, we continue from (9). (Backtracking.)
   If not, *we have finished the search successfully.*
3. If the actual node is labelled by a nonempty goal, let us consider its first subgoal. Let us label this node also with the predicate referred to by this subgoal, and with the list rules of this predicate. (Note that if this list contains two or more rules, it forms a *choice point*, which remains *living* while it is nonempty.) Now let us rename the variables of the rules, so that they share no variable with the goal. (It is the most effective to generate completely new variable names.)
4. If the actual list of clauses is empty, we have arrived at a dead end, that is, at a fail node, and we continue form (9). (Backtracking.)
5. Let q be the first item of the actual list of rules. Delete q from this list. (If the list becomes empty, the choice point possibly generated above lives no more.)
6. Try to match the first subgoal of the actual goal with the head of q (see 1.6.2).

7. If this matching fails, we continue from (4).
8. If this matching is successful, we perform a step of reduction: The variables substituted during the matching are appropriately substituted everywhere in the goal and in the rule. (This is usually ensured by a linked representation of the variables.) The actually stored branch of the search tree is extended by a new node. The edge leading to this node is labelled by the actually substituted variables of the actual node, and by q. In order to compute the first label of the new node, we substitute the first subgoal of the actual goal with the (possibly empty) body of q, where the matching substitution has already been performed on each participants. Next the new node becomes the actual node. Then we continue from (2).
9. *Backtracking:* If the actual node is the root of the tree, then the execution of the program finishes with *fail*.
   Otherwise, we delete the actual node of the actually stored branch of the search tree, and its parent will be the actual node. During this the substitution of the variables labelling the edge between this two nodes are also deleted. Then we continue from (4).

1.6.2 Pattern matching

We have seen that the basic operations of the Prolog machine are goal reduction and backtracking. And the key of a step of reduction is the unification of the actual subgoal and rule head. Theoretically, the Prolog machine should compute the *mgu* of them (see Section 1.2.3); it should try to unify atomic formulas with the same name and arity. Therefore, it should unify two sequences of formulas of the same length. Procedurally, a predicate is a procedure, and this unification is a kind of parameter passing. Now, we face the problem that computing the *mgu* is too expensive to be used as parameter passing. Provided that it is adopted, logic programs become unacceptably slow. Therefore, a simplified unification called *pattern matching* is adopted in Prolog:

1. If the too sequences of terms are empty, We are ready. Otherwise, we match the pair of the first elements of the two sequences according to (2). Then we continue with matching the rests of the two sequences according to (1).
2. Provided that we have two `atomic` terms (see 1.4), they match, *iff* they are identical. If they are identical, they match without variable substitution. If they are different, then this algorithm of pattern matching *fails*.
3. If one of the terms is `atomic`, but the other one is compound, then this algorithm of pattern matching fails.
4. If both of them are `compound` terms, and their functor names and arities are the same, then their sequences of parameters are matched according to (1). If both of them are `compound` terms, but their functornames or arities are different, then this algorithm of pattern matching fails.

5. Provided that we have two `var` terms, any of them may be substituted by the other one. (However, in our examples, we will always subtitute the `var` term of the rule head with the `var` term of the goal, because it is easier to follow, and it usually allows a more effective implementation.)
6. If just one of the terms is a `var`, then we substitute it with the other term, which is an `atomic` or a `compound`.

The variable substitutions of the algorithm above are performed on each occurence of these variables in the goal and rule taking part in the actual step of reduction.

We keep a record of each substitution of the variables of the goal. If the pattern matching eventually fails, the substitutions of the variables of the goal are deleted according to our records. If the pattern matching succeeds, then the appropriate edge of the actual branch of the search tree is labelled by this recorded set of variables (1.6.1).

In the first point of this algorithm, the pattern matching of the appropriate term pairs of the two term sequences may be performed in arbitrary order, or even in parallel.

Notice that in point (6) of the algorithm of pattern matching above we substitute a `var` with a `compound` unconditionally. Nonetheless in the original algorithm of unification, this is allowed *iff* the `compound` does not contain the `var`. The obligatory check of this condition before a `var` is substituted by a `compound` is called *occurs check* ([FGN90], [Kow79], [DEDC96], [ISO95] and [SS94]). The Prolog machine omits the occurs check for reasons of efficiency, because its operational complexity is $O(size(compound))$, and the size of the `compound` may be extremely large in practice.

For example, consider the built-in predicate $'='/2$ (see Section 1.2.2, i.e. fact "$\mathtt{X} = \mathtt{X}$."). Query $\mathtt{Y} = \mathtt{f(Y)}$ should fail, at least according to mathematical logic. However, according to a typical Prolog implementation the solution of the query is the substitution $\{\mathtt{X} < -\mathtt{f(Y)}, \mathtt{Y} < -\mathtt{f(Y)}\}$. According to mathematical logic, *this is not a unifying substitution.* According to the Prolog standard, the result is undefined, if during the pattern matching a `var` faces a `compound` containing it ([DEDC96] and [ISO95]). Our algorithm of pattern matching is more special than that of the Prolog standard ([DEDC96] and [ISO95]), wich may even abort or go to an infinite loop in this situation. First we subtitute `X` with `Y`, and then `Y` with `f(Y)`. Therefore, we avoid the explicite computation of the unifying substitution. In this way, we avoid infinite loops and program aborts, and we may say (following Colmerauer [Col82]), that the substitution $\{\mathtt{Y} < -\mathtt{f(Y)}\}$ generates a cyclic term $\mathtt{Y} = \mathtt{f(Y)}$. Some Prolog implementations even give us tools to handle cyclic terms [C+12]. However, we will not take benefit of such tools here, and even try to avoid generating cyclic terms, because it may be difficult to port a program working with cyclic terms between Prolog platforms.

After all, in programming practice it is extremely rare that we face the problem that Prolog omits occurs check. Most of the applications work well even if we are not aware of the possibility of the unhappy situations following this omission. One must admit that this omission is one of those briliant compromises which make Prolog a practical programming language: Without this compromise the cost of unifying a `var` and a `compound` would be proportional to the size of the `compound`, and this cost would arise in each predicate invocation where we work with structured terms. Many applications work with long lists and/or big trees. Then the cost of a simple procedure call would be proportional to the length of the list or size of the tree. However, in a typical Prolog environment a `var` and a structured term are matched with constant cost which is independent from the size of the structure.

After all, the Prolog standard handles the occurs check problem in an elegant and effective manner. And now we are going to discuss this topic.

1.6.3 NSTO programs

In NSTO programs the *occurs check* can be omitted safely. Now, we consider how to write programs where even the original algorithm of unification would perform no successful occurs check. Such predicate invocations, that is subgoals are called *NSTO* (Not Subject To Occurs check), otherwise we speak of *STO* subgoals. If each of the subgoals of a program is *NSTO*, then the program is also *NSTO*. Otherwise the program is *STO*.

Note that these subgoals may occur in the query at the Prolog prompt, in the bodies of the rules of the program, and in the directives of the program. A subgoal may refer to a programmer's predicate, or to a built-in predicate, too.

Based on [DM93], the following simple, sufficient condition of the NSTO property of a subgoal is suggested here, which is sophisticated enough for most practical cases.
If a subgoal g referring to a predicate p satisfies any of the following conditions, then this subgoal is NSTO.

1. g does not contain double `var` (a `var` with more than one occurence).
2. No head of any rule of p contains double variable.
3. The actual parameters of g are *simple terms*, and each rule head of p
   - contains just *simple terms* in its formal parameters,
   - or does not contain double variable.

Provided that we want to write an NSTO program, the conditions above call our attention to the critical rule heads. If a head of a rule contains double variable, and some subgoal referring to it contains double `var`, and the rule head or this subgoal contains `compound` parameter, then this goal may be STO, and then the whole program may be STO.

If we find a rule head like this, we may still prove the NSTO property of the goals referring to it, considering their special features. Or we can follow the

method shown in the next example. Let us suppose that we coded the predicate `member_/2` in the usual way:

```
% member_(X,Xs) :- X is member of list Xs (?NSTO?).
member_(X,[X|_Xs]).
member_(X,[_X|Xs]) :- member_(X,Xs).
```

Clearly, the first rule may imply STO property. Its head contains the double variable `X`, and even the compound `[X|_Xs]`. Considering programming practice, in this case the first sufficient condition of the NSTO property is almost always satisfied, as the subgoals of the form `member_(Y,Ys)` contain no double `var`.

However, if some goal of this form might contain double `var`, and we cannot prove that this goal is NSTO (or it is surely STO), we can produce a safe version of `member_/2`, and make this goal refer to it:

```
% safe_member(X,Xs) :- X is member of list Xs (NSTO)
safe_member(X,[Z|_Xs]) :- unify_with_occurs_check(X,Z).
safe_member(X,[_X|Xs]) :- safe_member(X,Xs).
```

In predicate `safe_member/2` goal `unify_with_occurs_check(X,Z)` invokes the appropriate built-in predicate of Prolog, which calculates the *mgu* of `X` and `Z`. (If they do not have unifier, it fails.) For example:

```
| ?- member_(X,[f(Y,Y),f(X)]).
X = f(Y,Y) ? ;
X = f(f(f(f(f(f(f(f(f(...)))))))))) ? ;
no
| ?- safe_member(X,[f(Y,Y),f(X)]).
X = f(Y,Y) ? ;
no
| ?- member_(f(X,X),[f(Y,g(Z)),f(Y,g(Y))]).
X = g(Z), Y = g(Z) ? ;
X = g(g(g(g(g(g(g(g(g(...)))))))))),
Y = g(g(g(g(g(g(g(g(g(...)))))))))) ? ;
no
| ?- safe_member(f(X,X),[f(Y,g(Z)),f(Y,g(Y))]).
X = g(Z), Y = g(Z) ? ;
no
```

In general, if we use compound terms together with a predicate, then we consider the rules containing double variables in their head. If we cannot prove the NSTO property of the goals referring to them, we make the following transformation on these rules: while we find double variable (critical from the point of NSTO property) in the head of the rule (let it be `X`), we rename one of its occurences to a fresh variable (let it be `Z`), and then we insert the subgoal `unify_with_occurs_check(X,Z)` as the first subgoal of the body of the rule.

In this way each predicate invocation of our program becomes NSTO, except the subgoals of the form `unify_with_occurs_check(X,Z)` generated by the program transormation above. Then our Prolog program works as if everywhere mathematical unification were used, but it runs much faster, because mathematical unification is used only when it is necessary. Otherwise, we use its much simpler and much more effective version: pattern matching.

Note: Considering the built-in predicates of Prolog, in most cases the subgoals referring to them are automatically NSTO. Selecting those mentioned in this work, the list of exceptions is the following: (=)/2 and (\ =)/2 (1.7), `arg/3` (1.8.3), `read/2` (1.10.2), `retract/1` and `retractall/1` (1.10.3), `findall/3` (1.11), and `catch/3` (1.12).

The NSTO property of the subgoals referring to the built-in predicates listed here can be checked and handled with a method similar to the one shown here. For example, if `built_in(X,X)` is an unsafe subgoal, in many cases it can be replaced by `built_in(X,Z)`, `unify_with_occurs_check(X,Z)`, where `Z` is a fresh variable. (It migth cause a problem that the search tree of the call `built_in(X,Z)` may be (much) bigger than that of `built_in(X,X)`.)

In the next two subsections we go on with two *de facto standard* optimizations of the Prolog machine. Taking them into consideration, we can significantly reduce the runtime and memory needs of our Prolog programs.

1.6.4 First argument indexing

During goal reduction, first argument indexing may significantly reduce the costs of searching through the rules of the appropriate predicate. According to a rough criteria it selects a subset of the rules of a predicate, appropriately narrowing the search tree. It often selects just one rule, so increases the effeciency dramatically, as generating a choice point and handling it is quite expensive.

However, the predicate invocations of the Prolog programs are often deterministic. This means that just one rule head matches the goal.[10] Provided that the Prolog machine recognises the determinism of a predicate invocation, it generates no choice point. First argument indexing supports it to recognise determinism. Let us see the details: Consider point (3) of the abstract Prolog interpreter described in (1.6.1). We labelled the actual node of the search tree with the list of each of the rules of the predicate to be invoked there. Provided that this list contains more than a single rule, we generate a choice point there, where the search tree branches according to rule heads possibly matching the predicate invocation.

[10] A predicate invocation is called *nondeterministic,* iff more rule heads match it, because in this case the run of a logic program can continue on the different branches of the search tree, although a standard Prolog environment resolves this nondeterminism with backtracking on these branches.

Clearly, it would be enough to put the rules whose head match the subgoal onto this list. However, it would be too expensive to produce this list before we try the branches. Therefore, most Prolog implementations apply a compromise here. This is called *first argument indexing*:

1. Provided that the first actual parameter of the predicate invocation is `atomic` term, only those rules are put onto the list, where the first formal parameter of the rule head is the same `atomic` or a `var`.
2. Provided that the first actual parameter of this invocation is `compound` term, only those rules are put onto the list, where the first formal parameter of the rule head is a `var`, or a `compound` whose functor is identical with the functor of the first actual parameter of the subgoal.
3. Provided that the first actual parameter of this invocation is a `var`, each rule of the predicate to be invoked is put onto the list.

The possible lists of rules are usually generated compile time, and a table is made of them. Then the run time selection of the appropriate list needs minimal, constant time. The variable renaming of the rules is solved like in the case of procedural languages. On the other hand, multiple argument indexing is rarely used, because the size of the table to be generated becomes too large.
Let us suppose that any of the predicates `list/1`, `append/3`, `rev_app/3` (1.5) is called with proper list actual parameter in ist first argument. Then neither its call nor its recursive calls generate any choice point while it runs. Namely, if the first actual parameter is an empty list, we will index on the first rule; and if it is a nonempty list, we will index on the second rule. The run of predicate `member_/2` is not affected by first argument indexing, because the first formal parameters of its rules are variables.

1.6.5 Last call optimization

Normally, each predicate invocation needs some space in the call stack. However, when this optimization is applied, the called predicate can reuse the memory needed by the caller. Eventually, a constant size of memory may be enough to run a recursive procedure.

In order to understand how it works, consider point (5) of the abstract Prolog interpreter described in (1.6.1). If the actual list has contained just one rule there, then after removing this single rule, an empty list of clauses remains. This means that a later backtracking to this node results in another bactracking from here. Therefore in point (8), instead of extending the momentarily represented part of the actual branch of the search tree, the actual node may be overwritten. The set of variables labelling the edge pointing to this node is extended by the set of variables which would label the edge pointing to the new node in the original algorithm. Therefore, in this case the actually represented branch of the search tree does not become longer.

Therefore, given the last subgoal of a rule body, consider the call of this subgoal. If there is no living choice point from the node and time, where and when the predicate containing this rule was invoked, than that node of the optimized representation of the search tree is equal to the node which is the result of invoking this last subgoal. Therefore, in this case this last subgoal can reuse the memory needed by its parent subgoal (i.e. the invocation of the predicate containing this last call). This memory reuse is called last call optimization.

Especially, if this last subgoal invokes the predicate containing it, this is *tail recursion*, and it can work like a loop in procedural languages. Then last call optimization can be simplified to the so called *tail recursion optimization.*

For example, because during the run of the predicates
`list/1`, `member_/2`, `append_/3`, and `rev_app/3` (1.5)
there is no living choice point when they are called recursively, in each case tail recursion optimization can be applied. However, when predicate `reverse/2` invokes `rev_app/3`, last call optimization can be applied.

1.7 Modifying the default control in Prolog

Language pure Prolog containes no possibility to express any form of negation. For example, it is easy to express that two terms can be matched (`A = B`), because we can define it with the universal fact `X = X`. However, we have no tool to express its negation (`A\ = B`).[11]

Similarly, we have defined predicate `member_/2` in (1.5.1), but we have no tool to define predicate `nonmember_/2`. Now it follows, that still we are not able to phrase the union or intersection of two unsorted lists, because we should say what to do when an element of one of the lists is not found on the other list.

In general, our rules are not suitable to derive negative information from our programs. At best, we can decide if a statement follows from our program or not. For example, if the search tree of a query is finite, then we can decide, whether it follows from the program.

For example, given a ground term `X`, and a ground list `Xs`, the search tree of the query `member_(X, Xs)` is finite. Therefore we can decide, if the statement `member_(X, Xs)` follows from our program or not. But this statement is true, *iff* it follows from the program. In practice, there are many similar statements. (For example, consider the queries referring to the list handling predicates of (1.5).) Therefore, if we could tell what to do when a goal were successful, and what to do when it failed, we would have a limited but practical tool to manage negation.

[11] In fact, `(=)/2`, and `(\=)/2` are built-in predicates ([DEDC96] and [ISO95]) of standard Prolog, and we cannot overdefine them. When we use them, we can write them in infix manner, i.e. between their parameters (1.9).

1.7.1 Disjunctions

To define this limited but practical negation, the first step is to introduce disjunctions. We already have implicit disjunction:

```
sibling(X,Y) :- sister(X,Y).
sibling(X,Y) :- brother(X,Y).
```

However, in Prolog we have explicit disjunction, too:

```
sibling(X,Y) :- sister(X,Y) ; brother(X,Y).
```

There is a strong tradition here: the ";" sign used to express explicit disjunction is written with *French spacing*, i.e. there is a blank before it. In addition, it is never put at the end of a line, but at the beginning of the next line.

Both of conjunction ",", and disjunction ";" are right associative. Conjunction "," binds stronger than disjunction ";", so the parentheses in the next example are essential.

```
son(Y,X) :- ( mother(X,Y) ; father(X,Y) ), male(Y).
```

However, the overuse of disjunctions is discouraged in Prolog. A rule should be as simple as possible. The earlier definition in (1.2.2) is more structured, and in many implementations it performs a bit better.

1.7.2 Conditional goals and local cuts

Perhaps the most important control structures are the *conditional goals*. They support structured programming, and define negation as failure. Their basic form is the following:

(*if* −> *then*
; *else*
)

Here the *if*, the *then*, and the *else* parts are arbitrary Prolog goals. The *if* part is the decisive condition. If goal *if* is successful, then the solutions of the conditional goal are the solutions of goal *then*. Otherwise the solutions of the conditional goal are the solutions of goal *else*.

It follows that it is not possible to backtrack into goal *if*. If it is successful, the Prolog machine will calculate just its first solution. For example:

```
% PreCond: Xs and Ys are ground lists.
% union(Xs,Ys,Zs) :-
%     Those members of Xs which are not members of Ys
%     concatenated in order before Ys produce list Zs.
union([],Ys,Ys).
union([X|Xs],Ys,Zs) :-
   ( member_(X,Ys) -> union(Xs,Ys,Zs)
   ; Zs = [X|Us], union(Xs,Ys,Us)
   ).
| ?- union([1,2,3,4,5],[1,3,5],Us).
Us = [2,4,1,3,5] ? ;
no
| ?- union([1,2,3],[1,1,3,3,5],Us).
Us = [2,1,1,3,3,5] ? ;
no
```

The second test shows that just the first solution of condition `member_(X,Ys)` is considered. If we forget the arrow, this happens:

```
malfunctioning_union([],Ys,Ys).
malfunctioning_union([X|Xs],Ys,Zs) :-
   ( member_(X,Ys), malfunctioning_union(Xs,Ys,Zs)
   ; Zs = [X|Us], malfunctioning_union(Xs,Ys,Us)
   ).
| ?- malfunctioning_union([2,3],[1,3,3,5],Us).
Us = [2,1,3,3,5] ? ;
Us = [2,1,3,3,5] ? ;
Us = [2,3,1,3,3,5] ? ;
no
```

The first solution is found twice, because goal `member_(3,[1,3,3,5])` finds number `3` on list $[1,3,3,5]$ twice. The second solution was found, because goal `member_(X,Ys)` is not a decisive condition here. Instead of a conditional goal, we coded just a disjunction.

In the conditional goals, the arrow "$->$" is a *local cut*, and the semicolon is the operator of disjunction. Considering the three operators forming compound Prolog goals, their (decreasing) priority order is "$,\ ->\ ;$". And each of them is right associative. Procedurally speaking, when a disjunction of the form
(*if* $->$ *then* ; *else*)
is called, a choice point with two alternatives is generated. Its first alternative is goal *if* $->$ *then* containing the local cut, and its second alternative is goal *else*. Provided that goal *if* succeeds, the local cut is performed, and it cuts the choice points (if any) left by goal *if*, and the choice point left by the disjunction.

Then goal *then* is called, and there is no possibility to backtrack neither into goal *if* nor to goal *else*. Provided that goal *if* fails, we backtrack, remove the choice point left by the disjunction, and call goal *else*. However, if any of goals *then* or *else* succeeds, it is still possible to backtrack into it, and find its alternative solutions. Because the operator ";" is right associative, goals
(`if1 -> then1 ; if2 -> then2 ; else`), and
(`if1 -> then1 ; ( if2 -> then2 ; else ) )` are equivalent.
We may also omit the " ; `else` " part. Then " ; `fail` " is the default, where `fail/0` is a standard built-in predicate of Prolog, and it always fails. Similar is `true/0`, but it always succeeds.

Now we can solve the problems posed at the beginning of this section (1.7):

```
% \+=(X,Y) :- X does not match Y.
\+=(X,Y) :- ( X=Y -> fail ; true ).

% PreCond: Xs is a proper list.
% does_not_have(Xs,Y) :- no member of Xs matches Y.
does_not_have([],_Y).
does_not_have([X|Xs],Y) :-
   ( X = Y -> fail
   ; does_not_have(Xs,Y)
   ).

% nonmember_(X,Xs) :- does_not_have(Xs,X).
nonmember_(X,Xs) :- ( member_(X,Xs) -> fail ; true ).

% PreCond: Xs and Ys are gound lists.
% intersection(Xs,Ys,Zs) :-
%    those members of Xs which are found also on Ys,
%    form in order proper list Zs.
intersection([],_Ys,[]).
intersection([X|Xs],Ys,Zs) :-
   ( member_(X,Ys) ->
        Zs = [X|Ms], intersection(Xs,Ys,Ms)
   ; intersection(Xs,Ys,Zs)
   ).
```

Notice that the order of the arguments of the recursive predicates
`does_not_have/2`, `intersection/3`, and `union/3`
is chosen to exploit first argument indexing (1.6.4). Tail recursion optimization can be applied to each recursive call (1.6.5).

Note on assignment statements

In predicates `intersection/3`, and `union/3` any goal of the form `Zs = [X|Vs]` tries to *match* its two parameters. Namely, standard Prolog does *not* have any form of *assignment statement*, because any logic variable refers to an unknown term (1.4), [DEDC96], [ISO95], [O'K90] and [SS94]: it refers to an object unknown when the program is coded, but it refers to an object to be computed while the program is being executed. Therefore, when a variable is substituted by a `nonvar`, its value (determined by the relation coded in our program) is computed, and it does not have sense to overwrite it. (If there were an assignment statement like `Xs := [Y|Xs]`, which occurence of `Xs` would stand for the unknown object?)

A typical error of a programmer starting to study logic programming is the use of Prolog goals like `Xs = [Y|Xs]`. This goal will surely fail to perform the assignment desired. For example, if `Xs` is a proper list, the pattern matching between `Xs` and `[Y|Xs]` will clearly fail. If Xs is a `var`, the goal is STO, and a cyclic list will be generated, which is not the desired case now. If `Xs` is a partial list, it may fail or generate a cyclic term wich is not the behaviour desired.

How to get just one solution from a query?

At last we show two solutions for checking whether a term is member of a list. Namely, goal `member_(X,Xs)` may have many solutions, even if `X`, and `Xs` are ground terms (provided that `Xs` has multiple occurences of `X`). After producing a solution, goal `member_(X,Xs)` always leaves a choice point which needs extra memory, and may prevent last call optimization. However, if we need just member checking, we need a goal with at most one solution which does not leave choice points.

Our first solution, predicate `member1/2` shows that a conditional goal can help us even to get rid of unwanted choice points. The second one, predicate `member_check/2` shows that failing rules should not be programmed explicitly. If we want to emphasize that we have not forgotten those cases, it is better to put the failing rules into comment, in order to keep efficiency.

```
member1(X,Xs) :- ( member_(X,Xs) -> true ).

% member_check(_X,[]) :- fail.
member_check(X,[Y|Ys]) :-
    ( X = Y -> true
    ; member_check(X,Ys)
    ).
```

1.7.3 Negation and meta-goals

Let us suppose that predicates `student/1`, and `married/1` are given, and we want to define predicate `unmarried_student/1`. Based on the previous subsection, the following solution springs up.

```
unmarried_student(X) :- student(X), unmarried(X).
unmarried(X) :- ( married(X) -> fail ; true ).
student('Peter'). student('John').
student('James').
married('Peter').   married('Joseph').
```

Notice that the common scheme of predicates
(`\ =`)/2, `nonmember_/2`, and `unmarried/1` follows.

```
not(P) :- ( P -> fail ; true ).
```

The question is, whether this more general scheme of negation is a Prolog predicate or not.

Considering it in a context free manner a Prolog goal is a callable term, i.e. a `compound` term or an `atom`. This makes it possible to compute and call it with the program: The program part computing it handles it just like any `compound` or `atom`. Then it is called like a *meta-goal*. A trivial example: goal `X = p(Y)`, `X` is equivalent with goal `p(Y)`.

Considering a *meta-goal* in the source code in context-free manner, it is a logical variable. But in context-dependent manner, it is a goal: it is part of the body of a rule, query, or directive. The important thing is that by the time it is invoked, it must be substituted by a callable term which can be interpreted as a valid goal of the program. (The appropriate predicates must be defined.)

A meta-goal appearing in the body of a rule may be formal parameter of the head of the rule, like in predicate `not/1` above. The appropriate argument of the head of that rule is a *meta-argument*. A predicate consisting of such rules is a *meta-predicate*. When a meta-predicate is invoked, its meta-arguments can be parameterized by goals. The parameter in a meta-argument is called *meta-parameter*.

For example, predicate `unmarried_student/1` can also be defined as follows.

```
unmarried_student(X) :- student(X), not(married(X)).
```

The only formal parameter of meta-predicate `not/1` above is `P` which is invoked in the rule body as a meta-goal. If `P` (for example `married(X)` in the previous example) is successful, the *local cut* cuts its choice points (if any), and also cuts the *else* branch of the conditional goal. Then on the *then* branch goal `fail` is performed and invocation `not(P)` (for example `not(married(X))`) fails. If `P` (for example `married(X)`) fails, we backtrack to the *else* branch of the conditional goal, goal `true` succeeds, and also invocation `not(P)` (for example `not(married(X))`) succeeds. Summarizing this, goal `not(P)` succeeds, *iff* goal `P`

fails (and therefore there is no solution of goal `P`). And goal `not(P)` fails, *iff* goal `P` succeeds.

This is *negation as failure*. It is *not* a logical negation, but it can be implemented quite effectively,and usually it can be used instead of that, if applied by some care.[12]

Notice the first four properties of goal `not(P)`:

1. It never leaves any choice points.
2. Independently from goal `P`, goal `not(P)` never substitutes its variables.
3. `not(P)` succeeds, *iff* the search tree of `P` is finite, and it contains no solution.
4. `not(P)` fails, *iff* the search tree of `P` contains some solution, but there is no infinite branch before the first solution.

(Notice that these observations are valid even if the negation is implicit, when it is programmed with conditional goals like in the like in the cases of predicates (`\ =`)/2, `nonmember_/2`, and `unmarried/1`.)

It is clear from (2) that a negated goal produces no solution. One cannot use them to produce any (partial) solution, just to test and validate (partial) solutions. According to this, it is not all the same, where the negated goal is inside a compound goal: the desired behaviour of the negated goal usually assumes that its variables have been substituted. For example:

```
| ?- unmarried_student(X).
X = 'John' ? ; X = 'James' ? ; no
```

This is the desired behaviour, independently from the actual one of the two definitions of predicate `unmarried_student/1` above. However, if we exchange the order of the subgoals in its body:

```
| ?- unmarried_student(X).
no
```

Namely, in this case goal `unmarried(X)` or `not(married(X))` fails, because `married(X)` succeeds (provided that `X` is still `var`).

In general, it is a sufficient condition of the logical soundness of Prolog negation that the negated goal must be ground when it is invoked, and its search tree must be finite.[13]

[12] Let us note that although goals `not(married(X))`, and `unmarried(X)` mean the same, in today's Prolog implementations usually `not(married(X))` is the less effective one, because most often the metagoals cannot be optimized while we compile the Prolog source code.

[13] Considering negation we adopt the *closed world assumption*. This means, we suppose that we have enough information: each statement of the actual model which does not follow from our axioms is *false*.

This negation defined with predicate `not/1` is implemented in Prolog more effectively with the built-in predicate (\+)/1. It can be used as a prefix operator like in the next version of predicate `married_student/1`.

```
unmarried_student(X) :- student(X), \+ married(X).
```

1.7.4 The ordinary cut

The local cut introduced in Subsection (1.7.2) is considered usually a better alternative of the more traditional ordinary cut to be discussed here. However, there are millions of lines of Prolog code with ordinary cuts around. In order to allow the reader to understand such code, we are going to present them through a simple example now.

Let us suppose that we want to define the relation `max(X,Y,Z)`, where X and Y are integer values given in advance, and it is true iff Z matches their maximum. Procedurally speaking, if X and Y are integer numbers, the invocation `max(X,Y,Z)` tries to match Z with the maximal one. Let us suppose that the arithmetic relations $'>'/2$ and $'=<'/2$ are given as usual. Our "zeroth" solution does not use any cuts:

```
max0(X,Y,X) :- X > Y.
max0(X,Y,Y) :- X =< Y.
```

Notice that in the first rule we have condition `X > Y` instead of `X >= Y` in order to avoid duplicated solutions. Clearly, this program is correct. But it is not effective: if X and Y are integers, X is greater than Y, Z matches X, and `max0(X,Y,Z)` is invoked, the query will be successful with the first rule of the predicate, and it leaves a choice point referring to the second rule.[14] Thus, if this call is element of a sequence of invocations, and we backtrack to it from a later one, we know that it will fail. Therefore, in this case the subgoal `max0(X,Y,Z)` leaves an *unnecessary choice point* which needs extra space, its handling eats extra runtime, and it may prevent last call optimization(s) in the program using this predicate.

In order to solve the problem of *unnecessary choice points*, the inventors of Prolog introduced a control tool, the (ordinary) *cut* statement into the language. It is denoted by "!". Let us see the improved version of predicate `max/3`:

```
max1(X,Y,X) :- X > Y, !. % green cut after test selects the rule
max1(X,Y,Y) :- X =< Y. % contra test
```

The cut (i.e. "!") statement is a procedural construction: a predicate is considered as a procedure. When it is called, and while it runs, it may generate many choice points, many branches of the program. When the cut is invoked, it tells Prolog that the actual branch of the predicate containing it is the only possible winner. Therefore, the other branches are pruned. We inform Prolog that if any, the rule

[14] For example, consider the subgoal `max0(3,2,M)` where M is a `var`.

containing the cut will perform the calculation associated by the predicate, and in the rule body the subgoals before the cut serve to test this selection. Prolog replies and prunes each choice point generated since the call of the predicate:

1. If a choice point was left when the predicate containing the cut was invoked, and it exists, then it is pruned.
2. All the choice points left by the subgoals preceding the cut in the actual rule body or query are pruned.

Let us consider predicate invocation `max1(X,Y,Z)` where X and Y are integers and Z is a `var`. If `X > Y`, the cut is called, and it prunes the choice point left by invoking the `max1/3`. The test `X > Y` did not leave a choice point. But if it left any choice point, the cut would prune it, too. If the test `X > Y` fails, Prolog backtracks from the first rule, automatically prunes the choice point left by the predicate invocation, and selects the second rule. Then the contra test `X =< Y` succeeds. There is no cut statement after the contra test, because there is no choice point to be pruned. In both cases the predicate invocation succeeds, and it does not leave any choice point, reflecting the fact that there is just one maximum of two numbers.

However, this cut has been inserted into a correct predicate, and it does not alter its semantics. It only implies some optimization in the code. Therefore, it is called a *green cut*. We remain close to "pure logic programming": first we write the program without cuts, and then we insert the green cuts necessary to prune the redundant choice points.

Let us reconsider the predicate invocation `max1(X,Y,Z)` where X and Y are integers. Notice that the first rule fails even if `X > Y`, provided that Z is a `nonvar` different from X. The second rule fails even if `X =< Y`, provided that Z is a `nonvar` different from Y.

Nevertheless, some Prolog predicates contain *red cuts* modifying their semantics. Considering the code of predicate `max1/3`, one may think that the contra test is superfluous: "if `X > Y` fails, and we go to the second rule, it is just waisting time to test whether `X =< Y` stands, because it is surely true". Then the following solution may be proposed.

```
max2(X,Y,X) :- X > Y, !. % red cut
max2(_X,Y,Y).            % contra test omitted, procedural code
```

This is a red cut: without the cut, even goal `max2(3,2,M)` (where M is a `var`) would have two solutions, namely `M = 3` and `M = 2`. With the cut, the only solution is `M = 3`. Therefore, predicate `max2/3` is no more a logic program, just a procedure written in Prolog. There is one even more serious problem. Provided that X and Y are integers, goal `max2(X,Y,Z)` should succeed iff Z matches their maximum. However, the reasoning which is the base of predicate `max2/3` implicitly supposes that condition `X > Y` is always evaluated. Nonetheless it is true only if Z is a `var` or Z is identical with X. Otherwise query `max2(X,Y,Z)` tries the

first rule, but its head does not match it and fails.[15] Then it tries the second rule without evaluating `X > Y`. For example, goal `max2(3,2,2)` succeeds. In general, simply omitting the contra test is dangerous.

Clearly, if we want to spare the contra test, we can use a conditional goal:

```
max3(X,Y,Z) :-
    ( X > Y -> Z = X
    ; Z = Y
    ).
```

Unlike here, the contra test is often expensive, and such solutions can save runtime. Fortunately, if one uses a conditional goal, rules representing different cases are melted into a single one, and the programmer must have formal parameters general enough to cover each case. In this way, the condition is evaluated, and the contra test can be safely omitted, except if the condition part contains more than the condition selecting the appropriate branch of the conditional:

```
max4(X,Y,Z) :-
    ( X > Y, Z = X -> true. % WRONG SOLUTION!!!
    ; Z = Y
    ).
```

This predicate tries to match the output before the local cut, and this matching becomes part of the condition. So we have the same problem with the predicates `max2/3` and `max4/3`: if Z does not match X, the second alternative is tried, independently from condition `X > Y` (although there is no early failure here).

But this kind of error is quite rare. Even beginners intuitively use the conditional goals in the correct way. The main disadvantage of using conditionals is that we have to restructure the "pure logic programs" in order to include the local cuts. Although most programmers do not feel this to be a problem.

Nevertheless, sometimes ordinary cuts can be preferred. Therefore, we will show the safe use of ordinary red cuts. Reconsidering the conditional goals used here and in (1.7.2) it is easy to see that the local cuts are usually red cuts: the control test in the next branch of the disjunction is normally omitted. And the output matching is done normally after the local cut. This is useful even with ordinary cuts. Therefore let us see the correct use of ordinary red cuts:

```
max5(X,Y,Z) :- X > Y, !, Z = X. % output matching after the cut.
max5(_X,Y,Y).      % contra test omitted: procedural, but safe.
```

Clearly, this solution prevents early failure. Condition `X > Y` really selects the appropriate clause. The method can be used in any "deterministic" predicate for pruning unnecessary choice points while avoiding redundant contra tests. (A predicate is called *deterministic* iff for any given data just one branch of it can succeed.)

[15] This is called *early failure*.

The general rule of inserting cuts is this: *cut as early as possible*, i.e. where it is already known that the actual branch of the program is the only candidate to perform the actual computation. If we cut earlier, we select that branch before the tests necessary to select it have been completed. If we cut later, we may try alternative branches of the program, even when these branches should not run.

1.8 The meta-logical predicates of Prolog

In practical Prolog programming we often need information about the actual state of the variables and other components of our program, but such questions cannot be formulated with the tools we already have. We often have to compute the numeric value of an expression, but numeric computations still cannot be performed effectively with these tools. Also we have to make symbolic computations on terms with unknown functors, i.e. terms coming from external sources, but we are able to formulate rules on terms with known functors only. In order to solve such problems, we need meta-logical predicates.

1.8.1 Arithmetic

The usual symbols of arithmetic operations, like $+, -, *, /, **$ etc., are only function symbols in Prolog, and these symbols can be written especially in infix or prefix mode. Therefore they are only constructors of compound data structures, and they force no computation. This is necessary because Prolog supports symbolic computations, for example, the derivation or integration of polynomials like $-\mathtt{x}^2 + 3 * \mathtt{x} - 4$. It follows that, for example, the expression $2 + 3$ means just the Prolog term $+(2, 3)$, and the Prolog environment does not evaluate it, except if it occurs in an *arithmetic argument* of an invocation of an *arithmetic predicate*. These are:

- The right-hand side argument of the built-in predicate `is/2`;
- And both arguments of the arithmetic comparison predicates
  . $(=:=)/2$, $(= \backslash =)/2$, $(<)/2$, $(>)/2$, $(=<)/2$, $(>=)/2$.

When we invoke any of these predicates its name can be used with infix notation. Goal `Term is Exp` first evaluates the arithmetic expression `Exp`, next it tries to match the result with `Term`. An arithmetic comparison compares the arithmetic values of its parameters.

An arithmetic expression is a number (`integer` or `float`), or a compound term with arithmetic functor, and arithmetic expressions in the arguments. (The arithmetic functors are the usual arithmetic operators, logarithmic, trigonometric function symbols etc. [C+12], [DEDC96] and [ISO95]) This means that the variables of the *arithmetic arguments* of a Prolog goal must be appropriately

substituted by the time of invoking this goal. For example (// denotes the integer divison, `floor` denotes the integer part):

```
| ?- X is -2**4, Y is floor(cos(0))+3*(8-7//2), Z is -2+1.0.
X = 16.0,  Y = 16,  Z = -1.0
| ?- X = -2**4, Y = floor(cos(0))+3*(8-7//2), Z = -2+1.0.
X = -2**4,  Y = floor(cos(0))+3*(8-7//2), Z = -2+1.0
| ?- 5 is 2+3, 5.0 =:= 2+3, -4+1 =< -2-1.0.
yes
| ?- 2+3 is 2+3.
no % 5 does not match 2+3 which is a compound.
| ?- 5.0 is 2+3.
no % 5 does not match 5.0 although they are arithmetically equal
```

A typical error of Prolog beginners is the goal "`X is X + 1`". Nonetheless this goal never performs the operation desired. If `X` has been substituted by a number, the value of `X + 1` does not match `X`, because these are two different constants. If `X` is a compound arithmetic expression, the value of `X + 1` does not match `X`, because a constant never matches a compound. If `X` is not an arithmetic expression (for example, `X` is a `var` or `atom`), the evaluation of `X + 1` raises the appropriate exception (1.12).

Note that the *arithmetic comparisons* just compare the values of proper arithmetic expressions. For example, goal `Y =:= X + 1` cannot be used to get the result of `X + 1`. A goal like `Y is X + 1` serves for this purpose:

```
| ?- Y is 6+1.
Y = 7
| ?- Y =:= 6+1.
{INSTANTIATION ERROR: _157=:=6+1 - arg 1}
```

Finally, we consider some classic arithmetic computations. The next two predicates calculate the sum and scalar product of two vectors represented by lists of numbers of the same length.

```
% add(V1,V2,V) :-
%     Vector V is the sum of vectors V1 and V2.
add([],[],[]).
add([X|Xs],[Y|Ys],[Z|Zs]) :-
   Z is X+Y, add(Xs,Ys,Zs).


% mult(V1,V2,S) :-
%     S is the scalar product of vectors V1 and V2.
mult([],[],0).
mult([X|Xs],[Y|Ys],S) :-
   mult(Xs,Ys,S0), S is S0+X*Y.
```

The predicates above are deterministic according to first argument indexing. In case of `add/3` even tail recursion optimization can be applied, and it is not hard to tranform `mult/3` accordingly: we introduce an accumulator in order to collect the partial sums, that is, we generalize the original problem in order to add the scalar product to some initial value.

```
mult(V1,V2,S) :- mult(V1,V2,0,S).

% mult(V1,V2,A,S) :- S = A+V1*V2.
mult([],[],A,A).
mult([X|Xs],[Y|Ys],A0,S) :-
    A1 is A0+X*Y, mult(Xs,Ys,A1,S).
```

1.8.2 Type and comparison of terms

The standard types of Prolog terms are organized according to the hierarcy of the different kinds of terms introduced in (1.4).

The list of standard types: `var`, `nonvar`, `atomic`, `number`, `float`, `integer`, `atom`, `compound`.

The names of these types are the names of the standard type-checking predicates, too. Each of them has arity 1. For example, the following test is successful.

```
| ?- var(X), var(Y), var(Z), var(U), X=1, Y=1.0, Z=a, U=f(a),
     nonvar(X), nonvar(Y), nonvar(Z), nonvar(U), compound(U),
     atomic(X), atomic(Y), atomic(Z), atom(Z),
     number(X), number(Y), integer(X), float(Y).
```

In the next example the effectivity of predicate `grandparent/2` (1.2.2) is increased: If the grandson is known, and the grandparent is to be computed, we exchange the order of the subgoals, in order to narrow the search tree. Otherwise, we leave the original order and take advantage of first argument indexing.

```
grandparent(X,Y) :-
    ( var(X), nonvar(Y) -> parent(Z,Y), parent(X,Z)
    ; parent(X,Z), parent(Z,Y)
    ).
```

In Prolog two arbitrary terms are comparable. The identity of two terms can be tested with the built-in $(==)/2$ and $(\backslash ==)/2$. For example, the following test is successful, because the two logical variables are originally different, but having matched them they are identical: `| ?- X \== Y, X = Y, X == Y.`

In the **standard order of terms** a `var` is *smaller* than a `float`, this is *smaller* than an `integer`, this is *smaller* than an `atom`, and this is *smaller* than a `compound`. goal `A @< B` is succcessful **iff** `A` precedes `B` in the standard order of terms, that is `A` is *smaller* than `B`

In the standard order, the *floats* are compared arithmetically, and the *integers*, too. But do not forget that a `float` is always *smaller* than an `integer`:

```
| ?- 5 < 5.1, 5.1 @< 5, 9.9e99 @< -999999999.
yes
```

The names or *atoms* are compared lexicographically. And their characters are compared according to their code values, in the actual coding system (for example, latin-1, utf-8, etc.). The structures or *compounds* are compared first according to their *arities*, next according to their *functor names*, and at last according to their parameter lists, lexicographically. The parameter pairs are compared according to the standard order of terms, recursively.

The standard order of the *variables* is implementation defined. The standard order of two terms may be changed by substitution, if any of them is a `var`, or any of them is a `compound` containing some `var`(s). For example, the following test is successful.

```
| ?- X@<10.0, X=1, X@>10.0.
```

According to their standard order the terms can be compared with the built-ins (@ >)/2, (@ <)/2, (@ >=)/2, (@ =<)/2. For example:

```
% PreCond: Ys is a proper list sorted increasingly
%         according to the standard order.
% sorted_insert(Ys,X,Zs) :-
%    Zs is received by the sorted insert of X into Ys.
sorted_insert([],X,[X]).
sorted_insert([Y|Ys],X,Zs) :-
   ( X @=< Y -> Zs = [X,Y|Ys]
   ; Zs = [Y|Us], sorted_insert(Ys,X,Us)
   ).
| ?- sorted_insert([b(z(1,2)),a(X,Y),a(2,1)],a(2,3.14),Zs).
Zs = [b(z(1,2)),a(X,Y),a(2,3.14),a(2,1)]
```

1.8.3 Term manipulation

Up till now we have supposed that the functors, i.e. constructors of the terms processed by our programs are known in advance. However, in many applications this is not true (for example, when we process Prolog terms coming from an input file or channel). In such cases the input terms can be analized and new terms can be synthetised using the built-in predicates manipulating terms.

In order to handle `compound` terms constructed with unknown functors, the most important built-ins are `functor`/3, and `arg`/3.

If `Term` is a `nonvar`, we can determine or test the functorname and/or arity of `Term` using the invocation `functor`(`Term`, `Functorname`, `Arity`). (The arity of an `atomic` is zero.) If `Term` is a `var`, `Functorname` is an `atom`, and `Arity` is a

positive `integer`, the invocation generates an appropriate `compound` term with fresh variables in its arguments and substitutes it into `Term`:

```
| ?- functor(t(a,b),F,N), functor(T,F,N).
F = t,   N = 2,   T = t(_A,_B)
```

The goal `arg(I,Structure,Arg)` matches `Arg` with the $\texttt{I}^{\text{th}}$ parameter of the `compound` term `Structure`, where `1 =< I =< Arity`. Therefore this predicate can *read* and/or *fill* the parameters of a `compound`:

```
% substitute(T0,X,Y,T) :-
%    T is a copy of T0 except that each occurence of X in T0
%    has a corresponding occurence of Y in T.
substitute(T0,X,Y,T) :-
   ( T0 == X -> T = Y
   ; compound(T0) ->
      functor(T0,F,N), functor(T,F,N),
      substitute_args(N,T0,X,Y,T)
   ; T = T0
   ).
substitute_args(N,T0,X,Y,T) :-
   ( N > 0 ->
      arg(N,T0,A), substitute(A,X,Y,B), arg(N,T,B),
      N1 is N-1, substitute_args(N1,T0,X,Y,T)
   ; true
   ).

% Test:
| ?- substitute(a(X,nil,b(nil,2,B,nil)),nil,[],T).
T = a(X,[],b([],2,B,[]))
```

Constants, i.e. `atomic` terms can be taken into pieces and put together using the built-in predicates
`atom_codes(Atom,ListOfCharacterCodes)`, and
`number_codes(Number,ListOfCharacterCodes)`.
(Futher term manipulators in [DEDC96], [ISO95] and [C+12].) Using these predicates, any textual manipulation of a constant can be reduced to manipulating

the appropriate list of character codes. For example:

```
% aA(A,B) :- B is a copy of atom A except that
%     each lower-case letter in A is substituted
%     by the appropriate upper-case letter in B.
aA(A,B) :-
   atom_codes(A,Cs), cC(Cs,Ds), atom_codes(B,Ds).

cC([],[]).
cC([C|Cs],[D|Ds]) :- bB(C,D), cC(Cs,Ds).

bB(C,D) :- % the code of character a is 0'a
   ( 0'a=<C, C=<0'z -> D is C-0'a+0'A
   ; D = C
   ).

% Test:
| ?- aA('How beautiful is She!',Unknown).
Unknown = 'HOW BEAUTIFUL IS SHE!'
```

1.9 Operator symbols in Prolog

We have mentioned that the names of some predicates (like the names of arithmetic comparisons) can be written in infix mode. Built-in negation can be written like a prefix operator (1.7.3). And some arithmetic function symbols (like $+, -, *, /$) can be written as operator symbols. However, considering these predicate names and function symbols in a context-free manner, they are just functors of `compound` terms. This means that the notion of *operator (symbol)* in Prolog is just notational convenience. It is a completely syntactic category. A Prolog operator *does not do anything*, unlike the operators of the procedural (C,C++, etc.) and functional (Lisp, Haskell, etc.) languages. It is just a syntactic sugar, an optional notation, like the list notation, although it is very important: without operators it is hard to write easy-to-read Prolog programs.

An *operator symbol* is *predifined* or *user-defined*, and it has three basic properties: *priority*, *mode*, and *name*. Its name is a Prolog `atom`.

There are 1200 *piority* levels. The operators of the 1^{st} level bind the strongest and those of the 1200^{th} level bind the weakest. For example, the infix operators $+$, and $*$ have priorities 500 and 400. Therefore $\texttt{a}+\texttt{b}*\texttt{c} == +(\texttt{a}, *(\texttt{b}, \texttt{c}))$. Each Prolog expression has priority, which is zero, if it is a `var` or `atomic`, it is directly in brackets, it is written with list notation, or it is a `compound` written in the standard functional notation in the form $f(t_1, \ldots, t_n)$. The priority of an expression is equal to the priotity of its main functor, if this functor is written as

operator, and the expression is not directly in brackets. For example, the priority of `a + b * c` is 500.

Considering roughly the *mode*, an operator can be *infix* (if it is written between the operands), *prefix* (if it is put before the operand), or *suffix* (if it comes after the operand). At the same time two operators may exist with the same name. In this case, one of them is infix, and the other is prefix or suffix.

Let us consider first the infix operators. And let us suppose that p and q are two infix operators where $pr(p)$ and $pr(q)$ are their priority levels, respectively. Let us consider the expression $apbqc$. If $pr(p) < pr(q)$ then $apbqc = (apb)qc$, because p binds stronger. If $pr(p) > pr(q)$ then $apbqc = ap(bqc)$, because q binds stronger. If $pr(p) = pr(q)$, the interpretation of $apbqc$ depends on the associativity of the operators. There are three different associativities.

The exact mode or associativity of an infix operator is
yfx (left-associative, for example `a - b - c` == `(a - b) - c`), or
xfy (right-associative, for example `(a; b; c)` == `(a; (b; c))`), or
xfx (non-associative: `a = b = c` must be parenthesized explicitly).

In this notation f symbolizes the functor (i.e. operator) name, x and y symbolize the parameters; y=yes: on this side of the operator there must be an expression with the same or smaller priority level. x=no: on this side of the operator there must be an expression with smaller priority level, and an expression with the same priority level is not allowed. However, there are some ambiguous expressions yet: if the right side of the first operator and the left side of the second one are denoted by y, too. In these cases, the expression is considered right-associative.

Based on these rules, this is the default bracketing of $apbqc$, provided that $pr(p) = pr(q)$, $m(p)$ denotes the mode or associativity of operator p, and ? stands for x or y:

- $apbqc = (apb)qc$ **if** $m(q) = yfx \wedge (m(p) = yfx \vee m(p) = xfx)$,
- $apbqc = ap(bqc)$ **if** $m(p) = xfy \wedge m(q) = ?f?$,
- $apbqc$ must be explicitly bracketed in any other case.

The exact mode or associativity of a prefix operator is fx or fy. And that of a suffix operator is xf or yf. The meaning of f, x, and y is the same as before.

It follows that **if** $pr(p) = pr(q)$,

- $pbq = (pb)q$ **if** $m(q) = yf \wedge m(p) = fx$,
- $pbq = p(bq)$ **if** $m(p) = fy \wedge m(q) = ?f$,
- pbq must be explicitly bracketed **if** $m(p) = fx \wedge m(q) = xf$.

The default bracketing of expressions like $apbq$, $pbqc$, etc. goes in a similar way.

A new operator is created by the statement `op(Priority, Mode, Name)`, or it can be overdefined in the same way.[16] (`Name` can be an `atom` , or a proper list

[16] `op/3` is an example of extra-logical predicates (1.10).

of atoms.) If `Priority` == `0`, the operator with the given name and mode is deleted. This statement is most often used in directives (see 1.2). For example:

```
:- op( 1200, xfx, [ :-, -> ]).
:- op( 1200, fx, [ :-, ?- ]).
:- op( 1100, xfy, [ ; ]).
:- op( 1050, xfy, [ -> ]).
:- op( 1000, xfy, [ ',' ]). % The priority of ',' is 1000.
:- op( 900, fy,  \+ ).
:- op( 700, xfx, [ =, \=, ==, \==, @<, @>, @=<, @>=,
              =. ., is, =:=, =\=, <, >, =<, >= ]).
:- op( 500, yfx, [ +, -, /\, \/ ]).
:- op( 400, yfx, [ *, /, //, mod, rem, «, » ]).
:- op( 200, xfx, ** ).  :- op( 200, xfy, ^ ).
:- op( 200, fy, [ -, \ ]).
```

Now we have introduced the predefined operators of Prolog according to the ISO standard ([DEDC96] and [ISO95]).

Notice that the comma symbol is a predefined right-associative infix operator, too. An extra rule: if it is used as infix operator, it must be written without apostrophes. However, commas separate the elements of a list-like term, and the expressions of a parameter list, too. Therefore, in order to know the right interpretation of the operator and separator commas in a Prolog expression, the priority of an element of a (parameter) list must be less then 1000. (If the priority of such an element is greater or equal to 1000, it must be put between brackets.

We have already known that each parameter of any atomic formula is a term in mathematical logic, logic programming, and Prolog. Notice now that the signs used to formulate the Prolog sentences ($':-'$, $','$, $';'$, $'->'$, and $'\backslash+'$) are predefined operators. This means that the goals, queries, heads and bodies of rules, and even the sentences (rules, directives, and declarations) of the Prolog programs are terms besides the parameters of atomic formulas. For example, term `p(X)` can be a piece of data, a query, a head or body of a rule, or a fact of a program depending on its context.

Another example: if negation were not a built-in predicate, we could define it with the following rule.

```
\+ Goal :-
    ( Goal -> fail
    ; true
    ).
```

In this rule, the first occurence of logical variable `Goal` means a piece of data, especially a formal parameter, but its second occurence denotes a Prolog goal. (Notice that in this case the brackets of the conditional goal can be omitted, but in order to increase the readability of the programs, we will always use such brackets.) Because a rule considered in a context-free manner is just a term,

using the functional notation of compound terms, the previous rule could be written as follows (although this notation would destroy readability).

```
:-(\+(Goal),;(->(Goal,fail),true)).
```

We can conclude that in Prolog there is no strict difference between program and data. This property of the language helps us to write language processors, programs manipulating other programs like interpreters, compilers, program transformators, intelligent programs which are able to learn and forget things, and so on. Surely, it is the easier case of the language processors when the sentences of the source and target languages are also fitting the term notion of Prolog ([SB04] and [War80]). (If this is not the case, we can help us with *logic grammars* [DM93], [C$^+$12], [SS94], [SB04] and [War80].)

1.10 Extra-logical predicates of Prolog

These predicates do not have declative reading like pure Prolog programs. These have *only* operational semantics. Theoretically, they lie outside the logic programming model [SS94]. The predicates of pure Prolog, and the meta-logical predicates communicate with their programming environment only through their parameters, and their effects are backtrackable. The extra-logical predicates access global information, usually have side-effects, and typically these side-effects are not backtrackable. Therefore they allow us to pass information among the different branches of the search tree of a Prolog query, so we need them, if we want to write practical applications. They are responsible for the program interfaces, and for the access and manipulation of the Prolog environment. The different kinds of the interface predicates are those loading (and saving) programs, the I/O predicates, the foreign language interfaces (C/C++, Java, .NET, Tcl/Tk, etc.), data base handler, GUI, operating system, web, and other interfaces ([C$^+$12] and [SB04]).

In this chapter we consider only loading programs, I/O predicates, and manipulating the program loaded.

1.10.1 Loading Prolog programfiles

Loading a program means loading a Prolog source or object code into the Prolog environment (where it is activated through queries).

The only standard predicate of this category is `ensure_loaded/1`, whose actual parameter can be a filename, or a proper list of filenames. For each file it checks the time of its last modification, and loads it only if necessary. Some Prolog implementations may allow the use of this predicate only in *directives*, but usually it can be invoked without restrictions, even during the run of a query. It compiles the loaded code into the memory.

Most Prolog environments still provide the traditional predicates of loading in interpreted mode (`consult/1`), and in compiled mode (`compile/1`); file-to-file compilation producing object code, loading object code; there are special predicates for loading module files (`use_module/2` ...(1.13.1)), etc. ([C+12] and [SB04]).

The run of the predicates loaded in interpreted mode can traced (`trace/0`, `notrace/0`, etc.), while loading in compiled mode results optimized code 5-20 times faster ([C+12] and [SB04]).

1.10.2 Input and output

We consider only the handling of sequential textfiles, and only its basic predicates ([DEDC96], [ISO95] and [C+12]). A textfile can be opened in `read`, `write`, or `append` mode, using the query `open(File,Mode,Stream)`. Then our operations refer to the `Stream`, until we close it using `close(Stream)`. The standard I/O streams should not, and must not be closed. They can be referred to through the name `user`. The I/O predicates read or write a single character or a whole term with no restriction on the complexity of that term.

Consider first the character I/O. The goal `peek_code(S,C)` is true if the code of the actual character of stream `S` is `C` (it tries to match it with `C`). The goal `get_code(S,C)` is similar, but as a side-effect, the actual character becomes the next one, even if the code of the old actual character does not match `C`, and `get_code(S,C)` fails. The goal `at_end_of_stream(S)` is true, if stream `S` has already been read until its end (`S\ = user`). The goal `put_code(S,C)` writes the character with code `C` on stream `S`, while `nl(S)` sends a newline onto `S`. E.g.:

```
% PreCond: F1 and F2 are textfiles with appropriate access.
% appf(F1,F2) :- the content of F2 is inserted at the end of F1.
appf(F1,F2) :-
    open(F1,append,A), open(F2,read,R),
    af(A,R),
    close(A), close(R).


af(A,R) :-
    get_code(R,C),
    ( C == -1 -> true     % end of R
    ; put_code(A,C), af(A,R)
    ).
```

Consider now the term I/O. The goal `read(S,T)` reads a whole term from stream `S`. Then it tries to match it with `T`. Because any term can be continued with an infix operator and a connected term, the term to be read must be terminated with a dot, and at least one whitespace character (like the sentences of a program):

```
| ?- read(user,X), read(user,Y).
|: 12.  a+b*c.
X = 12,  Y = a+b*c
```

The goal `write(S,T)` writes term `T` on stream `S` without the terminating dot. (Term `T` can be any term, even a whole list.) But the atoms are never put between quotes or backquotes, because these usually represent messages to be printed:

```
| ?- write(user,'Value of Pi = '), read(user,Pi).
Value of Pi = 3.14159265358979323846.
Pi = 3.141592653589793
```

If we need an output readable for other Prolog programs, we use `writeq(S,T)`:

```
| ?- writeq(user,father('Isaac','Jacob')), write(user,'.\n').
father('Isaac','Jacob').
```

1.10.3 Dynamic predicates

Up till now we have considered only static Prolog predicates: a predicate is static by default, i.e. its code cannot change when it has been loaded. However, a predicate may be defined dynamic. For example, if we have the following declaration: `:-dynamic(p/2).` predicate `p/2` will be dynamic, i.e. its code will be variable even while the program runs. In this way, the running program can dynamically learn new rules and forget obsolete ones.

A decaration must precede the first rule of the predicate referred to. But a declared (and therefore existing) predicate does not necessarily have any rule. If there is a predicate with no rule, each goal invoking this predicate silently fails. But any goal invoking a non-existing predicate raises an exception called `existence_error` (1.12).

We can add new rules to a dynamic predicate. Goal `asserta(R)` adds a fresh copy of rule `R` to the predicate identified by the head of `R` as its new first rule, while `assertz(R)` adds this copy of `R` to that predicate as its new last rule.[17] Let us notice that this making a fresh copy of `R` is necessary because the further run of the program may substitute some logical variables of term `R`, and later backtracking may also delete some actual variable bindings in term `R`. Clearly such modifications of term `R` must not modify the rule added to the program.

Consider now a simple example of dynamic predicates: let us suppose that file `lpp.pl` contains predicate `connected/2`:

[17] A fresh copy is a copy of the term where the variables are substituted by fresh (i.e. new) variables. If a variable has more occurences, the same fresh variable is used for each occurence, but the different variables are substituted with different fresh ones. And the variable bindings to nonvar terms (i.e. variables substituted by nonvar terms) inside the original term are substituted with direct references to the appropriate terms in the fresh copy.

```
:- dynamic(connected/2).
connected(X,X).
```

and it is processed as follows. (We suppose that predicate `edge/2` defines an acyclic directed graph.)

```
| ?- consult(lpp).
% consulting c:/documents and settings/pl/book/lpp.pl...
% consulted c:/documents and settings/pl/book/lpp.pl
%          in module user, 0 msec 1536 bytes
yes
| ?- assertz((connected(X,Y):-edge(X,Z),connected(Z,Y))).
yes
| ?- listing(connected/2). % print the clauses of the predicate
connected(X, X).
connected(A, B) :-
       edge(A, C),
       connected(C, B).
```

A dynamic rule may be deleted by a goal like `retract(RulePattern)` where the *head* part of `RulePattern` must be a Prolog `atom` or `compound` idetifying the dynamic predicate referred to by the `retract` statement. This goal deletes the first rule of the predicate referred to which matches term `RulePattern`. Forcing backtracking even each rule matching `RulePattern` can be deleted:

```
| ?- retract((connected(X,Y):-Body)).
Y = X,   Body = true ? ;
Body = edge(X,_A),connected(_A,Y) ? ;
no
```

Based on this example – if it were not a built-in predicate – we could define predicate `retractall/1` as follows.

```
retractall(P) :- retract((P:-_)), fail.
retractall(_P).
```

If we call any of the program manipulating predicates above, and it refers to a non-existing predicate, then this predicate is created as a dynamic one:

```
% my_consult(F) :- read the sentences of file F,
% omit the declarations and directives, and
% create dynamic predicates from the rules.
my_consult(F) :-
   open(F,read,S),
   consulting_loop(S),
   close(S).
consulting_loop(S) :-
   read(S,Sentence),
   ( Sentence == end_of_file -> true % end of S
   ; Sentence = :-(_) -> consulting_loop(S)
   ; assertz(Sentence), consulting_loop(S)
   ).
```

Dynamic predicates are typically used to memorize lemmas and negative lemmas derivable from the actual run of a program, in order to avoid recomputing this information ([SS94], [O'K90] and [SB04]).

For example, if predicate `edge/2` defines a graph with complex structure, and we are interested, which nodes can be reached from a given one, the following program may be useful, because it remembers the visited nodes, it avoids infinite looping, and if it fails to achieve goal from an internal node, after backtracking and arriving at it again it does not recompute the pathes going out from that node. (Note: `ground/1` may not be a built-in in your Prolog, but it is easy to program it: see Exercise 1.7 in Section 1.7.)

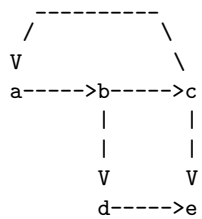
```
% Z can be reached form a given A.
reach_from(A,Z) :-
   ground(A), % A must be ground term
   retractall(visited(_)), asserta(visited(A)),
   reach_2(A,Z).

% do not call directly:
reach_2(A,A).
reach_2(A,Z) :-
   edge(A,B), \+ visited(B),
   asserta(visited(B)), reach_2(B,Z).
```

However, we cannot encourage the use of dynamic predicates to emulate global variables, especially if their usage can be avoided. This can lead to procedural programming style. And the overuse of dynamic predicates implies high runtime costs. For example, each modification of a dynamic predicate forces rebuilding the tables used for first argument indexing, each *assert* call makes a fresh copy of the whole rule to be inserted, independently from the size of the data structures stored in it, etc.

Especially in our previous problem, if we need even the pathes going form node A to Z, or the structure of the graph is not so complex, or node A may not be given in the query, the next solution is suggested.

```
% Example graph:
edge(a,b). edge(b,c). edge(b,d).
edge(c,a). edge(c,e). edge(d,e).
```



```
% path(A,Z,Path) :-
%     Path is an acyclic list of nodes from A to Z.
path(A,Z,Path) :- path_2(A,[],Z,Path).
path_2(A,Ancestors,A,Path) :-
   reverse([A|Ancestors],Path).
path_2(A,Ancestors,Z,Path) :-
   edge(A,B), B\=A, \+member(B,Ancestors),
   path_2(B,[A|Ancestors],Z,Path).
```

1.11 Collecting solutions of queries

A Prolog query may have many solutions, but these are at leaves of different branches of the search tree of the query. Therefore, we can receive these solutions through backtracking, and the Prolog environment forgets one, when it backtracks to search for another. So we would never have them all together, but Prolog has built-in predicates ([DEDC96], [ISO95], [O'K90], [C$^+$12], [SS94] and [SB04]) for collecting the solutions. The most important, and also the simplest one is `findall(Solution,Goal,Results)`, where `Goal` is any query (if it is a conjunction or disjunction, it must be bracketed), `Solution` is the term to be collected from the solutions of `Goal`, and `Results` is the list of the collected terms. Usually `Goal` and `Solution` shares variables. Typically `Solution` is one of the variables of `Goal` or it is a `compound` consisting of some variables of `Goal`. Procedurally speaking, the following algorithm computes `Results`.

```
Results := [];
while( still there is (another) solution of Goal )
   { solve Goal; % substituting some vars of Solution
     append a copy of Solution to the end of Results;
     backtrack Goal; % deleting the bindings of its vars
   }
```

For example, using the example programs in the previous subsection (1.10.3):

```
| ?- findall(X,edge(X,Y),Xs).
Xs = [a,b,b,c,c,d]
| ?- findall(e(X,Y),edge(X,Y),XYs).
XYs = [e(a,b),e(b,c),e(b,d),e(c,a),e(c,e),e(d,e)]
| ?- findall(Z,reach_from(a,Z),Zs).
Zs = [a,b,c,e,d]
| ?- findall(Z,reach_from(d,Z),Zs).
Zs = [d,e]
| ?- findall(Path,path(_,_,Path),Ps).
Ps = [ [_A],[a,b],[a,b,c],[a,b,c,e],[a,b,d],[a,b,d,e],
       [b,c],[b,c,a],[b,c,e],[b,d],[b,d,e],
       [c,a],[c,a,b],[c,a,b,d],[c,a,b,d,e],[c,e],[d,e] ]
```

Considering the first test of `findall(Solution, Goal, Results)`, we can see that the multiple solutions of `Goal` are put on the list `Results` in general. If we want to eliminate the duplications, we need something like predicate `sort(Xs, Ys)` which sorts list `Xs` into strictly increasing order according to (@ <)/2 (removing duplications) and try to match the result with `Ys`. (It is a built-in of most Prolog implementations. Anyway, it is not hard to write our own version: see `unionsort/2` in Section 1.6 in the solution of Exercise 1.6. `sort(Xs, Ys)` is very effective: its computational complexity is $\Theta(n * log(n))$ where n is the length of `Xs`.) Predicate `collect/3` is similar to `findall/3` except that it removes the multiplications of the solutions:

```
% collect(Solution,Goal,Results) :- Result is a
%     strictly increasing proper list of the Solutions of Goal.
collect(Solution,Goal,Results) :-
   findall(Solution,Goal,ListOfSols), sort(ListOfSols,Results).
```

Considering the directed graph defined by predicate `edge/2` we can compute the set of nodes having successor as follows (compare to the first test of `findall/3`):

```
| ?- collect(X,edge(X,_Y),Xs).
Xs = [a,b,c,d]
```

1.12 Exception handling in Prolog

Error and exception handling is standard part of modern programming languages. Therefore, it is also important part of the ISO Prolog standard.

In Prolog a query typically fails, succeeds (possibly leaving one or more choice points) or goes into infinite recursion. However, sometimes a predicate invocation cannot be interpreted. For example, this is the case with an invocation referring to a non-existing predicate (in the given scope), the query `arg(I,T,A)` if `I` is a negative `integer` or `T` is a *simple term* (see Section 1.4), the goal `X is Exp` where `Exp` is not an arithmetic expression, and goal `read(S,X)` if it finds a non-term in `S`. In such cases the invocation raises the appropriate *exception* like in other high-level programming languages. And the call finishes with this *exception* instead of success or fail.

A *Prolog exception* can be represented with an `atomic` or `compound` (i.e. a `nonvar`). It can be raised also with the call `throw`(*exception*). The unhandled exceptions become *runtime errors*. However, the Prolog development environment handles these errors (at the level of its interactive shell) printing the appropriate error message, and the Prolog prompt (| ?−).

If a Prolog goal raises an exception but does not handle it, we say that the goal *propagates* the exception. Therefore, we can say that *runtime errors* are exceptions propagated to the Prolog shell.

Exceptions can be handled with the built-in predicate
`catch`(*Goal*,*ExceptionPattern*,*HandlerGoal*).

A *catch* invocation may handle just the exceptions raised (but not handled) during the evaluation of the *Goal* in its first argument. Therefore, the first actual parameter of a *catch* invocation is a *protected goal*.

A *catch* invocation first calls its first parameter.

If *Goal* propagates no exception, then it works as if it were called bare (without the *catch* protecting it).

However, if *Goal* propagates an exception E, first a fresh copy F of E is made. Next, each variable bindings performed through the evaluation of *Goal* are deleted, together with each choice point, each call frame pushed into the call stack, and each backtrackable events. (However, the side-effects of the extra-logical predicates are not deleted.)

Next F and *ExceptionPattern* are matched.

If this pattern matching is successful, then *HanderGoal* is called, and its evaluation is equal to the further evaluation of the *catch* invocation.

Otherwise the *catch* invocation propagates exception F.

The standard built-in predicates always raise exceptions of the form `error(IsoErr, ImpErr)` where term `IsoErr` is defined by the standard, while `ImpErr` is implementation dependent. For example:

```
| ?- catch( X is a, error(IsoErr,ImpErr), true ).
ImpErr = domain_error(_A is a,2,expression,a),
IsoErr = type_error(evaluable,a/0)
```

In the next example, predicate `read1term(Term)` can read any term (terminated by a dot and a whitespace character) from the standard input, and skips the remainder of the actual input line. Nonetheless if the standard input starts with a non-term (a term with syntax error), it prints the error-term defined by the ISO standard, and repeats reading.

```
read1term(Term) :-
    catch( ( read(user,Term), skip_line(user) ) ,
           error(Err,_), ( write1term(Err), read1term(Term) )
         ).

write1term(Term) :- writeq(user,Term), nl(user).

| ?- read1term(Term).
|: a*(b+ .
syntax_error('. cannot start an expression')
|: a*(b+c.
syntax_error(') or operator expected')
|: a*(b+c).
Term = a*(b+c)
```

1.13 Prolog modules

Although the "general core" of ISO Prolog ([DEDC96] and [ISO95]) is widely accepted and supported by Prolog implementors, and there is also an ISO standard for Prolog modules [ISO00], this latter is considered a failure, and as far as we know, there is no implementation of this part of the standard. Therefore, here we introduce only the module system of SICStus Prolog [C$^+$12] which is flat and predicate-based: it is widely accepted, and it is a starting point of many new developments.

1.13.1 Flat, predicate-based module system

Predicate-based module system means that the Prolog terms are global, and visible in each module. But each module has its own predicate space, and the predicates are local to their module by default. However, they can be defined as

public predicates, and then they can be imported (from their module) into other modules, becoming part of these modules, and accessed directly in them.

Flat module system means that there is no hierarchy, and all the modules of a program are at the same level. One file may contain just one modul, but one module may be spread into several files.

There is a default module called `user`. Each source file is loaded into it by default. The default type-in module of the queries at the Prolog prompt is also module `user`. Therefore, if one does not want to use the module system, one does not have to use it. However, programming in large makes it necessary.

The first sentence of a module file must be a module declaration in the following form.
: − `module`(*ModuleName*, *PublicPredicates*).
ModuleName must be a Prolog atom. *PublicPredicates* must be a proper list of predicate specifications in the form *name/arity*. The predicates encountered here are the *public* predicates of the module. The predicates of a module may be defined in that module. Or these may be imported into it, using, for example, one of the the built-in predicates
`use_module`(*ModuleFileName*, *ImportList*)
`use_module`(*ModuleFileName*)
called typically in a directive (1.2). *ImportList* must represent a subset of the public predicates of the module defined in file *ModuleFileName*. If we invoke `use_module(MFN)` in module `M`, each of the public predicates of the module defined in file `MFN` is imported into module `M`. These calls load the file *iff* it is necessary.

For example, let us suppose that we have the module file `graph.pl` with the module `graph_of_edges` in it. The only public predicate of this module is `edge/2` defining the edges of the graph. We also suppose that goal `edge(X,Y)` can check or return an edge, that is, there is no restriction on its use. For example:

```
:- module( graph_of_edges, [edge/2] ).
edge(a,b). edge(b,c). edge(b,d).
edge(c,a). edge(c,e). edge(d,e).
```

Let us write module `search` with the only public predicate `path/3` which is able to find simple pathes (i.e. pathes without loops) on a graph defined by predicate `edge/2`. (Predicate `path/3` has already been defined at the end of (1.10.3), and tested in (1.11).)

```
:- module( search, [ path/3 ] ).


:- use_module( graph, [edge/2] ).
:- use_module( library(lists), [reverse/2] ).
```

```
% path(A,Z,Path) :-
%     Path is an acyclic list of nodes from A to Z.
. . .
```

In order to define the file containing the graph dynamically, we have to change only the first two sentences of the module:

```
:- module( search, [ new_graph/1, path/3 ] ).
new_graph(File) :- use_module( File, [edge/2] ).
. . .
```

We suppose that the latter version of module `search` is defined in the module file called `search_path.pl`. Now we can initialize this module with any module file defining the graph through the public predicate `edge/2`:

```
| ?- use_module(search_path).
% ...module search imported into user...
yes
| ?- new_graph('../book/graph').
% ...module graph_of_edges imported into search...
yes
| ?- findall(Path,path(_,e,Path),Pathes).
Pathes = [ [e],[a,b,c,e],[a,b,d,e],
         [b,c,e],[b,d,e],[c,a,b,d,e],[c,e],[d,e] ]
```

1.13.2 Module prefixing

The strict module system outlined above implies some basic problems.

First, it prevents us from testing our program without altering it, because we cannot directly invoke the local predicates of a module. This problem is simply solved: any goal can be prefixed with a module name in the form *module:goal* overriding the source module of *goal* to *module* (`op(550, xfy, :)`). We can similarly override the source module of a predicate specification, rule head or sentence [C+12].

For example, predicate `search : path_2/4` in the previous subsection (1.13.1) can be invoked from any module using the prefix "`search :`":

```
| ?- search:path_2(d,[b,a],E,Path).
E = d,   Path = [a,b,d] ? ;
E = e,   Path = [a,b,d,e] ? ;
no
```

If there are more module prefixes of some query, rule head or sentence, just the rightmost one has effect. For example, goal `k : m : n : p(X)` is equivalent to goal `n : p(X)`.

If you consult a module file (load it in interpreted mode: see (1.10.1)), even the local predicates of the module loaded can be listed using module prefix. For example:

```
| ?- listing(search:path_2/4).
search:path_2(A, Ancestors, A, Path) :-
       lists:reverse([A|Ancestors], Path).
search:path_2(A, Ancestors, Z, Path) :-
       graph_of_edges:edge(A, B),
       B\=A,
       \+member(B,Ancestors),
       search:path_2(B, [A|Ancestors], Z, Path).
```

This means that this module system cannot really hide a predicate, but it can prevent conflicts of predicates with the same name and arity in different modules.

Note that module prefixing used without sensible control may destroy the module structure of our program.

1.13.3 Modules and meta-predicates

The problem shown through the next example is related to the interactions of *meta-predicates* and modules. Predicate `forall(P,Q)` (below) enumerates the solutions of goal `P` and for each solution it tries to find the first solution of goal `Q`: if `Q` fails, `forall(P,Q)` also fails immediately. It does not collect the solutions. It is used to check whether `P` implies `Q`, or it is invoked to force side-effect `Q` for each solution of `P`:

```
% forall(P,Q) :- for each solution of P, Q can be solved,
%                 that is P implies Q.
forall(P,Q) :- \+ ( P, \+Q ).

 ?- forall(path(b,_,Path),(write(Path),write(' '))).
[b] [b,c] [b,c,a] [b,c,e] [b,d] [b,d,e]
```

The `forall/2` form is a meta-predicate and its arguments are meta-arguments parameterized with meta-goals. If that `forall/2` is a public predicate of module `m` and we invoke `forall(path(b,_,Path),(write(Path),write(' ')))` in module `h`. Probably we want to use predicate `path/3` visible in module `h`. But the body of `forall/2` is invoked in module `m`. Therefore, `path(b,_,Path)` is invoked also in module `m`. But `path/3` may not be visible in module `m`, or even worse, `m:path/3` may differ form `h:path/3`. (It follows that in the first case goal `path(b,_,Path)` raises `existence_error`, in the second case it silently invokes another predicate.)

One possible solution is to prefix the meta-parameters of a meta-predicate invocation with the name of the calling module (notice that in general each meta-

parameter needs module prefix, but in the next example the second parameter refers only to built-in predicates and so this qualification may be omitted):

```
?- forall(h:path(b,_,Path),(write(Path),write(' '))).
```

Nevertheless, this manual qualification is inconvenient, and error-prone: one may forget the qualification, and it may lead even to silent errors; or one may rename the calling module, but forget to rename the module prefix with it. Therefore, it is better if this qualification of the meta-parameters with the calling module is automatic. This automatic qualification is called *module name expansion.*

Therefore, we can declare *module name expansion* for the meta-arguments of the public predicates of the modules. In such a *meta-predicate declaration* a meta-argument must be indicated with a *colon* (or an integer) [C$^+$12]; a non-meta-argument can be indicated with any ground term except the colon and the integers. Then the meta-parameters of an invocation to the meta-predicate are automatically prefixed with the calling module, while the other actual parameters are not expanded:

```
:- module( m, [ forall/2, collect/3 ] ).

:- meta_predicate forall(:,:), collect(?,:,?).

forall(P,Q) :- \+ ( P, \+Q ).

collect(Solution,Goal,Results) :-
   findall(Solution,Goal,ListOfSols), sort(ListOfSols,Results).
```

The Prolog compiler expands the predicate invocation `forall(C,D)` into `forall(h:C,h:D)` provided that `h` is its calling module. Similarly, the predicate invocation `collect(X,G,Xs)` is expanded into `collect(X,h:G,Xs)` if `h` is its calling module.

There is the same case when we call predicates loading program files like `compile(F)` or `consult(F)`, e.g. from module `h`. They must also know that program code `F` must be imported into module `h`. Similarly, if module `m` is defined in file `lpp.pl`, and `use_module(lpp,[forall/2]` is invoked in module `user`, this invocation must also know that predicate `forall/2` must be imported into module `user`. And there are the predicates dynamically modifying the program like `asserta/1`, `assertz/1`, `retract/1`, etc. They must know, into which module to assert, or from which module to delete the rules.

Actually, the built-in predicates of SICStus Prolog are defined in the predefined module `prolog`, and they are its public predicates. They are automatically imported into each module. The predicates loading program files, those dynamically accessing (and possibly modifying) the loaded program, and those working with Prolog goals (like `findall(Sol,Goal,Solutions)`) are defined as meta-predicates specifying the arguments waiting for Prolog goals, rules, file names (or blackboard keys: see [C$^+$12]) as meta-arguments, that is, arguments to which

module name expansion is due. Then these invocations are informed about the module they have to operate on.

1.14 Conclusion

Clearly, it is over the scope of this short introduction to discuss the whole field of logic progamming or even to discuss the full language of Prolog and/or the details of its programming methodology. Nevertheless, we hope we can give you a hand if you want to start a detailed study.

1.14.1 Some classical literature

There are easy-to-read introductions to the logical basis in [FGN90] and [Nil82]. There is a more general introduction to the topic in [Fla94] and [Kow79]. The theoretical aspects of logic programming are detailed in [Llo87].

Originally published in 1981, the first textbook on programming in Prolog was that of Clocksin and Mellish. It became popular because of its comprehesive, tutorial approach, and general programming examples. An updated, extended version is [CM03]. The book of Sterling and Saphiro is considered one of the best handbooks on the programming methodology of Prolog [SS94] which is suggested for students at beginner, and intermediate level. For advanced students we suggest [O'K90], and still some chapters of [SS94].

There is a good description of the Prolog standard in the book of Deransart, Ed-Dbali, and Cervoni ([DEDC96], [ISO95] and [ISO00]). The programs of this chapter were tested in SICStus Prolog 4.2.3. The User's Manual, and more documentation is available at [C$^+$12]. Finally, the *Prolog 1000 database* includes more than 500 Prolog application entries, and it is available on the Internet.

1.14.2 Extensions of Prolog

Logic programming is not seperated from other branches of programming. There have been several attempts to unify LP with functional programming, which is the other main declarative programming paradigm. Some of them led to well functioning systems, like implementations of *ALF*, *λProlog*, *Curry*, *Mercury*, etc. ([DL86], [Han94] and [C$^+$13]). However, the practical use of the *functional logic prgramming* (FLP) languages is extremely rare, maybe because of the lack of a (de facto) standard, and the lack of the programmers properly trained.

There are interesting attempts to extend Prolog with object-oriented features ([Mos94] and [C$^+$12]). These extensions are useful in building simulator programs, and expert systems.

Phil Vasey's *flex* is a frame based extension of *LPA*, and *Quintus* Prolog. Frames are similar to classes of OOP, but they are specialised for building expert

systems. There are important business applications, intelligent query, registering, diagnostics, and scheduling systems.

Prolog implementations typically know the *definite clause grammar* (DCG) formalism as a built-in extension. The DCG formalism is strong enough to define any language described with a Chomsky grammar, Turing automaton, or attribute grammar. Their application makes it much easier to develop parsers, compilers, text generators of formal and natural languages, and intelligent interfaces processing the texts of such languages, although these application areas well fit Prolog in general. When we load a DCG into Prolog, its rules are transformed one by one into Prolog rules using an algorithm of simple syntactic transformations. In this way, the DCG notation is just a de facto standard syntactic sugar in Prolog. Because of its comprehesiveness, it is very popular, and in the Prolog folklore it is used even in application areas seemingly far from text processing, for example in list handling utilities.

1.14.3 Problems with Prolog

Prolog, and library modules of *C++, Java, etc. implementations* containing parts of Prolog are often used in practice.

It is an important problem that only the standard of the core of Prolog is widely accepted ([DEDC96], [ISO95] and [ISO00]). But there is no *de facto* standard of modules, no standard of C/C++, Java, Oracle interfaces, no standard of GUI, etc.; although these are part of any serious Prolog implementation. Clearly, the developers of software packages do not want to depend on a particular Prolog implementation, because any time it may disappear from the market. Therefore, only half a dozen Prolog implementators are present in the business world and they have been there for more than twenty years now, although their number is much higher.

In many Prolog implementations, program tracing and debugging cannot go through the foreign language interface. Therefore, C++, Java, etc. programmers often switch from Prolog to a local package of their own language. (However, logic programming studies are needed to use effectively also these packages.) On the other hand, these local packages rarely have the effectivity and robustnes of a professional Prolog implementation.

There is a myth that the Prolog programs are slow. Astonishingly, it is based partly on an unsuccessful USA project in the *sixties of the last century* which was trying to build an effective LP language [SS94]. The key of the successful European projects in the seventies was adopting *pattern matching instead of unification* in goal reduction, especially in parameter passing.[18] On the other hand, a good Prolog programmer needs special programming experiences, and studies. The reputation of the language does not benefit from the fact that it is misused by Prolog hackers.

[18] See the occurs check problem in (1.6.2,1.6.3).

1.14.4 Fifth generation computers and their programs

Maybe Japan's Fifth Generation Computer Systems project (FGCS) of the eighties was the most famous initiative applying some kind of "Prolog" as its main language. It aimed to develop highly parallel computers with many CPUs instead of increasing the complexity of a single CPU. Some prototype computers were created. They can be programmed with a *committed choice concurrent constraint logic programming* language: its predicates consist of guarded Horn clauses where each rule contains a guard which is similar to a Prolog cut. When a predicate has been invoked, the rules matching the invocation start to work in parallel, and when one of them arrives at the guard, it cuts the others. Therefore, there are no search trees like in LP in general, and there is no possibility to find the different solutions satisfying a query while traversing its search tree with backtracking or other strategy.

So it has turned out that the *committed choice* feature conflicts with the fundamental strength of *logic programming* compared to *functional programming*: the applications needing intelligent search cannot take advantage of search trees. And the prototype machines of the FGCS project were soon outperformed by general purpose computers. (The same had happened before to the Lisp Machine and to the Thinking Machine.) But in spite of these failures, for example "multi-core architectures at the low-end and massively parallel processing at the high end" seem to be winner ideas of the FGCS Project now.[19]

1.14.5 Newer trends

Mercury is a promising FLP language (1.14.2) developed at the University Of Melbourne under the supervision of Zoltán Somogyi. The core of the language is similar to pure Prolog; and a query is evaluated through backtracking traversal of its search tree. But it is purely declarative with strict, static type and mode system (in a predicate invocation one parameter can be only purely input or purely output), which allows the highest level of compile-time checks and code optimization while prevents the use of some effective and elegant programming technics, that is the organisation of data with partial lists and trees ([SS94] and [O'K90]), and therefore the step-by-step, top-down approximation of the output (1.5.2). Probably these later features imply that it could not become the successor of Prolog, although its strong, polymorphic type system, and its module system allow the development of high quality, standard libraries.

The development of effective, parallel Prolog systems retaining backtracking search and in general, the support of conventional Prolog programs so that the goals and/or the branches of the program run in parallel without forcing the programmer to organize parallelism, this is a strong trend of the search in the field of LP. There are two basic approaches. If some branches of the search tree are evaluated in parallel, the system is *or-parallel* (for example, the *Aurora Prolog*

[19] See http://en.wikipedia.org/wiki/Fifth_generation_computer.

by Warren; Gigalips project). If some goals of a goal sequence are reduced in parallel, this Prolog is *and-parallel* (for example, the *Muse* version of SICStus Prolog). Warren's *Andorra-I* unifies both approaches. However, the best sequential Prolog implementations still often overperform these implementations.

Constraint Logic Programming (CLP) originates from the end of the eighties. Today this is one of the most successful areas of LP ([Hen86], [MS98], [SB04] and [C$^+$12]). It is typically used to solve optimization problems. The program code is similar to that of standard Prolog but it is more declarative than that and the search is more intelligent: performing special computations it takes advantage of the problem solving methods of other scientific areas. Beside Prolog goals we have special goals called *constraints* behaving like demons. The variables of these constraints have predefined domains (for example, `X :: [1..5]` means that the domain of `X` is the integer range `1..5`). The run of the program means the backtracking traversal of the search tree, like in Prolog, but there is also a *constraint store*. When the consraints are encountered during the run of the program they are put into the constraint store. When any parameter of a constraint in this store is modified (for example, it is substituted by a value or its domain narrows), the consistency of the constraint store is checked, and if it is found inconsistent, the search backtracks. For example, the consistency check may find that a modification of a variable of a constraint implies the modifications of other variables, and if the domain of some variable contains just one element, it is substituted. Any modification implies further check. If some domain becomes empty, the search backtracks.

For example, SICStus Prolog contains three different constraint solvers implemented in libary modules [C$^+$12]. There are also constraint programming packages built on OO languages, for example the *ILOG Solver* of IBM implemented as a C++ library.

We mention that other newer trends are Abductive, Answer Set, and Inductive Logic Programming.

1.15 Summary

Today – as a result of successes and failures – the place of LP seems to be in the "thinking" components of multiagent systems. Using LP we can effectively develop intelligent data base interfaces, packages needing intelligent search (even on the Internet), text processors, translator programs, natural language interfaces, expert systems, programs of logic puzzles, programs with symbolic computations, optimizations, etc.

The approaches of the problems are different with LP, and different with more traditional programming paradigms. Therefore, LP helps you to find new, elegant solutions.

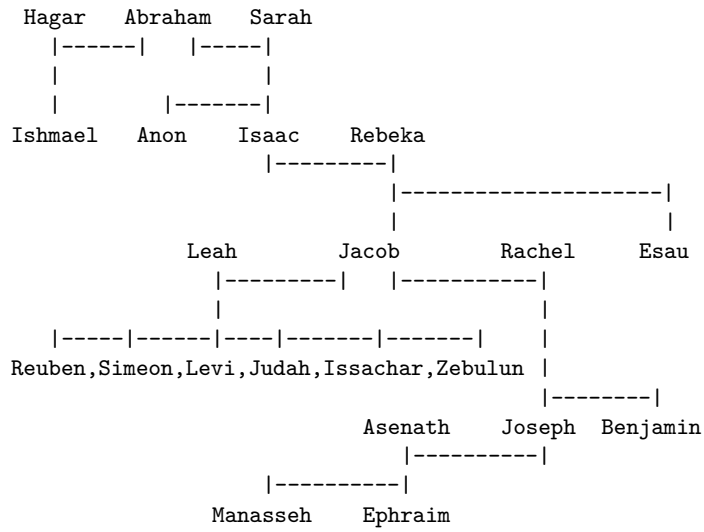


Figure 1.4: family/3 in Exercise 1.2

1.16 Exercises

Exercise 1.1. *Apply the algorithm of goal reduction (1.6.1) to this query tested in (1.5.2):* `append_(Xs, Ys, [1, 2, 3])`. *Draw the corresponding search tree.*

Exercise 1.2. *Let us suppose that we have facts of the form "*`family(Father, Mother, Children)`*.". For example:*

```
family('Abraham','Sarah',['Isaac','Anon']).
family('Abraham','Hagar',['Ishmael']).
family('Isaac','Rebeka',['Jacob','Esau']).
family('Jacob','Rachel',['Joseph','Benjamin']).
family('Jacob','Leah',['Reuben','Simeon','Levi','Judah',
                       'Issachar','Zebulun']).
family('Joseph','Asenath',['Manasseh','Ephraim']).
```

Define predicates `brother_or_sister/2`, `aunt_or_uncle/2`, *and* `parent_in_law/2` *based on* `family/3` *with the usual meaning. (In order to help their test, Figure 1.4 illustrates predicate* `family/3`*.)*

Exercise 1.3. *Write list handling predicates to receive the prefixes, suffixes, continuous sublists, and possibly discontinuous subsequences of a proper list. Write predicate* `divide(Xs, Odds, Evens)` *which takes proper list* `Xs`*; puts the first, third, fifth, etc. items into* `Odds`*, and the second, fourth, sixth, etc. items into* `Evens`*, thus generating two subsequences of* `Xs` *of approximately equal length. Write predicate* `sorted_union(Xs, Ys, Us)` *producing the sorted union of* `Xs`*, and* `Ys`

in `Us` *where* `Xs`*, and* `Ys` *are strictly increasing proper lists according to the standard order; and* `Us` *will have the same properties. Each predicate must be able to check the appropriate property, and it must be able to generate the lists satisfying that property. Take advantage of the programming methods (step-by-step approximation of the output, accumulator pairs, etc.) while coding a predicate. Verify the finiteness of the search tree of each corresponding query.*

Exercise 1.4. *Consider the representation suggested for binary trees introduced at the very end of Section 1.4 , page 22.Using this representation (*`o` *is the empty tree, and* `t`(leftSubTree, root, rightSubTree) *is the scheme of a nonempty tree), define predicate* `inorder(Tree,List)` *where* `List` *contains the inorder traversal of* `Tree`*.*

Exercise 1.5. *Use the representation suggested above for binary search trees (sorted according to the standard order of the Prolog terms). Write the usual operations of such search trees (creating an empty tree, inserting a given piece of data, checking for it, deleting it: the tree must not contain duplications).*

Be careful: *do not mix the notion of binary search tree with the notion of the search tree of a Prolog query.*

Exercise 1.6. *Implement the well-known algorithms of sorting lists like Insertsort and Mergesort in Prolog. Be careful that the computational complexity of the Prolog programs should not be higher than that of the corresponding algorithm. (For example, the computational complexity of Insertsort is* $O(n^2)$*, and that of Mergesort is* $\Theta(n * log(n))$ *where* n *is the length of the proper list to be sorted.)*

Note: *The computational complexity (i.e. operational or time complexity) of a Prolog program is measured in LI (number of Logical Inferences) which means the number of predicate invocations performed during the run of the program.*

Exercise 1.7. *Try to implement your own version of* `ground/1`*:* `ground(Term)` *is true,* iff `Term` *is a ground term (see Section 1.4).*

Exercise 1.8. *Try to implement your own version of predicate* `findall/3`*.*
It can be called `find_all/3`*. Put it into a module.*

Exercise 1.9. *Find the description of the standard Prolog predicate* `setof/3`
*([*DEDC96*], [*ISO95*], [*C$^+$12*], [*SS94*] and [*SB04*]).*
Compare it to predicate `collect/3` *in (1.11).*

Exercise 1.10. *Implement the standard operations of queues in a module.*
`empty(Q)`*:* `Q` *is empty queue (create and check);*
`add(Q,X,QX)`*: add* `X` *to the end of* `Q`;
`rem(XQ,X,Q)`*: remove the first item of queue* `XQ`*.*

Exercise 1.11. *In Subsection (1.13.1) we defined predicate* `path/3`*, which is able to traverse a graph in depth-first (backtracking) manner. Write another predicate*

which implements breadth-first-search on a graph computing a shortest path between a given start and goal node. Use the queue handling predicates defined in Exercise 1.10. (A shortest path is a path consisting of the least number of nodes here.)

Exercise 1.12. *Implement the well-known algorithm of Quicksort on Prolog lists: Quicksort selects an arbitrary item X of a nonempty list, and separates the remainder of the list into two lists. The first list contains the items smaller than X. The second list contains the items greater than X. (The items equal to X can go into either of them.) Then the two lists are sorted recursively. The resulting list is generated from the two sorted lists with X as a middle element: these are concatenated together.*

1.17 Useful tips

Tip 1.1. *Consider now a search tree of query* `append_(Xs,Ys,[1,2,3])` *referring to predicate* `append_/3`*:*

```
append_([],Ys,Ys).                                % a1
append_([X|Xs],Ys,[X|Zs]) :- append_(Xs,Ys,Zs).   % a2
```

In the solution of Exercise 1 we use the following notations. The two rules of `append_/3` *are called* **a1**, *and* **a2**. *If the depth of a goal in the search tree is* **i**, *then it is denoted with* **gi**. *If there is a goal* **gi**, *a goal reduction is applied to it, and rule* **aj** *is involved in it, then the goal reduction is denoted with* **a(i,j)**. *If rule* **aj** *is involved in a goal reduction* **a(i,j)**, *then we suppose that the LP system renames each variable* **V** *of rule* **aj** *to* **Vi**. *The* $\mathbf{k}^{\text{th}}$ *solution is called* **sk**. *Only the output substitutions (see Section 1.2.4) are shown.*

Tip 1.2. *You can choose a child using* `member/2`.

Tip 1.3. *Take care of the base cases and of the recursive cases of the predicates. For each predicate, give a sufficient condition of the finiteness of the search tree ("PreCond"). The finiteness of the search tree of each corresponding query can be based on two facts:*

1. *The input is proper list;*
2. *And its length is decreasing through the recursion.*

Tip 1.4. *A straitforward solution contains two rules: the inorder traversal of the empty tree is empty list, while that of a nonempty tree is that of the left subtree, the root, and that of the right subtree concatenated into a single list. (Try to solve it with single* `append/3` *call in the rule body.) A more refined version avoids the* `append/3` *calls, and it has linear time complexity: we generalize the problem to append the inorder traversal of a tree before a list. (Consider the note of Exercise 1.6.)*

Tip 1.5. *The time complexity of the solutions should be proportional to the height of the search tree. (Consider the note of Exercise 1.6.) This effectivity can be achieved by distinguishing the cases of empty and nonempty trees: the recursive cases go down into the appropriate subtree.*

Tip 1.6. *Generalize Insertsort to* `sorted_inserts(Xs,As,Ys)` *inserting the elements of* `Xs` *into the sorted list* `As`*. Try to give tail recursive solutions. In Mergesort the base cases are the empty list, and the lists of a single item. Longer lists are divided into two lists of approximately equal lengths. (Use* `divide/3` *from Exercise 1.3 to divide a list.) The two lists are sorted with Mergesort, and then results are merged in a sorted way. The code of this sorted merge is similar to the code of* `sorted_union/3` *from Exercise 1.3, except that the duplicates are preserved.*

Note: *If we use* `sorted_union/3` *in Mergesort instead of sorted merge, we receive* `unionsort/2` *which is equivalent to* `sort/2` *in Section 1.11.*

Tip 1.7. *Recursively, a ground term is an atomic or a compound with ground arguments.* **Note:** *Although predicate* `ground/1` *is not part of the Prolog standard, many implementations contain it with the obvious meaning.*

Tip 1.8. *As a first try, one may assert the first parameter of* `find_all/3` *for each solution of the second argument (the parameterizing goal) into a dynamic predicate local to the module. You may use* `assertz/1` *for this purpose. Then (using* `retract/1`*) you can retract the asserted facts while collecting the parameters into a list applying step-by-step approximation of the output (1.5.2).*

It is a useful initialization of `find_all/3` *to delete each possible clauses of the dynamic predicate used for collecting the solutions. In this way one gets rid of the garbage left by any previous run propagating an exception.*

Another problem is risen if one wants to embed a call to `find_all/3` *into another invocation to it. Let us see a simple example of the appropriate behaviour (the example is a bit artificial, because* `Xs` = `Zs`*, but hopefully easy to understand):*

```
| ?- find_all( Zs+Ys,
          ( append(Xs,Ys,[1,2,3]),
            find_all(Z,member(Z,Xs),Zs) ),
            As ).
As = [[]+[1,2,3],[1]+[2,3],[1,2]+[3],[1,2,3]+[]]
```

Unfortunately, if the method above were programmed in a straightforward way, the inner invocation of `find_all/3` *could modify the partial results of the outer one, because they use the same dynamic predicate to collect their solutions. In order to solve this problem we have to identify levels of the embedded* `find_all/3` *invocations, and each invocation works only at its level. In addition, if a goal parameterizing a* `find_all/3` *call propagates an exception, still we have to restore the outer level.*

Tip 1.9. *They have similar parameters to those of* `findall/3`*, and both of them collects (the) solutions of the parameterizing goal into a strictly increasing list omitting multiple solutions. But consider the role of variable quantification in* `setof/3`*. You may also try to use alternatively* `collect/3`*, and* `setof/3` *instead of* `findall/3` *in Exercise 1.11. Be careful, because in some cases* `setof/3` *may fail (but it is easy to overcome this difficulty). You may also try to measure the the effeciency of the different solutions. Unfortunately, the way of these measurements is implementation dependendt in Prolog.*

Tip 1.10. *Using the trivial* proper list *representation, the maximal and average computational complexity of the operations* `add(Q,X,QX)`*, and* `rem(XQ,X,Q)` *are* $O(n)$ *where* n *is the length of the queue. (We suppose that the frequency of these operations is the same.) For example, one may use the trivial proper list representation where a nonempty queue has the form* $[\mathtt{X1}, \mathtt{X2}, ..., \mathtt{XN}]$ *and* $[]$ *represents the empty one. Then the computational complexity of* `add` *is linear, and that of* `rem` *is constant, therefore the average computational complexity of* `add` *and* `rem` *is also linear. (If a nonempty queue had the form* $[\mathtt{XN}, ..., \mathtt{X2}, \mathtt{X1}]$*, the computational complexity of* `add` *would be constant, but that of* `rem` *would be linear.)*

If we use a clever double-stack *representation, the average computational complexity of these operations become* $\Theta(1)$*.*

In this representation $\mathtt{d}([\mathtt{X1}, ..., \mathtt{XM}], [\mathtt{YN}, ..., \mathtt{Y1}])$ *represents the abstract queue* $< \mathtt{X1}, ..., \mathtt{XM}, \mathtt{Y1}, ..., \mathtt{YN} >$*;* `add` *pushes a new item into the second stack (as the new first item of the list representing the second stack) with constant computational complexity;* `rem` *removes the topmost item of the first stack (that is the first item of the list representing it) also with constant computational complexity, provided that the first stack is nonempty. But if that first stack (i.e. list) is empty, we have to check the second one. If it is empty, the operation fails. If it is nonempty, we can* reverse *this second list and move it into the first one. Then we can remove its first item as before. The problem is that the computational complexity of the* reverse *operation is linear, and therefore sometimes the computational complexity of the* `rem` *operation is also linear. Besides this, the average computational complexity of these operations is still* $\Theta(1)$*, because considering*

```
rem(  d( [], [Y|Ys] ),  FirstItem,  ResultQueue  )
```

$[\mathtt{Y}|\mathtt{Ys}]$ *is as long as the number of* `add` *calls before the actual* reverse *call inside* `rem` *(but not before any previous* reverse *call inside* `rem`*). Therefore, the cost of the actual* reverse *call can be scattered among those* `add` *calls when we calculate the average computational complexity of* `add` *and* `rem`*, and so it is* $\Theta(1)$*.*

If we use a pair of partial lists *where the second component is the logical variable at the end of the first partial list, we are able to succeed in reducing the maximal computational complexity of these operations to* $\Theta(1)$*. Using this*

representation an empty queue has this form: `Z – Z`*, and a nonempty queue has this form:* `[P1,P2,...,PN|Z] – Z` *(*`var(Z)` *must be true in both cases). If one checks the solution, one finds that the implementation is somewhat tricky and considering a queue which is the input of an* `add` *operation, its second component is better to be a* `var`*. But after the* `add` *call this property of the input queue does not stand. Therefore, it does not clearly fit any* `add` *operations any more. A* `rem` *operation does not change the second component of the queue. Therefore, if a* `rem` *is performed on queue Q resulting in queue Qr, and then an* `add` *is performed on queue Q resulting in queue Qa, then even the second component of Qr is a* `nonvar`*, and it should not be used for subsequent queue operations. However, for practical purposes it is usually enough that in a linear sequence of queue operations the output queue of any operation can be used as the input queue of the next operation.*

Tip 1.11. *A breadth-first-search (BFS) of a graph begins with a chosen start node.*[20] *Then it explores the children of the start node, then the nodes available through at least two arcs (edges), then thoose available through at least three arcs, and so on. BFS may have goal node(s). If yes, it stops when a goal node is achieved. Otherwise it goes on until each node accessible from the start node is explored. (Therefore, it terminates* iff *the number of nodes accessible from the start node is finite or there is a goal node accessible from the start node.) In BFS the parent of a node is defined as the node from which it was found first. During the search for each node visited BFS usually records a reference to its parent, in order to record a shortest path from the start node to the actual node through these backward references (and the parent reference of the start node is somewhat extremal). The key operation of BFS is* expanding *a node which means collecting its children. Also we need the set of visited nodes, and a queue containing the nodes already visited but still not expanded. The algorithm initialises the queue with the start node. In each step it removes the first element of the queue, expands it, and puts its unvisited children at the end of the queue. If it is relevant, the goal property of a node is checked immediately before it is put into the queue. Instead of booking the parent references of the visited nodes, in Prolog the queue of BFS may contain nonempty lists: the first element of such a list is the node to be expanded, the second element is its parent, the third one is its grandparent, and so on, the last element of the list is always the start node. Therefore, such a list with a given first node always contains a shortest path to this node.*

Tip 1.12. *The solution is straightforward. However, one may get rid of the costs of appending the parts of the result at each level of recursion, and this is not trivial. One may use the so called* d-lists*, i.e. pairs of (partial or proper) lists where the second list is a suffix of the first one.*

[20] We suppose that the graph is locally finite, i.e. there is no node with infinitely many children.

1.18 Solutions

Solution 1.1. *The search tree of* `append_(Xs,Ys,[1,2,3])`

```
%% append_(Xs,Ys,[1,2,3]): apply the algorithm of
%% goal reduction to this query.
%% Draw the corresponding search tree.
%% (explanation of notations at Section "Clues")

?- append_(Xs,Ys,[1,2,3]).      % g0
     { Xs <- [], Ys <- [1,2,3] }      % a(0,1)
Xs = [], Ys = [1,2,3]                          % s1
     { Xs <- [1|Xs0] }                % a(0,2)
     append_(Xs0,Ys,[2,3]).     % g1
       { Xs0 <- [], Ys <- [2,3] }     % a(1,1)
Xs = [1|Xs0] = [1|[]] = [1], Ys = [2,3]            % s2
       { Xs0 <- [2|Xs1] }             % a(1,2)
       append_(Xs1,Ys,[3]).     % g2
         { Xs1 <- [], Ys <- [3] }     % a(2,1)
Xs = [1|Xs0] = [1,2|Xs1] = [1,2], Ys = [3]         % s3
         { Xs1 <- [3|Xs2] }           % a(2,2)
         append_(Xs2,Ys,[]).    % g3
           { Xs2 <- [], Ys <- [] }    % a(3,1)
Xs = [1,2|Xs1] = [1,2,3|Xs2] = [1,2,3], Ys = []    % s4
           { match([],[X3|Zs3]) fails }    % a(3,2)
```

Solution 1.2. *Family relationships*

```
family('Abraham','Sarah',['Isaac','Anon']).
family('Abraham','Hagar',['Ishmael']).
family('Isaac','Rebeka',['Jacob','Esau']).
family('Jacob','Rachel',['Joseph','Benjamin']).
family('Jacob','Leah',['Reuben','Simeon','Levi','Judah',
                       'Issachar','Zebulun']).
family('Joseph','Asenath',['Manasseh','Ephraim']).



brother_or_sister(X,Y) :-
    family(_,_,Xs),
    member(X,Xs), member(Y,Xs), X \== Y.

aunt_or_uncle(X,Y) :-
    brother_or_sister(X,Z),
    ( family(Z,_,Cs) ; family(_,Z,Cs) ),
    member(Y,Cs).

parent_in_law(X,Y) :-
    ( family(X,_,Cs) ; family(_,X,Cs) ),
    member(C,Cs),
    ( family(C,Y,_) ; family(Y,C,_) ).
```

Solution 1.3. *Basic operations on lists*

```
%% PreCond: Xs or Ys is proper list.
%% prefix(Xs,Ys) :- Ys is a prefix of Xs.
prefix(_Xs,[]).
prefix([X|Xs],[X|Ys]) :- prefix(Xs,Ys).

%% PreCond: Xs is proper list.
%% suffix(Xs,Ys) :- Ys is a suffix of Xs.
suffix(Xs,Xs).
suffix([_X|Xs],Ys) :- suffix(Xs,Ys).
```

```
%% PreCond: Xs is proper list.
%% sublist_x(Xs,Ys) :- Ys is a continuous sublist of Xs.

%% a: Suffix of a prefix
sublist_a(Xs,Ys) :- prefix(Xs,Ps), suffix(Ps,Ys).

%% b: Prefix of a suffix
sublist_b(Xs,Ys) :- suffix(Xs,Ss), prefix(Ss,Ys).

%% c: Recursive definition of a sublist
sublist_c(Xs,Ys) :- prefix(Xs,Ys).
sublist_c([_X|Xs],Ys) :- sublist_c(Xs,Ys).

%% PreCond: Xs is proper list.
%% subseq(Xs,Ys) :-
%%     Ys is a possibly discontinuous subsequence of Xs.
subseq([],[]).
subseq([X|Xs],[X|Ys]) :- subseq(Xs,Ys).
subseq([_X|Xs],Ys) :- subseq(Xs,Ys).

%% PreCond: Xs is proper list.
%% divide(Xs,Odds,Evens) :- in their original order, the
%%     first, third, fifth, etc. items of Xs are in Odds, the
%%     second, fourth, sixth, etc. items of Xs are in Evens.
divide([],[],[]).
divide([X|Xs],[X|Ys],Zs) :- divide(Xs,Zs,Ys).

%% PreCond: Xs and Ys are proper lists, sorted strictly
%%          increasingly according to the standard order.
%% sorted_union(Xs,Ys,Us) :-
%%     Us contains the sorted union of Xs and Ys.
sorted_union([],Ys,Ys).
sorted_union([X|Xs],[],Us) :- !, Us = [X|Xs].
sorted_union([X|Xs],[Y|Ys],Us) :-
    ( X @< Y -> Us = [X|Zs], sorted_union(Xs,[Y|Ys],Zs)
    ; X @> Y -> Us = [Y|Zs], sorted_union([X|Xs],Ys,Zs)
    ; Us = [X|Zs], sorted_union(Xs,Ys,Zs)
    ).
```

Solution 1.4. *Inorder traversals of a binary tree*

```
%% PreCond: Tree is a binary tree represented as follows.
%%     o - empty tree
%%     t( LeftSubTree, Root, RightSubTree ) - nonempty tree
%% inorder(Tree,Is) :- the inorder traversal of the data
%%                     in Tree results proper list Is.
inorder(o,[]).
inorder(t(Lt,X,Rt),Xs) :-
    inorder(Lt,Ls), inorder(Rt,Rs), append(Ls,[X|Rs],Xs).

%% The optimized version of predicate inorder/2.
%% Without appends, linear time complexity.
inorder_opt(T,Is) :- inorder_app(T,[],Is).

%% inorder_app(Tree,List,Is) :- the inorder traversal
%%           of Tree appended before List results Is.
inorder_app(o,Xs,Xs).
inorder_app(t(Lt,X,Rt),Xs,Is) :-
    inorder_app(Rt,Xs,Ys), inorder_app(Lt,[X|Ys],Is).
```

Solution 1.5. *Basic operations on binary search trees*

```
%% The usual operations of proper binary search trees
%% (creating an empty tree, inserting a given piece of data,
```

```
%% checking for it, deleting it:
%% the search trees must not contain duplications).

%% empty_tree(T) :- T is an empty tree.
empty_tree(o).

%% PreCond: T is proper binary search tree.
%% tree_ins(T,X,TX) :- TX proper binary search tree is
%%     received by the sorted insert of X into T.
tree_ins(o,X,t(o,X,o)).
tree_ins(t(Lt,Root,Rt),X,TX) :-
    ( X @< Root -> TX = t(LXt,Root,Rt), tree_ins(Lt,X,LXt)
    ; X @> Root -> TX = t(Lt,Root,RXt), tree_ins(Rt,X,RXt)
    ; TX = t(Lt,Root,Rt)
    ).

%% PreCond: T is proper binary search tree.
%% tree_has(T,X) :- T contains item X.
tree_has(t(Lt,Root,Rt),X) :-
    ( X @< Root -> tree_has(Lt,X)
    ; X @> Root -> tree_has(Rt,X)
    ; true
    ).
%% Queries like tree_has(o,X) automatically fail.

%% PreCond: TX is proper binary search tree.
%% tree_del(TX,X,T) :- T proper binary search tree is
%%     received by the sorted delete of X from TX.
tree_del(t(Lt,Root,Rt),X,T) :-
    ( X @< Root -> T = t(Ldt,Root,Rt), tree_del(Lt,X,Ldt)
    ; X @> Root -> T = t(Lt,Root,Rdt), tree_del(Rt,X,Rdt)
    ; Rt == o -> T = Lt
    ; Lt == o -> T = Rt
    ; T = t(Lt,Min,Rmt), Rt = t(L,Y,R),
      out_min(L,Y,R,Min,Rmt)
    ).

%% PreCond: t(L,X,R) is a proper binary search tree.
%% out_min(L,X,R,Min,Tm) :-
%%     the leftmost element of the proper binary tree
%%     t(L,X,R) is Min, the remaining tree is Tm.
out_min(o,Min,Rt,Min,Rt).
out_min(t(L,X,R),Y,Rt,Min,t(Lm,Y,Rt)) :-
    out_min(L,X,R,Min,Lm).
```

Solution 1.6. *Sorting lists*

```
%% PreCond: Xs is a proper list.
%% insertsort(Xs,Ys) :- proper list Ys contains
%%     the items of Xs increasingly sorted according to the
%%     standard order of Prolog terms.
insertsort(Xs,Ys) :- sorted_inserts(Xs,[],Ys).

%% PreCond: Xs is a proper list.
%% sorted_inserts(Xs,As,Ys) :- sorted insert of
%%     the items of proper list Xs into the sorted
%%     proper list As results sorted proper list Ys.
sorted_inserts([],Ys,Ys).
sorted_inserts([X|Xs],As,Ys) :-
    insert_sorted(As,X,Bs), sorted_inserts(Xs,Bs,Ys).

%% PreCond: As is an increasingly sorted proper list.
%% insert_sorted(As,X,Bs) :- sorted insert of X
%%               into As results the increasingly
%%               sorted proper list Ys.
```

```
insert_sorted([],X,[X]).
insert_sorted([Y|Ys],X,Zs) :-
    ( X @=< Y -> Zs = [X,Y|Ys]
    ; Zs = [Y|Us], insert_sorted(Ys,X,Us)
    ).

%% PreCond: Xs is a proper list.
%% mergesort(Xs,Ys) :- proper list Ys contains
%%     the items of Xs increasingly sorted
%%     according to the standard order of Prolog terms.
mergesort([],[]).
mergesort([X|Xs],Ys) :- mergesort(Xs,X,Ys).

%% PreCond: Xs is a proper list.
%% mergesort(Xs,X,Ys) :- proper list Ys contains
%%     the items of [X|Xs] increasingly sorted
%%     according to the standard order of Prolog terms.
mergesort([],X,[X]).
mergesort([Y|Ys],X,Zs) :-
    divide(Ys,As,Bs),
    mergesort(As,X,Es), mergesort(Bs,Y,Fs),
    sorted_merge(Es,Fs,Zs).

%% PreCond: Xs,Ys: increasingly sorted proper lists.
%% sorted_merge(Xs,Ys,Zs) :- proper list Zs is
%%     the result of the sorted merge of Xs and Ys.
sorted_merge([],Ys,Ys).
sorted_merge([X|Xs],[],Ms) :- !, Ms = [X|Xs].
sorted_merge([X|Xs],[Y|Ys],Ms) :-
    ( X @< Y -> Ms = [X|Zs], sorted_merge(Xs,[Y|Ys],Zs)
    ; X @> Y -> Ms = [Y|Zs], sorted_merge([X|Xs],Ys,Zs)
    ; Ms = [X,Y|Zs], sorted_merge(Xs,Ys,Zs)
    ).

%% Note: if we use sorted_union/3 instead of sorted_merge/3,
%% we receive a strictly increasing list, duplicates removed:

unionsort([],[]).
unionsort([X|Xs],Ys) :- unionsort(Xs,X,Ys).

unionsort([],X,[X]).
unionsort([Y|Ys],X,Zs) :-
    divide(Ys,As,Bs),       % See Exercise 3.
    unionsort(As,X,Es), unionsort(Bs,Y,Fs),
    sorted_union(Es,Fs,Zs). % See Exercise 3.
```

Solution 1.7. *Checking, whether a term is ground or not*

```
%% ground_(Term) :- Term is a ground.
ground_(Term) :-
    ( atomic(Term) -> true
    ; compound(Term),
      functor(Term,_F,N), ground_args(N,Term)
    ).

ground_args(N,Term) :- N>1,
    arg(N,Term,A), ground_(A),
    N1 is N-1, ground_args(N1,Term).
ground_args(1,Term) :-
    arg(1,Term,A), ground_(A).
```

Solution 1.8. *Implementing the predefined predicate* `findall/3`

```
%% find_all(X,Goal,Xs) :- findall(X,Goal,Xs).
```

```
:- module( findall, [ find_all/3 ] ).
:- meta_predicate find_all(?,:,?).

:- dynamic((solution/2, counter/1)).

set_counter(C) :-
    retractall(counter(_)),
    asserta(counter(C)).

:- set_counter(1).

find_all(X,Goal,Xs) :-
    counter(I),
    %% to handle embedded calls to find_all/3:
    I1 is I+1, set_counter(I1),
    catch(
          (
            %% clear things, if an old goal crashed:
            retractall(solution(I,_)),
            %% assert solutions at the Ith level:
            Goal, assertz(solution(I,X)), fail
          ;
            collect_solutions(I,Xs),
            set_counter(I)
          ),
          Error,
          ( set_counter(I),
            throw(Error)
          )
         ).

collect_solutions(I,Ys) :-
    ( retract(solution(I,X)) ->
          Ys = [X|Xs], collect_solutions(I,Xs)
    ; Ys = []
    ).
```

Solution 1.9. *Collecting solutions without duplications*

.

%% Compare the standard Prolog predicate `setof/3`
%% to our predicate `collect/3`.

Predicate `setof/3` *is less effective than* `collect/3`, *but it is more complex and it has more expressive power: it may produce more sets of solutions through backtracking depending on the variable quantifications of the goal parameterizing* `setof/3`. *However, this extra expressive power is useless in many practical applications. Also,* `collect/3` *never fails, and produces exactly one set of solutions, but* `setof/3` *fails iff the parameterizing goal has no (more) solutions.*

Solution 1.10. *Queue handling and Prolog modules*

```
%% Implement the standard operations of queues in a module.
%% empty(Q): Q is empty queue (create and check).
%% add(Q,X,QX): add X to the end of Q.
%% rem(XQ,X,Q): remove the first item of queue XQ.

:- module( queue3,
   [ empty0/1, add0/3, rem0/3,
     empty/1, add/3, rem/3,
     emptyq/1, addq/3, remq/3 ] ).
```

```
%% proper list representation:
%% A queue has the form: [X1,X2,...,XN]

empty0([]).

%% add0/3 has linear computational complexity:

add0([],X,[X]).
add0([Y|Ys],X,[Y|Zs]) :- add0(Ys,X,Zs).

rem0([X|Xs],X,Xs).

%% double-stack representation:
%% d([X1,...,XM],[YN,...,Y1]) represents
%% the abstract queue <X1,...,XM,Y1,...,YN>

empty(d([],[])).

add(d(Xs,Ys),E,d(Xs,[E|Ys])).

rem(d(Xs,Ys),E,ResultQueue) :-
    ( Xs = [Z|Zs] -> E = Z, ResultQueue = d(Zs,Ys)
    ; % Xs == [], Ys \== [],
      reverse(Ys,[E|Us]), ResultQueue = d(Us,[])
    ).

:- use_module( library(lists), [reverse/2] ).

%% Queue represented with a pair of partial lists
%% where the second component is the logical variable
%% at the end of the first partial list:
%% a nonempty queue has this form: [P1,P2,...,PN|Z]-Z
%% and an empty queue has this form: Z-Z
%% where var(Z) must be true.

emptyq(Q-Q) :- var(Q).

addq(Q1-[ITEM|Y],ITEM,Q1-Y).

%% A call: addq([P1,P2,...,PN|Z]-Z,ITEM,R).
%% After the call: Q1 = [P1,P2,...,PN|Z],
%%                 Z = [ITEM|Y], R = Q1-Y.
%% Therefore: R = [P1,P2,...,PN,ITEM|Y]-Y.

remq([H|T]-Z,H,T-Z) :-
     [H|T]\==Z.  % the input queue was nonempty

%% Using this last representation
%% each single operation needs a constant number of LI
%% (Logical Inferences = predicate invocations).
```

Solution 1.11. *Shortest path*

```
%% Graph-search with breadth-first-search strategy.

%% breadth_first_search(Start,Goal,SolPath) :-
%%     SolPath is a proper list of nodes representing a path
%%     of the minimal length (i.e.optimal) from Start to Goal.

breadth_first_search(Start,Goal,SolPath) :-
    ground(Start), ground(Goal),
    ( Start == Goal -> SolPath = [Start]
    ; empty(InitQueue), add(InitQueue,[Start],Queue),
      bfs(Queue,[Start],Goal,SolPath)
    ).
```

```
:- use_module( solution10_lp, [ empty/1, add/3, rem/3 ] ).

%% Queue is a queue of lists of the form [Node|Ancestors]
%% where Node\=Goal, and to Node we have found an optimal
%% path but Node has not been expanded. Ancestors consists
%% of the ancestors of Node on this optimal path starting
%% with its parent. Visited contains the visited nodes.

bfs(Queue,Visited,Goal,SolPath) :-
    rem(Queue,[Node|Ancestors],RemainderQueue),
    children(Node,Children,Visited),
    ( has(Children,Goal) ->
          reverse([Goal,Node|Ancestors],SolPath)
    ; process_children(Children,[Node|Ancestors],
                       RemainderQueue,ResultQueue),
      append(Children,Visited,NewVisited),
      bfs(ResultQueue,NewVisited,Goal,SolPath)
    ).

:- use_module( library(lists), [ reverse/2 ] ).

%% expansion: Children is the list of
%%     the unvisited children of Node.

children(Node,Children,Visited) :-
    findall(Child,child(Node,Child,Visited),Children).

%% Child is an unvisited child of Node.

child(Node,Child,Visited) :-
    arc(Node,Child), \+ has(Visited,Child).

has([X|Xs],Y) :-
    ( X == Y -> true
    ; has(Xs,Y)
    ).

%% Add the children with their ancestors to RemainderQueue.

process_children( [FirstChild|Children], Ancestors,
                  RemainderQueue, ResultQueue ) :-
    add(RemainderQueue,[FirstChild|Ancestors],TempQueue),
    process_children(Children,Ancestors,TempQueue,
                                        ResultQueue).
process_children([],_Ancestors,ResultQueue,ResultQueue).


/*  Test data   */

%% Acyclic component
    arc(a,b).   arc(a,c).   arc(a,d).  arc(a,e).
    arc(b,f).   arc(b,i).
    arc(c,f).   arc(c,g).
    arc(d,j).
    arc(e,k).
    arc(f,h).   arc(f,i).
    arc(g,h).
    arc(j,g).
    arc(k,j).

%% Cyclic component
    arc(x,y).   arc(y,z).   arc(z,x).
    arc(y,u).   arc(z,t).
```

```
/*
     a------>b-------->i
    //|\       \      /
   | / | \       \    /                    __________
   | /  \        \  /                    /          \
   | V   V        V/                    V            \
   | d   c------>f---->h                 x----->y----->z
   |  \   \          /                          |      |
   |   \   \        /                           |      |
   |    \   \      /                            V      V
   |     \   \    /                             u      t
   |      \   \  /
   V       V   V/
   e-->k-->j-->g
*/

%% | ?- breadth_first_search(a,g,Sol).
%% Sol = [a,c,g]
%% | ?- breadth_first_search(a,i,Sol).
%% Sol = [a,b,i]
%% | ?- breadth_first_search(x,t,Sol).
%% Sol = [x,y,z,t]
```

Solution 1.12. *Quicksort and d-lists*

```
%% PreCond: Xs is a proper list.
%% quicksort(Xs,Ys) :- proper list Ys contains
%%     the items of Xs increasingly sorted according to the
%%     standard order of Prolog terms.
%% Note:
%% Our quicksort selects the first item X of a nonempty list,
%% and separates the remainder of the list into two lists.
%% The first list contains the items smaller than X.
%% The second list contains the items greater than X.
%% (The items equal to X can go into either of them.)
%% Then the two lists are sorted recursively.
%% The resulting list is generated from the two sorted lists
%% with X as a middle element: these are concatenated together.
quicksort0([],[]).
quicksort0([X|Xs],Ys) :-
    separate(Xs,X,Smallers,Greaters),
    quicksort0(Smallers,Littles),
    quicksort0(Greaters,Bigs),
    append(Littles,[X|Bigs],Ys).
separate([],_X,[],[]).
separate([Y|Ys],X,Ss,Gs) :-
    ( Y @< X -> Ss = [Y|Ls], separate(Ys,X,Ls,Gs)
    ; Gs = [Y|Bs], separate(Ys,X,Ss,Bs)
    ).
%% One may use d-lists to get rid of the costs of append/3:
quicksort(Xs,Ys) :- quicksort(Xs,Ys,[]).
%% quicksort(Xs,Ys,Zs) :-
%%     d-list Ys-Zs represents the sorted Xs
%%     where list Zs is a suffix of list Ys.
%% Note: d-list [Y1,...,Yn|Zs]-Zs represents
%%       sequence <Y1,...,Yn>, and
%%       d-list Zs-Zs represents sequence <>.
quicksort([],Zs,Zs).
quicksort([X|Xs],Ys,Zs) :-
    separate(Xs,X,Smallers,Greaters),
    quicksort(Smallers,Ys,[X|Bigs]),
    quicksort(Greaters,Bigs,Zs).
```

Final remark: Remember to TEST, what happens, if you force BACKTRACKING into your Prolog program?

Bibliography

[C+12] M. Carlsson et al. SICStus Prolog 4.2.3 User's Manual. Technical report, Swedish Institute of Computer Science, 2012.
`http://www.sics.se/isl/sicstuswww/site/documentation.html`

[C+13] M. Carlsson et al. Functional logic programming. Technical report, Wikipedia, 2013.
`http://en.wikipedia.org/wiki/Functional_logic_programming`

[CM03] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer, 2003.

[Col82] A. Colmerauer. Prolog and Infinite Trees. In K. L. Clark and S. A. Tarnlund (eds.): *Logic Programming*, pages 231–251. Academic Press, London, 1982.

[DEDC96] P. Deransart, A. A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard (Reference Manual).* Springer-Verlag, Berlin, 1996.

[DL86] D. DeGroot and G. Lindstrom. *Logic Programming: Functions, Relations and Equations.* Prentice Hall, Englewood Cliffs, NY, 1986.

[DM93] P. Deransart and J. Maluszyńsky. *A Grammatical View of Logic Programming.* MIT Press, Cambridge, Massachusetts, 1993.

[FGN90] I. Fekete, T. Gregorics, and S. Nagy. *Bevezetés a mesterséges intelligenciába.* Műszaki Könyvkiadó, Budapest, 1990.

[Fla94] P. Flach. *Simply Logical. Intelligent Reasoning by Example.* John Wiley and Sons, 1994.

[Fó83] Á. Fóthi. Bevezetés a programozáshoz. Egyetemi jegyzet, ELTE TTK, Budapest, 1983.

[Han94] M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *The Journal of Logic Programming*, 1994.

[Hen86] P. V. Henterick. *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, Massachusetts, 1986.

[Hor94] E. Horowitz. *Fundamentals of Programming Languages*. Computer Science Press, Rockville, Maryland, 2nd edition, 1994.

[ISO00] International Organization for Standardization. *Information technology - Programming languages - Prolog - Part 2: Modules*, 2000.
ISO/IEC 13211-2

[ISO95] International Organization for Standardization. *Information technology - Programming languages - Prolog - Part 1: General core*, 1995.
ISO/IEC 13211-1

[Kow79] B. Kowalski. *Logic for Problem Solving*. North-Holland, New York, Amsterdam, Oxford, 1979.

[LG96] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*. MIT Press, Cambridge, Massachusetts, 1996.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 2nd edition, 1987.

[Mey00] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, New York, 2nd edition, 2000.

[Mos94] C. Moss. *Prolog++, The Power of Object-Oriented and Logic Programming*. Addison-Wesley, Reading, Mass., 5th edition, 1994.

[MS98] K. Marriot and P. J. Stuckey. *Programming with Constraints*. MIT Press, Cambridge, Massachusetts, 1998.

[NG+98] J. Nyéky-Gaizler (ed.) et al. *Az Ada95 programozási nyelv*. ELTE Eötvös Kiadó, Budapest, 1998.

[Nil82] N. J. Nilsson. *Principles of Artificial Intelligence*. Springer-Verlag, Berlin, 1982.

[O'K90] R. A. O'Keefe. *The Craft of Prolog*. MIT Press, Cambridge, Massachusetts, 1990.

[Rob65] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *ACM*, 12:23–41, January 1965.

[SB04] P. Szeredi and T. Benkő. Deklaratív programozás: Bevezetés a logikai programozásba. Technical report, Budapesti Műszaki Egyetem, Budapest, 2004.
`http://dp.iit.bme.hu/documents.html`

[Seb13] R. W. Sebesta. *Concepts of Programming Languages*. Pearson, Boston, 10th edition, 2013.

[SS94] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, Cambridge, Massachusetts, 2nd edition, 1994.

[War80] D. H. D. Warren. Logic programming and compiler writing. *Software Practice and Experience*, 10(2), 1980. Article first published online: 27 OCT 2006.

Index