

23 Intelligent Image Processing Using Prolog

Bruce G. Batchelor

Cardiff University, Cardiff, Wales, UK

23.1	<i>Part 1: Basic Features of Prolog</i>	1032
23.1.1	Adding Intelligence	1032
23.1.2	Introducing Prolog	1033
23.1.2.1	Sample Program	1036
23.1.2.2	Queries	1038
23.1.2.3	Back-tracking	1041
23.1.2.4	Recursion	1043
23.1.3	Prolog+	1043
23.1.4	Sample Programs	1047
23.1.4.1	Recognising Bakewell Tarts	1047
23.1.4.2	Recognising Printed Letters	1049
23.1.4.3	Identifying Table Cutlery	1050
23.1.4.4	Analysing All Visible Objects	1053
23.1.4.5	Recognising a Table Place Setting	1053
23.1.5	Justifying the Design	1055
23.1.5.1	Additional Spatial Relationships (🔗 Fig. 23.8)	1056
23.1.5.2	High-Level Concepts	1058
23.1.5.3	Why Base Prolog+ on QT?	1060
23.1.5.4	Using Prolog+	1061
23.2	<i>Part 2: Using Prolog to Understand Natural Language</i>	1062
23.2.1	Prolog and Natural Language	1062
23.2.1.1	Definite Clause Grammars	1062
23.2.1.2	Example: Moving Chess Pieces	1064
23.2.1.3	Parsing	1064
23.2.1.4	Another Example: Controlling a Set of Lamps	1065
23.2.1.5	Extracting Meaning	1067
23.2.1.6	Remarks	1070
23.3	<i>Part 3: Implementation of Prolog+</i>	1071
23.3.1	Implementation of PQT	1071
23.3.1.1	Structure of a PQT System	1071
23.3.1.2	Processing Prolog+ Goals	1073
23.3.1.3	Prolog Programs Implementing Prolog+	1074

Abstract: Some potential applications of Machine Vision, particularly those involving highly variable objects (🔗 Chap. 2), require greater reasoning power than standard image processing languages can provide. This chapter is based on the premise that an intelligent vision system can be made, by the simple expedient of embedding image manipulation and measurement routines, such as those provided by QT (🔗 Chap. 21), within the Artificial Intelligence language Prolog. The great strength of Prolog is its ability to perform symbolic reasoning. It is a declarative language. That is, it operates by searching automatically for a solution to a problem, given a description of (part of) the world. The user specifies the desired characteristics that a solution must have. Prolog's search engine then hunts for the specific conditions that must exist to obtain a solution. If the user chooses, or programs Prolog appropriately, it will search for multiple solutions. Unlike a conventional imperative language, Prolog does not have any instructions. Instead, it accepts, a series of statements that are then tested individually for their veracity. It does not possess any of the usual flow-control tools, such as IF ... THEN, FOR, WHILE, CASE, etc. Instead, it relies on back-tracking and recursion. The former allows temporary assumptions that have been made while searching for a solution to be revised before continuing the search. A software interface interconnecting QT with a modern implementation of Prolog (SWI-Prolog) was designed. The result, called PQT, allows image processing operations to be invoked procedurally within a Prolog program, or executed interactively, as in QT. Image measurement values can be used to define temporary values for (i.e. instantiate) Prolog variables. Using PQT, it is possible to define general spatial relationships (left, above, concentric, parallel, etc.), geometric features (T, X, U, V), shape characteristics (round, curved, elongated), colour and texture (yellow, smooth, spotty). These would be much more difficult to define and use in a standard programming language. PQT can also be used effectively to control external peripheral devices but not in real time. PQT inherits Prolog's limited ability to analyse and extract meaning from natural language sentences, such as spoken commands. While PQT was designed to perform vision-based tasks, such as inspecting food products, it is also capable of the high-level reasoning needed to incorporate them into a well-balanced meal.

This chapter is in three parts. In the first, we explain how certain tasks involving abstract symbolic reasoning about images, can be accomplished, using a Prolog program with embedded image processing commands. In the second, we discuss the processing of natural language using Prolog. This allows us to build a vision system that is able to understand spoken commands expressed in English. In addition, such a machine can perform such tasks as operating an array of lights, or controlling a simple robot. In the third part, we describe how Prolog and QT, the image processing software package discussed in 🔗 Chap. 21, can be interfaced to one another. This combination implements a convenient, easy-to-use language that has the ability to process and analyse images and then reason logically about them.

23.1 Part 1: Basic Features of Prolog

23.1.1 Adding Intelligence

Many potential applications of machine vision require greater reasoning power than the image processing languages described earlier can provide. This chapter is founded on the premise that an intelligent vision system can be made, by the simple expedient of embedding image

manipulation and measurement routines within the AI language Prolog. The top-level control language for such a system will be called *Prolog+*. One of the most recent implementations of Prolog+ was constructed by interfacing QT (➤ Chap. 21) to SWI-Prolog [1] and will therefore be called *PQT*. We shall see later that Prolog+ is able to solve some of the problems that do not yield to standard image processing techniques (➤ Chap. 14) operating alone. (QT should not be confused with Qt which is a cross-platform application and user-interface framework, produced by Nokia. URL <http://qt.nokia.com/products/>.)

Prolog was devised specifically for developing AI programs and is used extensively for tasks involving Natural Language processing [2], machine learning [3] and for building rule-based and expert systems [4]. The range of applications is impressive in its breadth and includes areas as diverse as financial risk assessment, bridge design, planning health care, designing computer systems, route planning, symbolic mathematical manipulation [3, 5].

Prolog is an unusual computer language and bears no real resemblance to any of the “conventional” languages of Computer Science. In the core language, there are no FOR loops, IF ... THEN statements, CASE statements, or any of the usual flow-control constructions. There is no variable assignment process; Prolog does not even allow instructions. The reader is, therefore, warned against trying to reconcile the distinctive features of Prolog with his/her prior programming experience. *Prolog is different!* In the following section, we describe this fascinating language for readers with no prior knowledge; we shall try to explain *why* Prolog is important. However, to obtain a more comprehensive understanding, the uninitiated reader is urged to refer to the standard textbooks. [4, 6, 7] Later, we shall attempt to justify the choice of Prolog, by describing and listing PQT programs that solve some interesting vision problems. For the moment, let it suffice to say that many major benefits have been obtained through the use of Prolog that could not have been obtained nearly so easily using other languages.

From his own experience, which now stretches over more than 20 years, the author can report that no serious shortcomings of Prolog+ have come to light. PQT combines SWI-Prolog with the proven set of image processing utilities embodied in QT. In Prolog+, it is possible to define abstract relationships/attributes such as *left_of*, *above*, *bigger*, *between*, *beside*, *near*, *yellow*, *curved*, *spotty*, etc. Representing these concepts in more popular languages, such as Java, C, C++, etc. is much more difficult.

Experienced Prolog programmers will appreciate that Prolog+ is a super-set of standard Prolog, with numerous additional built-in predicates that perform image processing functions. These behave much like the standard *write/1* predicate; by trying to satisfy a goal, the desired effect (e.g., negating, thresholding or filtering an image) is actually achieved. Predicates that calculate numeric values (e.g., Euler number, blob count, etc.) instantiate variables in the normal way. Readers who are familiar with Prolog may wish to omit ➤ Sect. 23.2 and skip directly to ➤ Sect. 23.3.

23.1.2 Introducing Prolog

It took the author 2 years to understand *why* Prolog is important and just 2 weeks to become reasonably proficient at using it! The reason why Prolog is worth further study is simply that it allows a novel and very natural mode of programming to be used. It permits a person to state the nature of an acceptable solution to a given search-based task, rather than *how*

to find one. To understand this, consider the task of finding a marriage partner for a given hypothetical man. (*Not the author!*) It is relatively easy to specify the basic “requirements” to a dating agency, in terms such as those listed below. (The following list and Prolog program are intended to illustrate a point about Prolog; they do not constitute any statement about the absolute desirability of any type of religion, or personal/racial characteristics.)

Sex	Female
Age	[45,55]
Height	[150,180]
Weight	[45,75]
Language	English
Desirable characteristics	List might cover personality, faith, interests, hobbies, life-style, etc.

Clearly, this list cannot guarantee success in romance but it is sufficiently detailed to allow us to illustrate the general principles of what is known as *Declarative Programming*. Conventional programming languages are, by contrast, termed *imperative*, since they consist of a sequence of instructions. Prolog “programs” consist of a set of logical tests, rather than commands.

Writing a Prolog program to find a wife/husband is straightforward and has actually been used by at least one commercial dating agency. Here is a program to find a suitable wife, using the very small number of criteria specified above:

```
suitable_wife(X):-
    person(X),                % Find a person called X
    sex(X,female),            % Is person X female?
    age(X,A),                 % Find age, A, of person X
    A >= 45,                  % 45 or more years old?
    A <= 55,                  % 55 or fewer years old?
    height(X, H),             % Find height, H, of person X
    H >= 150,                 % Is X at least 150 cm tall?
    H <= 180,                 % Is X at most 180 cm tall?
    weight(X,W),              % Find the weight of person X
    W >= 45,                  % Weight >= 45 kg?
    W <= 75,                  % Weight <= 75 kg?
    speaks(X,English).        % Does X speak English?
    must_be(X,[christian,kind, truthful,generous,loving,loyal]).    % Obvious meaning
```

Given such a program and a set of stored data about a collection of people, Prolog will search the database, to find a suitable match. There is no need to tell Prolog how to perform the search. The reader is urged to consider rewriting the program using Basic, C, Fortran, Java, or Pascal. An imperative-language program will take much longer to write, will consist of many more lines of code and be less transparent, because it imposes an unnatural mode of thought on the problem. Declarative (Prolog) programming is very much more natural in its style and allows a person to think directly about the type of solution required, rather than

how to find it. Prolog differs from most other computer languages in several very important ways:

- (a) Firstly, a Prolog “program” does *not* consist of a sequence of instructions, as routines written in conventional languages do. (The correct term is “application”, since a program is, strictly speaking, a sequence of instructions. However, we shall continue to use the term “program”, since this is more familiar to most readers.) Instead, it provides a medium for describing (part of) the world. As we have already stated, Prolog is referred to as a *declarative* language, while most other computer languages, military orders, knitting patterns, automobile repair manuals, musical scores and culinary recipes are all examples of *imperative* languages. This is a vital difference, which distinguishes Prolog and a small group of related languages from the better known conventional languages of Computer Science.
- (b) The “flow of control” in a Prolog program does not follow the normal convention of running from top to bottom. In a Prolog program, the flow is very likely to be in the reverse direction, through a mechanism called *back-tracking*.
- (c) Through the use of back-tracking, it is possible to make and subsequently revise temporary assignments of values to variables. This process is called *instantiation* and is akin to re-evaluating assumptions made earlier. (The word “*instantiation*” is derived from the same linguistic root as “*instance*”. Prolog tries to find an *instance* of some variable(s) which cause the given predicate to be true.) Instantiation is performed, in order to try and prove some postulate, theorem or statement, which may or may not be true. As far as Prolog is concerned, theorem proving (also called *goal satisfaction*) is the equivalent process to running or executing an imperative language program.
- (d) It is possible to make very general statements in Prolog in a way that is not possible in most other languages. We shall see more of this feature later, but for the moment, let us illustrate the point with a simple example. In Prolog it is possible to define a relationship, called *right* in terms of another relationship, called *left*.

In English:	“A is to the <i>right</i> of B if B is to the <i>left</i> of A.”
In Prolog:	<i>right(A,B) :- left(B,A).</i>

(Read “:-” as “*can be proved to be true if*” or more succinctly as “*if*”). Notice that neither A nor B have yet been defined. In other words, we do not need to know what A and B are in order to define the relationship *right*. For example, A and B might be features of an image such as blob centres, or corners. Alternatively, A and B may be objects on a table, people (*Hitler* and *Lenin*) or policies (*National Socialism* and *Marxism*). The point to note is that Prolog+ allows the relationships *left* and *right* to be applied to any such objects, with equal ease. (Is *hitler* to the right of *lenin*? In the political sense, “yes”, while the answer is “no”, when we consider at the layout of words on this page.)

- (e) Prolog makes very extensive use of *recursion*. While, Java, C and many other imperative languages also allow recursion, in Prolog it forms an essential control mechanism.

Prolog is not normally recommended for writing programs requiring a large amount of numerical manipulation. Nor is Prolog appropriate for real-time control, or other computational processes requiring frequent processing of interrupts.

23.1.2.1 Sample Program

The following program deals with the ancestry and ages of members of two fictitious families. (Not the author's!) Remember that “:-” means “*is true if*”, while comma (“,”) means *AND*. Single-line comments are prefixed with %, while multi-line comments begin with /* and are terminated thus */.

```

/*
The following facts specify in which years certain people were born. Interpretation:
    born(roger,1943)
means that
    “roger was born in 1943”.
*/
born(roger,1943).
born(susan,1942).
born(pamela,1969).
born(ghraham,1972).
born(thomas,1953).
born(angela,1954).
born(elizabeth,1985).
born(john,1986).
born(marion,1912).
born(patricia,1911).
born(gertrude,1870).
born(david,1868).
/*
These facts describe the parent-child relationships that exist in the families. Interpretation:
parent(X,Y) means that
    “X is a parent of Y”.
*/
parent(roger,pamela).
parent(roger,ghraham).
parent(patricia,roger).
parent(anne,patricia).
parent(david,patricia).
parent(marion,susan).
parent(susan,ghraham).
parent(susan,pamela).
parent(thomas,john).
parent(angela,john).
parent(thomas,elizabeth).
parent(angela,elizabeth).
/*
Defining a relationship called “child”. Read this as follows:
    “A is a child of B if
        B is a parent of A.”
*/

```

child(A,B) :- parent(B,A).

*/**

Defining a relationship called "older". Read this as follows:

*"A is older than B if
the age of A is X AND
the age of B is Y AND
 $X > Y$ ".*

**/*

older(A, B) :-

*age(A,X), % Age of A is X
age(B,Y), % Age of B is Y
 $X > Y$. % Is X greater than Y?*

*/**

Defining a relationship "age". Read this as follows:

*"A has age B if
A was born in year X AND
it is now year Y AND
 $X \leq Y$ AND
B is equal to $Y - X$."*

**/*

age(A,B) :-

*born(A,X),
date(Y,_),
 $X \leq Y$,
B is $Y - X$.*

*/**

The definition of "ancestor" has two clauses. Prolog always tries to satisfy the top one first. If this fails, it then tries to satisfy the second clause.

Interpretation: ancestor(A,B) means that

"A is an ancestor of B."

The first clause should be interpreted as follows:

"A is an ancestor of B if A is a parent of B".

**/*

ancestor(A,B) :- parent(A,B).

*/**

The second clause should be interpreted as follows:

*"A is an ancestor of B if
A is a parent of Z AND
Z is an ancestor of B."*

Notice the use of recursion here.

**/*

ancestor(A,B) :-

*parent(A,Z),
ancestor(Z,B).*

*/**

Definition of "print_descendents".

*This uses backtracking to find all possible solutions.
The first clause always fails but in doing so it prints the descendants and their dates of birth.*

```
print_descendents(A) :-
    nl,                                % New line
    write('The known descendants of '),
                                % Print a message
    write(A),                        % Print value of A
    write(' are:'),                  % Print a message
    ancestor(A,Z),                  % A is ancestor of Z
    born(Z,Y),                      % Z was born in year Y
    nl,                              % New line
    tab(10),                        % 10 white spaces
    write(Z),                       % Print value of Z
    write(', born '),                % Print a message
    write(Y),                       % Print value of Y
    fail.                           % Force back-tracking
% Second clause always succeeds and prints a new line
print_descendents(_) :- nl.
```

23.1.2.2 Queries

A Prolog program consists of a set of *facts* and *rules* for interpreting them. The user then presents a *query*, which Prolog tries to prove to be true. The following goal was satisfied, because the program contains facts that explicitly state it to be true.

Query: *born(susan, 1942)*
Result: YES

In the following query, X is a variable. The query tests the proposition that susan was born some time (X). The goal succeeds, even if we refuse to accept that X = 1942.

Query: *born(susan, X)*
Result: *X = 1942*
 YES

It is possible to use a variable for the first argument, to find who was born in 1942.

Query: *born(X, 1942)*
Result: *X = susan*
 YES

It is also possible to use two or even more variables to specify very general queries:

Query: *born(X, Y)*

(X and Y are both variables, since their names have capital initial letters.) There are several possible solutions to this query.

Result: $X = \text{roger}$
 $Y = 1943$

$X = \text{susan}$
 $Y = 1942$

$X = \text{pamela}$
 $Y = 1969$

$X = \text{graham}$
 $Y = 1972$

$X = \text{thomas}$
 $Y = 1953$

$X = \text{angela}$
 $Y = 1954$

$X = \text{elizabeth}$
 $Y = 1985$

$X = \text{john}$
 $Y = 1986$

$X = \text{marion}$
 $Y = 1912$

$X = \text{patricia}$
 $Y = 1911$

$X = \text{gertude}$
 $Y = 1870$

$X = \text{david}$
 $Y = 1868$

NO MORE SOLUTIONS

(Notice the alternative solutions generated by this general query.)

The following query is satisfied by using the simple rule for *age/2* given above.

Query: $\text{age}(\text{marion}, Z)$
Result: $Z = 77$
 YES

However, the following query uses a higher-level rule; *older/2* is defined in terms of *age/2* which is itself defined using *born/1*. (It is customary to specify the *arity* (number of arguments) of a Prolog predicate in the following way: *predicate_name/arity*.)

Query: *older(marion, susan)*
 Result: YES

It is not always possible to satisfy a given goal.

Query: *older(susan, marion)*
 Result: NO
 (NO should properly be expressed as *NOT PROVEN*.)

Another query which can only be solved by using a rule (*child*), which relies on another explicit relationship (*parent*):

Query: *child(susan, Z)*
 Result: *Z = marion*
 YES

The following is an example of a query that is satisfied by using a recursive rule (i.e., *ancestor*, clause 2) This is tantamount to asking who is a descendent of susan.

Query: *ancestor(susan, Z)*
 Result: *Z = graham*
Z = pamela
 NO MORE SOLUTIONS

(Notice the alternative solutions.)

The following query finds all of the ancestors of a given object (We know that *graham* is likely to be a person, but Prolog does not, as we have not explicitly told it so.)

Query: *ancestor(Z, graham)*
 Result: *Z = roger*
Z = susan
Z = patricia
Z = anne
Z = david
Z = marion
 NO MORE SOLUTIONS

Finally, here are two queries involving a high-level predicate:

Query: *print_descendents(marion)*
 Result: *The known descendants of marion are:*
susan, born 1942
graham, born 1972
pamela, born 1969
 YES

Query: *print_descendents(anne)*
 Result: *The known descendants of anne are:*
patricia, born 1911

roger, born 1943
 pamela, born 1969
 graham, born 1972
 YES

23.1.2.3 Back-tracking

Consider the following program, which prints all of the grandparent-grandchild pairs that can be inferred from a *parent/2* database like that given above.

```

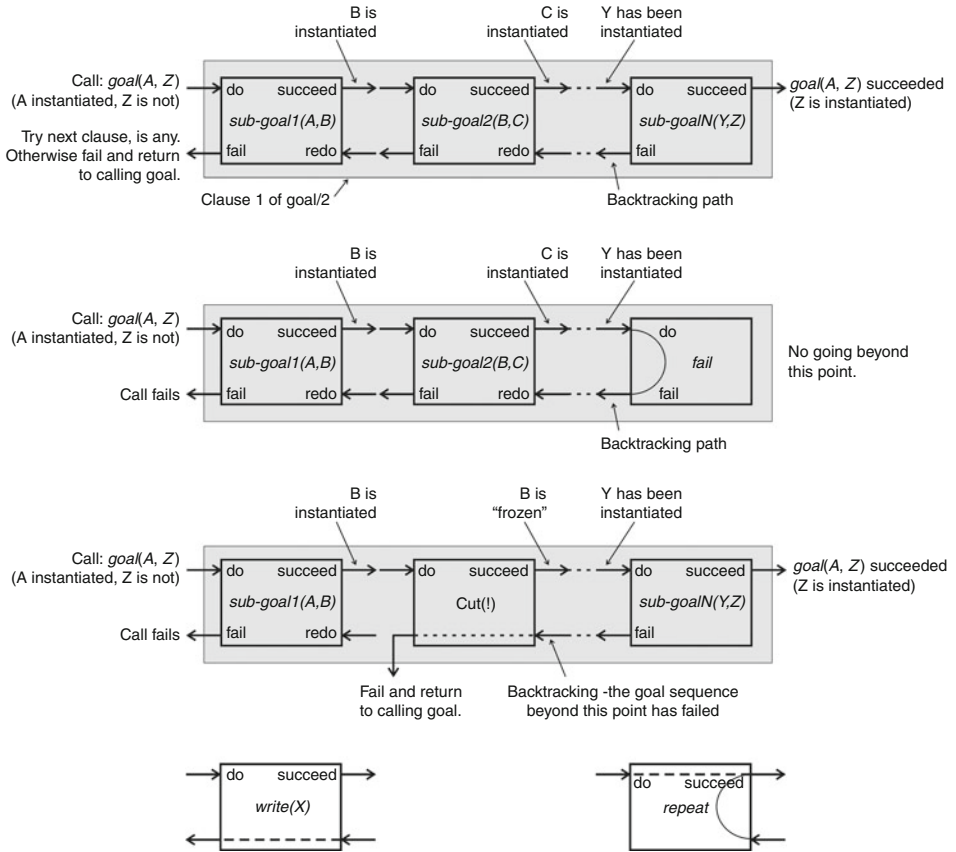
1. go :-                               % Starting the definition of the first clause
2.   parent(A,B),                     % parent/2 is defined above. Find one parent-child pair.
3.   parent(B,C),                     % Find child of B (i.e., grand-child of A)
4.   write([A,C]),                    % Writes one of the solutions i.e., a [grandparent,grandchild] pair.
5.   nl,                              % New line
6.   fail.                            % First clause ends. It fails after all solutions have been found

7. go :-                               % Second clause. Is tried when clause 1 fails
8.   write('The end '),               % Writes a simple message
9.   nl.                              % New line. This clause and the goal "go" always succeed here

```

The goal *go* always succeeds. Clause 1 is evaluated first. In line 2, a parent-child pair (A,B) is found. This value of the variable B is used in line 3, as Prolog seeks a child (C) of B. (C is, of course a grandchild of A.) In line 4, the pair [A,C] is printed, while line 5 simply prints a new line. Line 6 always fails. In Prolog, goal failure is not catastrophic; it simply forces Prolog to begin back-tracking, as it continues its search for other possible solutions. Line 5 is then re-evaluated, but the built-in predicate *nl/0* is not resatisfied. Back-tracking therefore continues to line 4. The built-in predicate *write/1* is not resatisfied either, so back-tracking continues to line 3. If B has more than one child, the variable C is re-instantiated in line 3. In this case, lines 4, 5 and 6 are evaluated next. This cycle is repeated until all of the children of B have been found. Once this happens (i.e., line 3 has exhausted all solutions for a given value of B), back-tracking returns to line 2. Here, a new pair [A,B] is found and the whole process described so far is repeated. (A and/or B may change as a result of this re-instantiation.) Eventually, line 2 runs out of possible solutions. When this happens, clause 2 fails and Prolog begins evaluating clause 2, which in this case always succeeds.

When writing Prolog programs, it is not normally necessary to pay much attention to the possibility of back-tracking. However, an experienced Prolog programmer is often able to structure the program so that its searching is performed efficiently. One way to do this is to “freeze” all instantiations, once one partial solution has been found. This is accomplished using a device known as the *cut*, which is represented in Prolog by the operator “!”. ➤ [Figure 23.1](#) explains the flow of control during backtracking and cut (1) using the so-called Box Model. This is a representation of Prolog execution, in which we think in terms of *procedures* rather than *predicates*. Each sub-goal is represented by a box with four ports:



■ Fig. 23.1

Box model of flow in a Prolog program. (a) Sequence of sub-goals not containing fail or cut (!). (b) Sequence containing fail but not cut (!). (c) Sequence containing cut (!) but not fail. (d) Model for printing by the built-in predicate write/1. The goal is not resatisfied on backtracking. Control continues to flow backwards. (e) Model for the built-in predicate repeat/0. The goal is satisfied on first meeting repeat and is always resatisfied on backtracking

<i>Do</i>	Input port.	This receives data from the previous sub-goal and then tries to satisfy the sub-goal.
<i>Succeed</i>	Output port.	When the sub-goal succeeds, previously and newly instantiated variables are sent to the next sub-goal.
<i>Fail</i>	Output port.	No new instantiations have been made. Return to the previous sub-goal. That is back-track one step.
<i>Redo</i>	Input port.	Receives control when the following subgoal fails.

The box representing the Built-in Predicate *fail* does not have *Succeed* or *Redo* ports. The box representing cut (!) directs the output from its *Fail* port to the calling goal/query, when it receives an input from the following sub-goal, without performing any other action. Any other clauses are ignored. When cut (!) receives a *Do* input, it freezes all instantiations in the clause.

23.1.2.4 Recursion

Prolog provides just two methods of controlling program “flow”: back-tracking and recursion. We have already encountered recursion, in the definition of *ancestor/2*. Prolog handles recursion very efficiently and is one of the main reasons why it is so powerful. In the following example, *left_of/2* builds on the predicate *left/2*, whose meaning is obvious and which will be defined later.

```
% A is left_of B if A is to (the immediate) left of B
left_of(A,B) :- left(A,B).                % Terminating clause

% A is left_of B if A is to (the immediate) left of C and C is left_of B
left_of(A,B) :- left(A,C), left_of(C,B).  % Recursive clause
```

Structurally, this is similar to *ancestor/2*, although it should be understood that there are many other ways to employ recursion. One of the dangers of using recursion is the possibility of creating programs that never halt. However, an experienced programmer avoids this trap by following one simple rule: place a terminating clause above the recursive clause.

Why is recursion so important? The reason is that many operations leading to “intelligent” behaviour are easily naturally expressed in terms of recursion, rather than iteration. The process of packing 2D shapes is a good example; the procedure is simple:

- (a) Define the space available for placing an object.
- (b) Choose an object to fit in place. This involves optimising its position and possibly its orientation and/or posture (“heads up” or “tails up”).
- (c) Put the object into its chosen place and calculate the space that is unoccupied.
- (d) Apply steps (a) to (d) recursively.

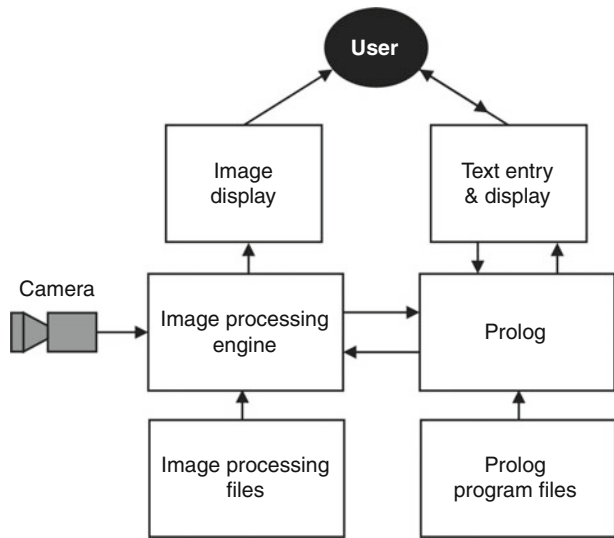
As we shall see later, packing is one of the tasks that Prolog+ is able to perform but which presents major problems for QT, or imperative languages such as C, Java, etc.

23.1.3 Prolog+

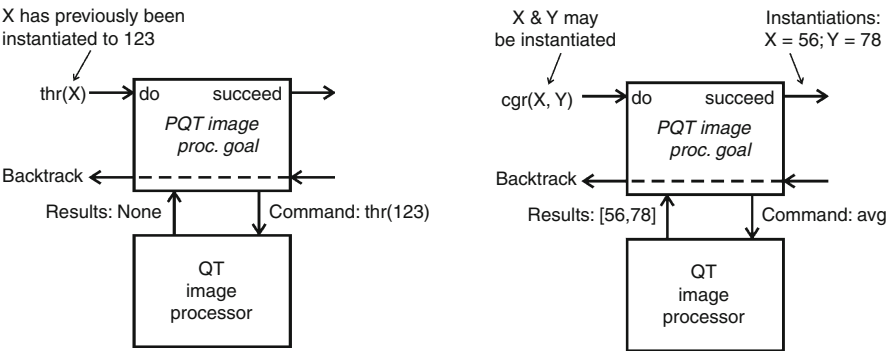
The reader is reminded that Prolog+ is an extension of standard Prolog, in which a rich repertoire of image processing functions is made available as a set of built-in predicates. For convenience, we shall employ QT’s image processing mnemonics in our description of Prolog+. This results in the language PQT that we shall discuss hereafter in this chapter. In the past, other implementations of Prolog+ have been devised [8]. PQT is simply the latest of these.

► [Figure 23.2](#) shows the basic structure of a generic Prolog+ system.

The way that the image processing predicates operate follows the standard pattern established for printing in Prolog (c.f. *nl*, *write*, *tab*). That is, they always succeed but are never resatisfied on back-tracking. For example, as a “side effect” of trying to satisfy the goal *neg*, Prolog+ calculates the negative of QT’s current image. Similarly, the goal *thr(125,193)* always succeeds and in doing so thresholds the current image. Again, this goal is never resatisfied on back-tracking. When X is initially uninstantiated, the goal *cwp(X)* succeeds and instantiates X to the number of white pixels in the current image. However, the goal *cwp(15294)* will only succeed if there are exactly 15294 white pixels in the current image; it will fail otherwise see (► [Fig. 23.3](#)).



■ Fig. 23.2
Basic structure of a Prolog+ system. This may be implemented using one, or two, computers, or one computer, hosting Prolog, and a hardware image processing module



■ Fig. 23.3
Box model for PQT image processing predicates. (a) Image processing operations (e.g., *thr/1*) that do not return results. The goal is not resatisfied on backtracking. Control continues to flow backwards. (b) Model for PQT image processing operations (e.g., *cgr/2*) that return results. The goal is not resatisfied on backtracking. Control continues to flow backwards. If the results calculated by QT do not match the instantiations of X and Y, the goal will fail and backtrack to the previous goal

The distinction between assignment in an imperative language and instantiation in Prolog is important and must be understood before the merits of Prolog+ can be appreciated.) While PQT is trying to prove the compound goal

$avg(Z),thr(Z)$

Z is instantiated to the average intensity within the current image (calculated by *avg*). This value is then used to define the threshold parameter required by *thr*. With these points in mind, we are ready to examine our first PQT program.

grab_and_threshold :-

```

    grb,                % Digitise an image
    caf,                % Blur (Low-pass filter)
    avg(Z),             % Calculate average intensity
    thr(Z).             % Threshold at average intensity

```

Since each sub-goal in this simple program succeeds, the effect is the same as if we had written a sequence of commands using a conventional (i.e., imperative) computer language. Here is the QT equivalent M-function

```

function grab_and_threshold
grb                % Digitise an image
caf                % Blur (Low-pass filter)
Z = avg(Z);        % Calculate average intensity
thr(Z)             % Threshold at average intensity

```

The Prolog+ goal *grab_and_threshold* always succeeds. However, the following PQT program is a little more complicated.

big_changes(A) :-

```

    repeat,           % Always succeeds on backtracking
    grb,              % Digitise an image from the camera
    raf,              % Perform raf (low-pass filter)
    sca(3),           % Retain 3 bits of each intensity value
    rei,              % Read image stored in previous cycle
    swi,              % Switch current & alternate images
    wri,              % Save the image for the next cycle
    sub,              % Subtract the two images
    abv,              % Absolute value of intensity
    thr(10),          % Threshold at intensity level 10.
    cwp(A),           % A is number of white pixels
    A > 100.           % Are image differences significant?

```

The built-in predicate *repeat* succeeds, as does each of the image processing operators, [*grb*, *raf*, ..., *cwp*(A)]. If the test $A > 100$ then fails, the program back-tracks to *repeat*, since none of the image processing predicates is resatisfied on backtracking. (The predicate *repeat/0* is always resatisfied on backtracking.) Another image is then captured from the camera and the whole image processing sequence is repeated. The loop terminates when A exceeds 100. When this happens, the goal *big_changes*(A) succeeds and A is instantiated to the number of white pixels in the difference image. The goal *big_changes*(A) performs the processing sequence [*grb*, ..., *cwp*(A)] an indefinitely large number of times and only succeeds when two consecutive images are found that are significantly different from one another. This program could be used as the basis for a crude intruder alarm that detects when a large object (a person) enters a restricted area. By adjusting the number in the final sub-goal (i.e., $A > 100$), it is possible to tolerate small changes (e.g., a cat

wandering in front of the camera), while still being able to detect large variations due to a person being in view. By now, the reader should be able to write a QT program to perform the same function.

Although image processing commands, such as *grb*, *thr*, *cwp* etc. are always satisfied, errors will be signalled if arguments are incorrectly specified. Since *thr(X)* requires one numeric argument, an error condition will arise if X is uninstantiated. On the other hand, the following compound goal is satisfied

X is 197, thr(X),

as is

<i>gli(A,B),</i>	<i>% Lower & upper intensity limits</i>
<i>C is (A+B)/2,</i>	<i>% Average A and B</i>
<i>thr(C).</i>	<i>% Threshold at average value</i>

The compound goal

X is 186, Y is 25, thr(X,Y)

fails, since *thr* fails when its second argument is less than the first one. (The image processor signals an error, which causes the failure of the Prolog goal, *thr(186,25)*.) The compound goal

X is 1587, thr(X)

fails, because the parameter (X) is outside the range of acceptable values, i.e., [0,255].

Notice that *thr* has already been used with different numbers of arguments. We can use *thr* with 0, 1 or 2 arguments. Other image processing predicates will be treated in the same way. For example, we have already used *wri/1* (write image to disc) and *rei/1* (read image from disc) without arguments. We can also use arguments instantiated to a string of alpha-numeric characters. Such arguments may be generated according to the usual Prolog conventions. For example, the Prolog symbol generator, *gensym/2*, may be used to create a series of file-names, *image_file1*, *image_file2*,... as the following illustration shows:

<i>process_image_sequence :-</i>	
<i>grb,</i>	<i>% Digitise an image</i>
<i>process_image,</i>	<i>% Process the image</i>
<i>gensym(image_file,X),</i>	<i>% Generate new symbol name</i>
<i>wri(X),</i>	<i>% Write image</i>
<i>process_image_sequence.</i>	<i>% Repeat processing</i>

Notice here that we have “condensed” almost all of the image processing into the subsidiary predicate, *process_image/0*. Using this approach, a simpler, revised version of *big_changes/1* may be defined using the subsidiary predicate, *process/0*.

<i>big_changes(A) :-</i>	
<i>process</i>	<i>% Listed below</i>
<i>cwp(A),</i>	<i>% Instantiate A to number of white pixels</i>
<i>A > 100.</i>	<i>% Are differences between images large enough? If not, back-track.</i>

where *process/0* is defined thus:

<i>process :-</i>	
<i>grb,</i>	<i>% Grab an image from the camera</i>
<i>lpf, lpf, lpf,</i>	<i>% Low pass filter applied three times</i>


```

sca(3),      % Reduce number of bits representing intensity to 3
rei,         % Read image from disc
swi,         % Interchange the Current and Alternate images
wri,         % Save image on disc
sub,         % Subtract current and alternate images
abv,         % "Absolute value" (fold intensity scale around mid grey)
thr(1).      % Threshold at level 1

```

The use of subsidiary predicates, such as *process_image* and *process*, allows the programmer to think at a higher conceptual level and to defer deciding exactly what image processing is to be performed until later.

23.1.4 Sample Programs

Now that we have illustrated the basic principles of Prolog+, we are able to consider more advanced programs to illustrate its main features. It is important to realise that we must always use Prolog+ to describe the *image* generated by the camera, not the object/scene being inspected. The importance of this point cannot be over-emphasised. Additional points of general interest will be discussed as they arise.

23.1.4.1 Recognising Bakewell Tarts

► [Figure 23.4](#) which shows a small cake, called a Bakewell tart. We shall explain how Prolog+ might be used to inspect it from an overhead view.

The top level of the program consists of four separate tests, incorporated into the predicate *bakewell_tart/0*:

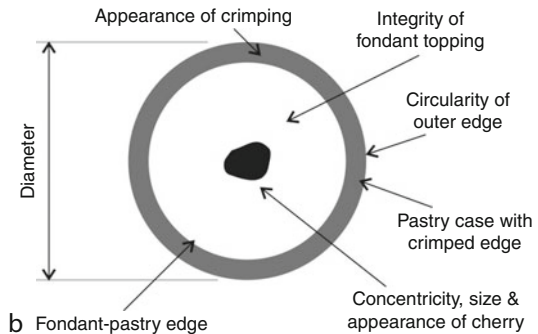
```

bakewell_tart :-
    segment_image,      % Convert image as in ► Fig. 23.4
    outer_edge,         % Check the outer edge
    cherry,             % Check the cherry
    icing.              % Check the icing

```



a



b Fondant-pastry edge

■ Fig. 23.4

Bakewell tart. (a) Photograph (Credit: B. R Wheatley, *copyright free*) and (b) critical features

Programs written in PQT are almost invariably written from the top level downwards. In this instance, *bakewell_tart/0* was the first predicate to be defined. Notice that there are four obvious stages in verifying that the tart is a good one:

- (a) Simplify the image (possibly with 2–4 intensity levels), using *segment_image/0*.
- (b) Check the integrity of the outer edge, using *outer_edge/0*.
- (c) Check the presence, size and placing of the cherry, using *cherry/0*.
- (d) Check the icing, using *icing/0*.

Clearly, *bakewell_tart/0* is only satisfied if all four of the subsidiary tests succeed. Although the secondary predicates are not defined yet, this is not necessary for us to understand the broad concepts involved in recognising a Bakewell tart. Three of these are defined below. (*segment_image/0* is not given here, because it is problem specific and would distract us from the main point.)

```

outer_edge :-
    thr(1),                % Select outer edge
    circular.              % Test for circularity. Defined below

cherry :-
    thr(1),                % Select outer edge
    cgr(X1,Y1),            % Centroid of outer edge
    swi,                   % Switch images
    thr(200),              % Select cherry
    swi,                   % Switch images
    cgr(X2,Y2),            % Centroid of the cherry
    distance([X1,Y1],[X2,Y2],D),
    % D is distance [X1,Y1] to [X2,Y2],
    D < 20.                % Are cherry & outer edge concentric?

icing :-
    thr(128),              % Select icing
    rei(mask),             % Read annular mask image from disc
    exr,                   % Differences between these images
    cwp(A),                % Calculate area of white region
    A > 50.                % Allow few small defects in icing

circular :-
    cwp(A),                % Calculate area
    perimeter(P),          % Calculate perimeter.
    S is A/(P*P),          % Shape factor
    S < 0.08.              % S >= 1/(4*π) (min. for circle)

```

Notice the highly modular approach to Prolog+ programming and the fact that it is possible to define what is an “acceptable” Bakewell tart, in a simple and natural way. Apart from the predicate *segment_image/0*, whose definition depends upon the lighting and camera, the program *bakewell_tart/0* is complete and provides a simple yet effective means of inspecting Bakewell tarts.

23.1.4.2 Recognising Printed Letters

This application is included to illustrate Prolog+, rather than explain how a practical optical character recognition system might work. The top two layers of a Prolog+ program for recognising printed letters are given below:

```

% Top level predicate for recognising printed
% letters, which may be either upper or lower case
% and in any one of three fonts.
letter(X) :- upper_case(X).           % May be upper case ...
letter(X) :- lower_case(X).          % ... or lower case

upper_case(X) :-
    font(Y),                          % Find what font we are using
    member(Y,[times,courier,helvetica]),
                                         % Is font Y known to us?
    recognise_upper_case(X,Y).        % X is upper case in font Y

lower_case(X) :-
    font(Y),                          % Find what font we are using
    member(Y,[times,courier,helvetica]),
                                         % Is font Y known to us? to us
    recognise_lower_case(X,Y).        % X is lower case in font Y

```

The complex task of recognising an upper- or lower-case letter in any of the three known fonts has been reduced to a total of 156 ($=3*2*26$) much simpler sub-problems. (A simple declarative definition of the *sans serif* upper-case letter A is presented later.) Now, let us consider what changes have to be made if a new font (e.g., *Palatino*) is to be introduced. Two changes have to be made:

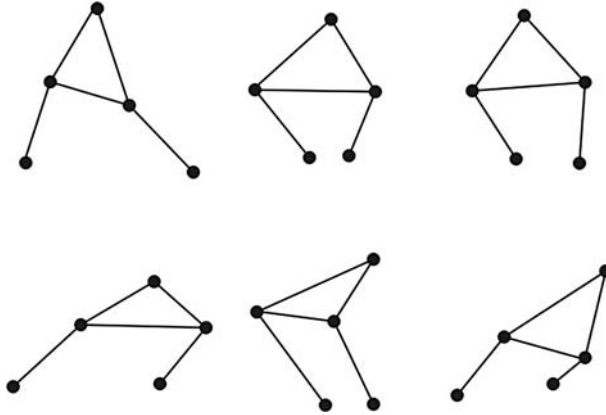
- (i) the second line in the body of *upper_case/1* and *lower_case/1* is changed to
`member(Y,[times,courier,helvetica,palatino])`
- (ii) two new clauses are added for each letter X, one for
`recognise_upper_case(X,palatino)`
 and another for
`recognise_lower_case(X,palatino).`

If we wanted to add recognition rules for the numeric characters, then 10 new clauses would be added, as in (ii). In other words, extending the scope of a Prolog+ program is conceptually simple, if rather tedious to accomplish. Here, as promised, is a naive but quite effective declarative definition of the *sans serif* upper-case letter A:

```

recognise_upper_case(a,sans_serif) :-
    apex(A),                          % There is an apex called A.
    tee(B),                           % There is a T-joint called B
    tee(C),                           % There is a T-joint called C
    line_end(D),                      % There is a line_end called D

```



■ Fig. 23.5

Objects that are inappropriately recognised as the sans serif upper-case letter A by the goal *recognise_upper_case(a,san_serif)*

```

line_end(E),                                % There is a line_end called E
above(A,B),                                % A is above B.
above(A,C),
about_same_vertical(B,C),
about_same_vertical(D,E),
above(B,D),
above(C,E),
connected(A,B),
connected(A,C),
connected(B,D),
connected(C,E),
connected(B,C),
left(B,C),
left(D,E).

```

The reader should be able to understand the above program without detailed knowledge about how the predicates *apex/1*, *tee/1*, *line_end/1*, *above/1*, *about_same_vertical/2*, *above/2*, *connected/2*, *left/2* and *right/2* are defined. ➤ Figure 23.5 shows some of the objects which are recognised by this program. Obviously, *recognise_upper_case/2* can be refined by adding further conditions, to eliminate some of the more bizarre objects that are recognised by the present definition and which are shown in ➤ Fig. 23.5.

23.1.4.3 Identifying Table Cutlery

The following Prolog+ program identifies items of table cutlery that are viewed in silhouette. (It is assumed, for the sake of brevity, that the input image can be segmented using simple thresholding.) The top-level predicate is *camera_sees(Z)* and is a general purpose utility that can find any object, given an appropriate definition for the subsidiary predicate

object_is/1. In its present somewhat limited form, the program recognises only forks and knives; additional clauses for *object_is/1* are needed to identify other utensils, such as spoons, plates, mats, etc. When the query *camera_sees(Z)* is specified, *Z* is instantiated to the type of object seen by the camera. In practice, there may be many objects visible to the camera and the program will progressively analyse each one in turn. Any objects visible to the camera that are not recognised are indicated by instantiating *Z* to the value *unknown_type*.

% Top level predicate for recognising individual items of table cutlery

camera_sees(Z) :-

<i>grb,</i>	<i>% Digitise the camera image</i>
<i>segment_image,</i>	<i>% Example: [enc, thr(128)]</i>
<i>ndo,</i>	<i>% Shade resulting binary image; each</i>
	<i>% blob has different intensity</i>
<i>wri(temp),</i>	<i>% Save image in disc file</i>
<i>repeat,</i>	<i>% Begin loop - analyse blobs in turn</i>
<i>next_blob,</i>	<i>% Select 1 blob from image in "temp"</i>
<i>object_is(Z),</i>	<i>% Blob is object of type Z</i>
<i>finished.</i>	<i>% Succeeds when no more blobs</i>

% Select one blob from the image stored in disc file "temp".

% Remove this blob from the stored image, so that it will not be considered next time.

next_blob :-

<i>rei(temp),</i>	<i>% Read image from disc file "temp"</i>
<i>gli(_A),</i>	<i>% Identify next blob - i.e., brightest</i>
<i>hil(A,A,0),</i>	<i>% Remove it from stored image</i>
<i>wri(temp),</i>	<i>% Save remaining blobs</i>
<i>swi,</i>	<i>% Revert to previous stored image</i>
<i>thr(A,A).</i>	<i>% Select one blob</i>

% Recognises individual non-overlapping objects in a binary image. See Fig. 23.6

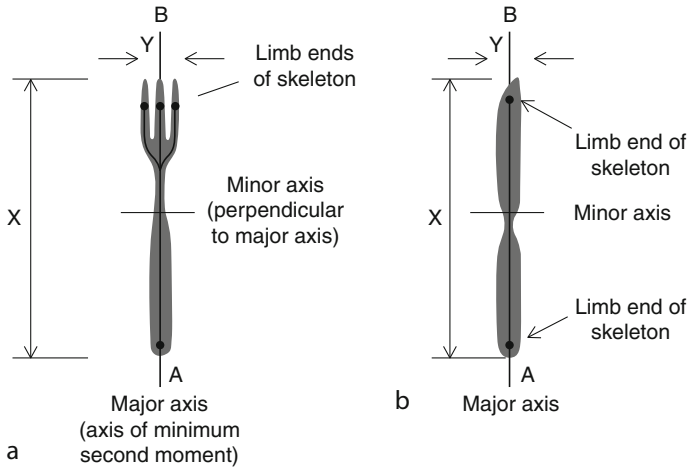
object_is(fork) :-

<i>mma(X,Y),</i>	<i>% Lengths of major & minor axes. mma is not defined here</i>
<i>X >= 150,</i>	<i>% Length must be >= 150 pixels</i>
<i>X <= 450,</i>	<i>% Length must be <= 450 pixels</i>
<i>Y >= 25,</i>	<i>% Width must be >= 25 pixels</i>
<i>X <= 100,</i>	<i>% Width must be <= 100 pixels</i>
<i>Z is X/Y,</i>	<i>% Calculate aspect ratio - whatever</i>
	<i>% orientation</i>
<i>Z <= 10,</i>	<i>% Aspect ratio must <= 10</i>
<i>Z >= 4,</i>	<i>% Aspect ratio must be >= 4</i>
<i>count_limb_ends(N),</i>	<i>% N is number of limb ends</i>
<i>N >= 3,</i>	<i>% Skeleton of fork has >= 3 limb ends</i>
<i>N <= 5.</i>	<i>% Skeleton of fork has <= 5 limb ends</i>

% Add extra clauses to recognise each possible type of object. See Fig. 23.4

object_is(knife) :-

<i>mma(X,Y),</i>	<i>% Lengths of major & minor axes</i>
<i>X >= 150,</i>	<i>% Length must be >= 150 pixels</i>
<i>X <= 450,</i>	<i>% Length must be <= 450 pixels</i>



■ Fig. 23.6

Explaining the operation of two object recognition programs (a) *object_is(fork)*. AB is the axis of minimum second moment. Criteria to be satisfied, before this object can be accepted as a fork: $150 \leq X \leq 450$; $25 \leq Y \leq 100$; $4 \leq X/Y \leq 10$; skeleton here has 3–5 limb ends. (b) *object_is(knife)*. AB is the axis of minimum second moment. Criteria to be satisfied, before this object can be accepted as a fork: $150 \leq X \leq 450$; $25 \leq Y \leq 100$; $6 \leq X/Y \leq 12$; skeleton must have exactly two limb ends

```

Y >= 25,           % Width must be >= 25 pixels
X <= 100,          % Width must be <= 100 pixels
Z is Y/X,          % Calculate aspect ratio - whatever
                   % orientation
Z <= 12,           % Aspect ratio must be <= 12
Z >= 6,            % Aspect ratio must be >= 6
count_limb_ends(2). % Skeleton of knife has 2 limb ends

object_is(unknown_type). % Catch-all clause. Not recognised

% Search is finished. Image is black everywhere.
finished :-
    rei(temp),      % Read image from disc
    thr(1),         % Threshold stored image
    cwp(0).         % Succeeds if no. of white points = 0
% Count the limb ends on a skeleton ("match-stick") figure.
count_limb_ends(N) :-
    mdl,            % Generate skeleton of the blob
    cnw,            % Count white neighbours, 3*3 window
    min,            % Ignore back-ground points
    thr(2,2),       % Select limb ends
    eul(N).         % Instantiate N to no. of limb ends

```

23.1.4.4 Analysing All Visible Objects


The list of all objects that are visible to the camera can be found using the predicate *list_all_objects/1* defined below. This will be useful in the next section.

```
list_all_objects(A) :-
    grb,                % Same pre-processing ...
    segment_image,      % ... as is used in ...
    ndo,                % ... the predicate ...
    wri(temp),           % ... "camera_sees"
    find_object_list     % Generate list of objects seen
    ([],A).

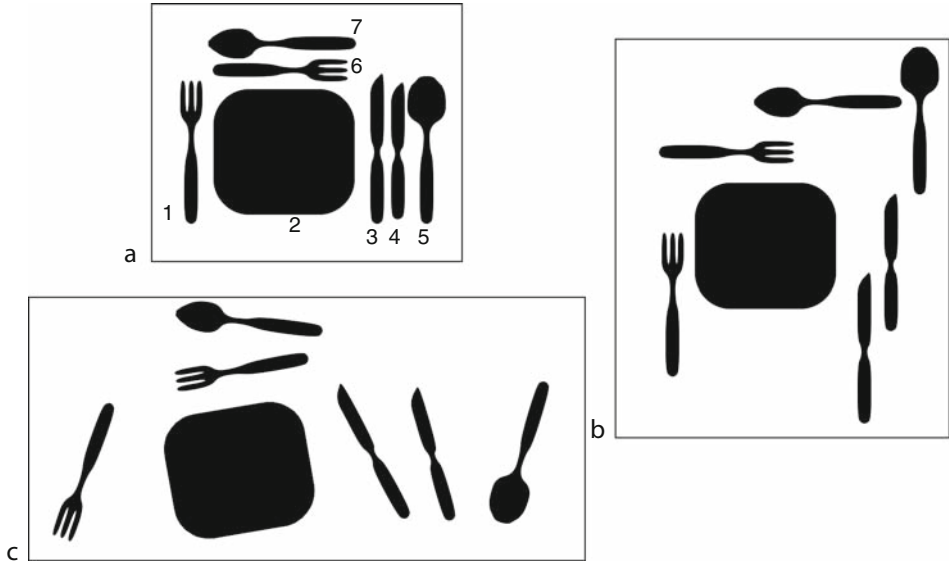
% Terminating recursion. Succeeds when there are no more blobs left to analyse.
find_object_list(A,A) :- finished.
% Analyse all blobs in the image
find_object_list(A,B) :-
    next_blob,          % Select blob from image in file "temp"
    object_is(C),       % Blob is object of type C
    !,                  % Cut (!) makes recursion more efficient as it freezes
                        % variable C
    find_object_list    % Recursion - analyse all blobs
    ([C|A],B).
```

This makes use of the fact that *object_is/1* is able to recognise an object in the presence of other objects, provided that they do not touch or overlap. We will now make good use of *list_all_objects/1* in performing a more challenging task, namely that of recognising a well-laid table place setting.

23.1.4.5 Recognising a Table Place Setting

The following Prolog+ program can recognise a table place setting with the cutlery laid out as shown in  Fig. 23.7a. For this program to work properly, we must first define additional clauses for the predicate *object_is*, so that appropriate types of object, such as *small_knife/1*, *tea_spoon/1*, *dinner_fork/1*, *plate/1*, *mat/1*, etc., can be recognised. Notice that *table_place_setting/0* is defined in standard Prolog, without any further use of the image processing built-in predicates.

```
table_place_setting :-
    list_all_objects(A),
    equal_sets(A, [mat, plate, dinner_knife, small_knife,
    dinner_fork, small_fork, soup_spoon, desert_spoon]),
    left(dinner_fork, mat),                % Defined below
    left(mat, dinner_knife),
    left(dinner_knife, small_knife),
    left(small_knife, soup_spoon),
    below(mat, small_fork),               % Defined below
    below(small_fork, desert_spoon).
```



■ Fig. 23.7

Table place settings: (a) Ideal arrangement. Key: 1, dinner_fork; 2, mat; 3, dinner_knife; 4, small_knife; 5, soup_spoon; 6, small_fork; 7, desert_spoon. (b) Scene that is misrecognised by *table_place_setting/0* with the original definitions of *left/2* and *below/2* but which is correctly rejected by the revised version. (c) A scene that is incorrectly recognised by the revised version of the program

For completeness, we now define the *predicates left/2, below/2 and equal_sets/2*. These and many other useful “general purpose” predicates like them form part of a PQT Library, which augments the basic language.

```

left(A,B) :-
    location(A,Xa,_),           % Horizontal position of A
    location(B,Xb,_),           % Horizontal position of B
    !,                          % Inhibit backtracking
    Xa < Xb.                    % Compare horizontal positions

below(A,B) :-
    location(A,_,Ya),           % Vertical position of A is Ya
    location(B,_,Yb),           % Vertical position of B is Yb
    !,                          % Inhibit backtracking
    Ya < Yb.                    % Compare vertical positions

equal_sets([],[]).              % Terminate recursion
% Checking two non-empty lists are equal
equal_sets([A|B],C) :-
    member(A,C),                % A is a member of C
    cull(A,C,D),                 % Delete A from C. Result D
    !,                          % Improve efficiency
    equal_sets(B,D).             % Are sets B & D equal?
cull(_,[],[]).                  % Empty list, Do nothing

```



```

% Delete A from list if A is at its head
cull(A,[A|B],C) :-
    !,                                % Don't back-track - improves efficiency
    cull(A,B,C).                      % Repeat until A & B are empty
% A is not head of "input" list [B|C] so work on its tail
cull(A,[B|C],[B|D]) :-
    !,                                % Improves efficiency
    cull(A,C,D).                     % Repeat until A & B are empty

```

Using these simple definitions, a range of unusual configurations of cutlery and china objects is accepted (see [Fig. 23.7b](#)). To improve matters, we should refine our definitions of *left/2* and *below/2*. When redefining *left(A,B)*, we simply add extra conditions, for example that *A* and *B* must be at about the same vertical position:

```

left(A,B) :-
    location(A,Xa,Ya),
    location(B,Xb,Yb),
    !,
    Xa < Xb,                        % Compare horizontal positions
    about_same(Ya,Yb, 25).          % Tolerance level = 25
about_same(A,B,C) :- A =< B + C.
about_same(A,B,C) :- A >= B - C.

```

The predicate *below/2* is redefined in a similar way:

```

below(A,B) :-
    location(A,Xa,Ya),
    location(B,Xb,Yb),
    !,
    Ya < Yb.                        % Compare vertical positions
    about_same(Xa,Xb, 25).          % Tolerance level = 25

```

In a practical application, it would probably be better for the tolerance parameter required by *about_same/3* (third argument) to be related to the size of the objects to be recognised. This would make a program such as *table_place_setting/0* more robust, by making it size-independent. This modification to our program improves matters, but it still recognises certain cutlery arrangements as being valid place settings, even though we would probably want to exclude them in practice (see [Fig. 23.7c](#)). Clearly, we can go on adding further conditions to our program, in order to reduce the number of cutlery arrangements accepted by it.

In [Part 2](#), we discuss the task of recognising a well-laid place setting, using a natural language (English) description of the ideal layout.

23.1.5 Justifying the Design

We have repeatedly emphasised that Prolog is able to handle abstract concepts, expressed in symbolic form. In this section, we shall try to justify this claim and explain why our latest implementation of Prolog+ was based on QT, rather than the unadorned MATLAB Image

Processing Tool-box. In doing so, we shall explain how to recognise items of fruit in PQT and then pack a lunch box that provides a well-balanced diet. This is achieved using the same language that we might use for other very different tasks such as inspecting piece parts, identifying flaws in a woven fabric, operating a robot or controlling a lighting array.

23.1.5.1 Additional Spatial Relationships (🔗 Fig. 23.8)

The predicate *table_place_setting/0* relies upon *abstract* concepts about spatial relationships between pairs of objects in an image. The relationships in question (i.e., *left/2* and *below/2*) were defined earlier. Another predicate based on the spatial relationship between two objects is *encloses/2* and is clearly related to its reciprocal relationship, *inside/2*:

```
% A encloses B if B is inside A
encloses(A,B) :- inside(B,A).
inside(A,B) :-
    wri(image),                % Save the original image
    isolate(A),               % Isolate object A
    wri(A),                   % Save image until later
    rei(image),               % Recover the original image
    isolate(B),               % Isolate object B
    rei(A),                   % Recover saved image during "wri"
    sub,                       % Subtract images
    thr(0,0),                 % Find all black pixels
    cwp(0).                   % There are zero black pixels
```

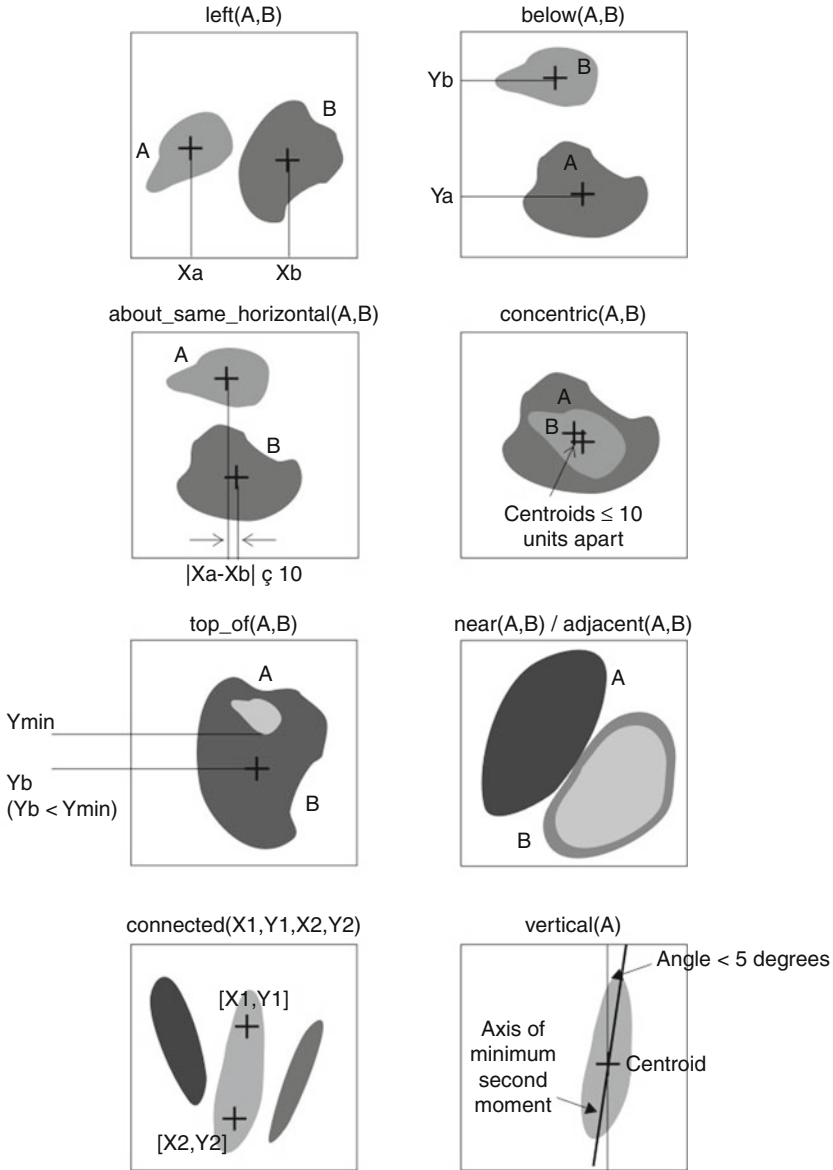
Notice that *inside(A,B)* creates two binary images, containing objects *A* and *B*. These images are then compared. If all white pixels in one image (i.e., the one containing object *B*) are also white in the other (containing object *A*), then we can conclude that *A* is inside *B* and the goal *inside(A,B)* succeeds.

There are, of course, many other abstract relationships of this kind. Here are the definitions of a few of them:

```
% Are objects A and B concentric?
concentric(A,B) :-
    location(A,Xa,Ya),        % Definition may be based on centroid
    location(B,Xb,Yb),
    near([Xa,Ya],[Xb,Yb],10). % Distance [Xa,Ya] to [Xb,Yb] =< 10?

% Are objects A and B in about the same vertical position?
about_same_vertical(A,B) :-
    location(A,Xa,Ya),        % Definition may be based on centroid
    location(B,Xb,Yb),
    about_same(Ya,Yb,10).     % Is |Ya - Yb| =< 10?
```

```
% Test whether object A is in the upper part of
% object B (i.e., the bottom-most point in A is
```



■ Fig. 23.8
Spatial relationships in Prolog+

% above B's centroid). A must be entirely contained inside B.

top_of(A,B) :-

```
isolate(A),
dim(_,Ya,_,_),
isolate(B),
cgr(B,_,Yb),
```

```
% Isolate object A
% Bottom of object A
% Isolate object B
% Centroid of B
```

```

Ya =< Yb, % Is bottom point in A above the centre of B?
inside(A,B).

% Are the points [X1,Y1] and [X2,Y2] connected by a continuous set of white pixels?
connected(X1,Y1,X2,Y2) :-
    ndo, % Shade blobs.
    pgt(X1,Y1,Z), % Z is intensity at [X1,Y1]
    pgt(X2,Y2,Z), % Z is intensity at [X2,Y2]
    Z = 255. % Both pixels are white

% Are regions A and B adjacent?
adjacent(A,B) :-
    isolate(A), % Isolate region A
    exw, % Expand region by 1 pixel
    wri, % Save image for use later
    isolate(B), % Isolate region A
    rei, % Read image A expanded
    mni, % Logical AND of the 2 images
    cwp(N), % Count white points
    N > 0. % Are there some?

```

The reader may like to consider how a predicate can be defined that can test whether an object in a binary image is nearly vertical (🔗 [Fig. 23.8h](#)). (The pivotal QT function is *lmi*.) This predicate could be used to prevent *table_place_setting/0* from detecting scenes like that shown in 🔗 [Fig. 23.7c](#).

23.1.5.2 High-Level Concepts

Suppose that we want to devise a Machine Vision system to identifying highly variable objects that conform to a set of rules, expressed either formally, or in a natural language, such as English, Dutch or German. We might, for example, want to recognise objects in an image that are

yellow
sausage-like in shape
curved, with a bend of 5–30°
15–50 mm wide
70–200 mm long.

There are no prizes for guessing what class of objects we are trying to recognise! We can express the recognition rule in Prolog+, in the following way:

```
object(banana) :-
    isolate(yellow),      % Keep "yellow" objects
    big,                  % Ignore small objects
    sausage,              % Shape recognition rule based on "length" and "width"
    curved,               % Another test for shape
```

```

width(W),           % Measure the width
W >= 15, W <= 50,   % Arithmetic tests
length(L),          % Measure the length
L >= 70, L <= 200.  % Arithmetic tests

```

Look around the room where you are reading this. Is there anything in it that could possibly match this specification? It is most unlikely that there is, unless of course, you happen to have a banana nearby! However, a few other things do look like bananas (e.g., plastic replica bananas, sweet-corn (maize) cobs, rolls of yellow paper, etc.) In addition, some perfectly good bananas may be too large, too small, or be so deformed that this rule fails. Indeed rule-based recognition criteria will almost always make some mistakes. However, if we have been diligent in seeking the most appropriate discriminating criteria, the error probability should be quite low. This approach frequently results in heuristic, rather than algorithmic, rule-based recognition procedures (see 🔍 Chap. 17). There are no formal, universally acceptable rules for recognising a “sausage”, or a “curved” object. Human beings do not always agree about what is/ is not “yellow”. (The author and his wife often disagree about what is blue/green! See 🔍 Chaps. 4 and 🔍 16) Despite this, our test for “banana-ness” is quite precise; very few objects exist that pass such a test. The reason is that we have juxtaposed several highly specific factors, relating to colour, shape and size. We might also add descriptions of texture, by including a test for “spottiness”, based upon the density of small dark features within a yellow blob. The reader may like to contemplate how such an idea can be expressed in PQT.

The recognition rule for “banana-ness” relies on elementary concepts that can each be specified in QT. Clearly, “raw” MATLAB, NeatVision (🔍 Chap. 22) and similar systems might be used instead. Each of these is, on its own, able to perform all of the necessary image processing, arithmetic and logical operations needed to implement the object(banana) recognition rule. We shall explain later why Prolog+ has been implemented using QT. For the moment we shall concentrate on justifying the use of Prolog. To do this, consider the following extensions of our program.

First of all, bananas can be green, yellow and green, yellow, or yellow with brown spots. Our program can easily be modified to recognise under-ripe, ready to eat and over-ripe bananas:

```

fruit(banana) :- object(under_ripe_banana).
fruit(banana) :- object(ripe_banana).
fruit(banana) :- object(over_ripe_banana).

```

Each of these three clauses then requires the description of a banana in a different state of ripeness. While this adds to the total quantity of program code that we must create, each of the new subsidiary predicates (i.e., *object(under_ripe_banana)*, *object(ripe_banana)* and *object(over_ripe_banana)*) is easy to write. We have divided the problem into simpler rules that each deals with a separate state of affairs.

Moreover, we can recognise the more general predicate *fruit*, by combining several rules like the one already listed:

```

fruit(apple) :- object(apple).
fruit(banana) :- object(banana).
fruit(grapefruit) :- object(grapefruit).
fruit(kiwi) :- object(kiwi).

```

```
fruit(orange) :- object(orange).
fruit(pear) :- object(pear).
```

Of course, each class, such as *apple*, can be sub-divided, which may will probably make the program easier to write:

```
object(apple) :- object(cox_orange_pippin_apple).
object(apple) :- object(golden_delicious_apple).
object(apple) :- object(macintosh_apple).
object(apple) :- object(pink_lady_apple).
```

In addition, we can plan a lunch box thus:

```
lunch_box(A,B,C,D,E) :-
    savoury(A),           % Savoury item (e.g., ham sandwich))
    desert(B),            % Choose a desert (e.g., cake)
    fruit(C),             % Choose one item of fruit
    fruit(D),             % Choose another item of fruit
    not(C = D),           % Items of fruit must be different
    drink(E).             % Choose a drink
```

It is a straightforward matter to add extra criteria for checking dietary balance; we simply add a database of important nutritional factors for each food type and add simple tests to make sure that the lunch-box contains a healthy meal.

We are able to plan the contents of the lunch box in the very same language that we use to perform a wide variety of image processing operators. We have moved quickly and effortlessly from image-based recognition criteria of specific types of fruit (e.g., ripe bananas), through the more general concept of *banana*, then to the bigger class *fruit*. After a short step further, we are able to discuss the planning of a healthy meal. We can do all this in PQT, which is also well suited to tasks such as controlling a robot and multi-function lighting rig.

PQT is an extension to Prolog, giving it the ability to sense the world visually. This does not limit Prolog's powers in any way. So, all of the arguments that support the use of Prolog in applications requiring abstract reasoning are all still valid.

Notice that the Prolog inference engine is not used to handle images directly; QT does that. This is an important point as Prolog does not handle large volumes of numeric data comfortably.

23.1.5.3 Why Base Prolog+ on QT?

Thus far, we have explained why we have used Prolog but have not yet justified the reliance on QT. There are several points in answer to this question.

1. QT is built on the solid ground of experience and is the latest in a series of interactive image processing systems whose development began over 30 years ago [9]. During this time, the command repertoire has been developed specifically for analysing industrial inspection and visual control applications. Some of the commands available in MATLAB, for example, do not have any obvious use in industrial Machine Vision systems.

2. QT's functional command structure fits in well with Prolog's syntax. Although the two are not directly compatible it is a straightforward matter to translate from one form into the other. On the other hand, MATLAB's basic command structure is not so easily matched to Prolog.
3. QT has a standardised image format and a simple operating paradigm, based on the current-alternate image model. While it may seem to be restrictive, QT's heavy reliance on monochrome images reflects current usage in many industrial Machine Vision systems and has never been observed to cause even minor difficulties. Recall that QT can also perform a wide variety of operations on colour images, despite this constraint. (If the 8-bit model ever proved to be unduly restrictive, it is reassuring to note that a 16-bit version of QT could be produced with reasonable ease.) On the other hand, "raw" MATLAB allows the use of certain image formats, most noticeably sparse and indexed arrays, that are unlikely to be used in dedicated, high-speed systems. In short, MATLAB's great versatility makes it more difficult to use, compared to QT.
4. PQT must be capable of being operated interactively, as well as in programmed mode. It is important therefore that the same commands be used in both modes of operation.

QT is simple to operate because it does not require the user to think about the identity of the image being processed nor where the result will go. Consider a simple image processing function, such as negation (*neg*). In QT, the input image is implicitly taken to be the *current* image (see [◆ Chap. 21](#)). The result is placed in the *current* image and the original (i.e., the contents of the current image just before *neg* was performed) is stored in the *alternate* image. Almost all QT functions behave in this way. Dyadic operators (e.g., *adi*, *sub*, *mxl*) require two inputs. Again, these are the current and alternate images. In neither case, is there any need to specify the source and destination images. Here is the equivalent operation to *neg* expressed in "raw" MATLAB format:

```
global current
global alternate
alternate = current;
current = imcomplement(current);
subplot(1,2,1)
imshow(current)
subplot(1,2,2)
imshow(alternate)
```

(For the sake of fairness, we should shorten the image names to minimal size.) It requires a lot of typing to perform even a simple task, such as negate (*neg* in QT) and is certainly not conducive to maintaining effective user interaction. In addition, the user has to keep a mental note of all the images that have been created. QT allows additional images to be created but normally discards those that will not be needed again.

23.1.5.4 Using Prolog+

Using PQT for interactive image processing is almost as easy as using QT. In response to each system prompt, the user types a simple query. The processed image is displayed almost

immediately. The syntax is, of course, slightly different for PQT and QT; the latter sometimes requires a few more characters to be typed. As a result, the computational speed and effectiveness of the user interaction are hardly diminished. It is just as easy to write a new PQT predicate, as it is to define a MATLAB script to execute a sequence of QT functions.

In normal practice, when using PQT, it is best to group together any long sequences of image processing commands before beginning to program in conventional “Prolog mode”. There is no point in carrying a burden of perhaps 20, or more, separate image processing operations when writing programs that involve backtracking and recursion. As far as possible, “wrap up” standard image processing sequences first. Thereafter, try, as far as possible, to write Prolog code from the top down, bearing in mind what image features can be detected, located and measured, easily. In many cases, morphology can be used to good effect to do this (see [Chap. 19](#)). It is a straightforward matter then to express any relationships that exist between them, image features by declarative programming.

We have already seen that the spatial relationships between the apex, T-joints and limb-ends of upper-case letter A are simple to define in terms of *left_of/2*, *above/2*, etc. Higher level spatial relationships have been defined ([Fig. 23.8](#)). In addition, region properties can be expressed in PQT predicates, including *spotty/1*, (*spotty(X)* tests whether region/feature *X* is spotty), *smooth/1*, *striped/1*, *darker/2* (*darker(X,Y)* tests whether region *X* is darker than region *Y*), *brighter/2*, etc. Do not be afraid to use very simple definitions of relationships and qualities like these. In general terms, do not be tempted to be very precise when programming in PQT. Such a blatant lack of precision presents some people with a fundamental difficulty. However, the maxim when programming in PQT is to accept that it is concerned with “sloppy programming”. (The word *fuzzy* would be better but this invokes ideas of Fuzzy Sets, which is not intended.) This approach is justified in [Chaps. 2](#) and [17](#).

23.2 Part 2: Using Prolog to Understand Natural Language

23.2.1 Prolog and Natural Language

It has long been the ambition of computer scientists to use natural language for programming. While the structure of a language, such as English, Welsh or German, is very complex, it is possible to represent a *small subset* using Prolog. For example, the sentences that might be needed to describe the movements of pieces on a chess board use both a restricted vocabulary and have a well-defined structure that can be represented using Prolog. Of course, a human being can always find ways to confuse a machine, by perversely using complex phrasology and abstruse references. For the purposes of the present discussion, it is expected that human beings will behave reasonably when holding conversations with a machine.

23.2.1.1 Definite Clause Grammars

One popular approach to linguistic analysis relies upon *Definite Clause Grammars* (DCGs), which form an integral part of Prolog. DCGs closely resemble the familiar Backus-Naur Form (BNF), which is frequently used to define computer language syntax. A definite clause grammar consists of a set of *production rules* of the form


```

head -> body.                % To recognise "head" recognise "body".
body -> part1, ..., partn.    % Concatenation of part1, ..., partn.

```

Production rules may also contain the connective operator ‘;’ (sometimes ‘|’ is used instead), which allows elements to be ORed together. For example,

```
head -> test1 ; test2 ; test3 ; . . . . . ; testn.
```

states that in order to recognise *head*, it is sufficient to recognise *any one* (or more) of the subsidiary tests: *test1*, *test2*, *test3*,, *testn*. An alternative is to use several separate rules. Thus:

```

head -> test1.
head -> test2.
head -> test3.
etc

```

has the same meaning as:

```
head -> test1 ; test2 ; test3.
```

or

```

head -> test1.
head -> test2 ; test3.
etc

```

The operator ‘,’ (comma) is used to signify that two or more tests must *all* be satisfied. Thus:

```
head -> test1, test2, test3.
```

states that, in order to satisfy *head*, all three of the subsidiary tests (*test1*, *test2*, *test3*) must be satisfied. Syntactically, production rules closely resemble Prolog rules. However, it should be understood that they are treated in a different way.

The body of a DCG is composed of terminal symbols and conditions, separated by commas or semi-colons. Here is a very simple DCG which describes traditional British personal names

```

name -> forename, surname.                % Forenames & surname
forename -> forename3 ; forename 4.        % Multiple forenames
forename3 -> forename1 ; forename3, forename1. % Male names
forename4 -> forename2 ; forename4, forename2. % Female names
% Items in square brackets [...] are terminal symbols, which Prolog does not try to analyse
forename1 -> [philip] ; [edward] ; [andrew] ; [jonathan] ; [godfrey].
forename2 -> [elizabeth] ; [elaine], [susanna] ; [louisa] ; [victoria] ; [catherine], [grace].
surname -> [smith] ; [higgins] ; [cook] ; [martin] ; [jones].    % Terminal symbols

```

Here are a few of the possible names that satisfy these grammar rules.

```

[philip, andrew, cook]
[elizabeth, susanna, martin]

```

[jonathan, philip, higgins]

[godfrey, jonathan, godfrey, philip, godfrey, edward, godfrey, andrew, godfrey, jones]

The first three are quite acceptable. However, the last one is silly and is permitted since we have chosen to use an over-simplified set of grammar rules.

23.2.1.2 Example: Moving Chess Pieces

The grammar rules listed below define a language for controlling a robot that moves pieces around a chess board. (They do not yet explain how “meaning” is extracted from the commands to operate a real robot.)

move → *order, man, [to], position.*

move → *order, man, [from], position, [to], position.*

order → *movement ; [please], movement.*

movement → *[move] ; [shift] ; [translate] ; [transfer] ; [take].*

man → *article, color, piece.*

article → *[] ; [my] ; [your] ; [a] ; [the].*

color → *[] ; [white] ; [black].*

piece → *[pawn] ; [rook] ; [castle] ; [knight] ; [bishop] ; [queen] ; [king].*

position → *[column], number, [row], number ; [row], number, [column], number.*

position → *number, number.*

number → *[1] ; [2] ; [3] ; [4] ; [5] ; [6] ; [7] ; [8].*

number → *[one] ; [two] ; [three] ; [four] ; [five] ; [six] ; [seven] ; [eight].*

The following sentences are accepted by this grammar:

[please, move, my, white, pawn, to, column, 5, row, 6]

[shift, a, black, pawn, to, row, 4, column, 6]

[please, move, the, white, queen, from, column, 3, row, 6, to, column, 5, row, 6]

[transfer, my, bishop, from, 5, 6, to, 7, 3]

[take, the, black, rook, to, 7, 3]

[please, transfer, my, knight, from, row, 3, column, 2, to, 6, 8]

23.2.1.3 Parsing

DCG rules are recognised by a parser, which is embodied in the Prolog predicate *phrase/3*. However, for our present purposes, the simpler form

parse(A,B) :- phrase(A,B,[]).

is preferred. The goal

parse(A,B)

tests to see whether the list B, which may contain both terminal symbols and variables, conforms to the grammar rules defined for A. Here is a simple example, showing how it is used in practice:

parse(move, [please, move, the, white, queen, from, column, 3, row, 6, to, column, 5, row, 6]).

Prolog will simply satisfy this goal. On the other hand, the following goal is not satisfied

```
parse(move, [please, do, not, move, the, white, queen, from, column, 3, row, 6, to, column, 5,
row, 6]).
```

An important feature of the parser is that it allows us to analyse sentences that are incompletely specified. (Notice that the variable *Z* is initially uninstantiated.) Consider the compound goal

```
parse(move, [take, the, Z, to, row, 3, column, 2]), write(Z), nl, fail.
```

This prints seven possible instantiations for *Z*:

```
pawn, rook, castle, knight, bishop, queen, king
```

This suggests a possible method for extracting “meaning” from a sentence:

- Use the parser to check that the sentence conforms to the given set of grammar rules.
- Use the unification process to search for specific symbols in certain places within the given sentence.

23.2.1.4 Another Example: Controlling a Set of Lamps

The following rules define a grammar for controlling a set of lamps. Comments indicate possible symbols that would be accepted by the parser. At this stage there is no attempt, to extract meaning or switch the lamps on/off.

% Simplify the problem by breaking it into simpler sub-problems.

Examples of each object type are given in the comments (i.e. following the % symbol)

```
light -> light1 ; light2 ; light3 ; light4 ; light5 ; light6 ; light7 ; light8.
```

```
light1 ->
```

<i>light_verb,</i>	<i>% set</i>
<i>type1,</i>	<i>% every</i>
<i>light_dev1,</i>	<i>% lamp</i>
<i>light_par1.</i>	<i>% [to,level,5]</i>

```
light2 ->
```

<i>light_verb,</i>	<i>% set</i>
<i>[all],</i>	<i>% all</i>
<i>light_dev2,</i>	<i>% lamps</i>
<i>light_par1.</i>	<i>% half_on</i>

```
light3 ->
```

<i>light_verb,</i>	<i>% put</i>
<i>light_dev1,</i>	<i>% lamp</i>
<i>numbers,</i>	<i>% 5</i>
<i>light_par2.</i>	<i>% on</i>

```
light4 ->
```

<i>light_verb,</i>	<i>% [], empty string</i>
<i>light_dev1,</i>	<i>% lamp</i>
<i>[number],</i>	<i>% number, no option</i>
<i>numbers,</i>	<i>% 5</i>
<i>light_par2.</i>	<i>% half_on</i>

```

light5 ->
    light_verb,                % switch
    light_dev3,                % front_light
    light_par2.                % off

light6 ->
    light_verb,                % switch
    light_dev1,                % lamp
    numbers,                   % 3
    [to],                      % to, no option
    numbers.                   % 7

light7 ->
    light_verb,                % put
    light_dev1,                % lamp
    numbers,                   % 1
    [to],                      % to, no option
    lamp_intensity,           % [brightness, lev]
    numbers.                   % 8

light8 ->
    light_verb,                % switch
    light_dev1,                % lamp
    [number],                  % number, no option
    numbers,                   % 3
    [to],                      % to
    lamp_intensity,            % level
    numbers.                   % 5

light_verb -> [] ; [set] ; [put] ; [switch].

type1 -> [each] ; [every].

light_dev1 -> [light] ; [lamp].
light_dev2 -> [lights] ; [lamps].
light_dev3 -> [laser] ; [projector] ; [back_light] ; [front_light].

light_par1 -> [on] ; [off] ; [half_on] ; [to], [level], numbers.
light_par2 -> [on] ; [off] ; [half_on].

lamp_intensity -> [] ; [level] ; [brightness] ; [brightness, level] ; [intensity] ; [intensity,
    level].

```

► **Table 23.1** Lists a sample of the sentences that are accepted by this set of grammar rules.

■ Table 23.1

Some of the sentences that are accepted by the DCG rules defined in the text

Clause	Accepted string
light1	[switch, each, lamp, off]
	[switch, every, light, on]
	[set, each, lamp, off]
	[put, each, lamp, half_on]
	[every, lamp, off]
	[each, lamp, half_on]
light2	[put, all, lamps, off]
	[set, all, lights, on]
	[all, lamps, half_on]
light3	[put,lamp, 6, off]
	[switch, light, 4, half_on]
light4	[put, lamp, number, 6, off]
	[light, number, 8, on]
light5	[switch,laser,off]
	[put, back_light, on]
	[projector, off]
light6	[set, lamp, 6, to, 7]
	[light, 5, to, 9]
light7	[put, lamp,6, to, level, 7]
	[light, 5, to, brightness,4]
light8	[put, lamp, number, 6, to, intensity, level, 7]
	[set, light, number, 6, to, level, 4]

23.2.1.5 Extracting Meaning

First Method: Discarding Non-relevant Terminal Symbols

Consider the goal sequence

```

X = [put, lamp, number, 6, to, intensity, level, 7],
parse(light8,X),                % We are parsing according to "light8" rule not "light"
discard_non_numbers(X,[Y,Z]), % Y is the lamp identifier. Z is the lamp brightness.
switch_lamp(X,Y)                % Switch lamp X to state Y

```

This instantiates Y to 6 and Z to 7, so lamp 6 is switched to brightness level 7. Notice that the parser is being used simply to check that it is reasonable to apply *discard_non_numbers/2*, which is unintelligent and therefore could not be used safely on its own. Of course, a similar approach is adopted for *light1*, *light2*, ..., *light8*.

Second Method: Pattern Matching

Consider the predicate

```
meaning(light7,X,[A,B]) :- parse(light7,X), X=[_,_,A,_,_,B].
```

This tries to match X to a list containing six elements, of which only the third and sixth are retained. Matching is only allowed if X first satisfies DCG rule *light7*. Again, the parser is being used to check that it is reasonable to apply the pattern matching goal, ($X=[_,_,A,_,_,B]$).

However, the rule for *light_verb* permits both *empty* and *non-empty* strings, which causes some slight complications. (The list X can vary in length.) To accommodate all of the cases where rule *light7* holds, we require *four* separate conditions:-

```
meaning(P,X,Y) :-
    parse(light7,P),
    (P = [_,_X,_,_,Y] ; P = [_X,_,_,Y]),      % Semi-colon (;) denotes OR
    number(X).                                % Check that X is a number
```

```
meaning(P,X,Y) :-
    parse(light7,P),
    (P = [_,_X,_,_,_,Y] ; P = [_X,_,_,_,Y]).    % Semi-colon (;) denotes OR
```

In this way, Prolog can “understand” the lighting control language. (🔗 [Table 23.2](#)) The goal *meaning(A,B,C)* tries to parse the given sentence A. If the parsing succeeds, then A is analysed, so that the lamp(s) (B) and brightness level (C) can be found.

```
% Understanding sentences which are accepted by "light1"
meaning(P,all,Z) :-
    parse(light1,P),
    (P = [_,_Z] ; P = [_Z]).

% Understanding sentences which are accepted by "light2"
meaning(P,all,Z) :-
    parse(light2,P),
    (P = [_,_Z] ; P = [_Z]).

% Understanding sentences which are accepted by "light3"
meaning(P,X,Y) :-
    parse(light3,P),
    (P = [_,_X,Y] ; P = [_X,Y]).
```

■ Table 23.2
Sample queries for *meaning(A,B,C)*

A	B	C
[put,all,lamps,off]	all	off
[switch,each,lamp,half_on]	all	half_on
[set,laser,on]	laser	on
[lamp,3,to,level,2]	3	2
[switch,lamp,number,6,to,intensity,level, 5]	6	5
[switch,lamp,number,6,off]	6	off

```

% Understanding sentences which are accepted by "light4"
meaning(P,X,Y) :-
    parse(light4,P),
    (P = [_,_X,Y] ; P = [_,_X,Y]).
% Understanding sentences which are accepted by "light5"
meaning(P,X,Y) :-
    parse(light5,P),
    (P = [_X,Y] ; P = [X,Y]).
% Understanding sentences which are accepted by "light6"
meaning(P,X,Y) :-
    parse(light6,P),
    (P = [_,_X,_Y] ; P = [_X,_Y]).
% Understanding sentences which are accepted by "light7"
meaning(P,X,Y) :-
    parse(light7,P),
    (P = [_X,_Y] ; P = [_X,_Y]),
    number(X).          % Resolve ambiguity with next clause
meaning(P,X,Y) :-
    parse(light7,P),
    (P = [_X,_Y] ; P = [_X,_Y]).
% Understanding sentences which are accepted by "light8"
meaning(P,X,Y) :-
    parse(light8,P),
    (P = [_X,_Y] ; P = [_X,_Y]),
    number(X).          % Resolve ambiguity with next clause
meaning(P,X,Y) :-
    parse(light8,P),
    (P = [_X,_Y] ; P = [_X,_Y]).
% Catch-all rule for use when the sentence is not recognised
meaning(_unknown,unknown) :-
    message_to_user([' Sentence was not recognised' ]).

```

Third Method: Including Variables and Prolog Goals Within DCG Rules

Consider the following revised DCG rule:

```

light6 ->
    light_verb,           % Unchanged
    light_dev1,           % Unchanged
    Y,                    % Variable - lamp identifier
    {number(Y)},          % Prolog goal: check that Y is a number
    [to],                 % Unchanged
    Z,                    % Variable lamp brightness
    {number(Z)},          % Prolog goal: check that Z is a number
    {control_lights6([Y,Z])}. % Prolog goal must be satisfied before parsing
                                is accepted

```

This grammar rule includes three Prolog sub-goals, which must all be satisfied before the parser accepts the input list as valid. An example will illustrate this:

parse(light6,[switch, lamp, 3, to 7]).

While the parser is trying to prove that this is true, the following tests are carried out:

1. *switch* is verified as being a *light_verb*.
2. *lamp* is verified as being a *light_dev1*.
3. *Y* is instantiated to 3.
4. *Y* is checked to be a number (embedded Prolog goal).
5. *to* is found in the input list.
6. *Z* is instantiated to 7.
7. *Z* is checked to be a number (embedded Prolog goal).
8. *control_lights6(,[Y,Z])* switches lamp *Y* to brightness level *Z* (Prolog code).

Choosing Which Method to Use

A parser is able to check whether a valid sentence has been presented. This can be envisaged as acting like a filter, eliminating sentences that are non-sensible to the system. A machine can respond by indicating that it does not understand, leaving it to the user to think again about what he is saying. Shifting the responsibility for achieving an effective dialogue is acceptable within limits; people learn quicker than machines and will tolerate some restrictions, if the overall benefits are large enough. This means that naive methods for extracting meaning, such as Method 1, are safer than we might imagine. Simply deleting words that have no obvious relevance to the discourse would be dangerous without such a filter. Individual grammar rules are usually very selective. That is, safe! DCGs can be made more general in a variety of ways. Each time we add a new rule, or generalise a rule, perhaps by including variables, the system becomes a bit less secure, simply because we allow more sentences to pass through the filter. However, most DCGs are safe because they are still highly selective compared to natural unconstrained human speech. As a result, Method 1 is well worth exploring first, because it is simple to program. Method 2 is conceptually straightforward but rather cumbersome, while the third method is conceptually the most elegant. Once mastered, Method 3 is probably the most attractive because actions, such as switching lights ON/OFF, are directly embedded in the DCG rules.

23.2.1.6 Remarks

The use of DCGs for understanding a limited sub-set of natural language has considerable potential that has, as yet, not been exploited to its full potential for vision systems. As stated earlier, it is assumed that people using such a system as the one described will act reasonably and will not deliberately look for ways to confuse the program. With this in mind, it is feasible now to build a system of moderate cost that accepts input from a speech recognition device and can perform tasks such as controlling

- Lights, within a fixed array
- Simple manipulator, including
 - (X,Y,θ)-table
 - Pick-and-place arm

- Conveyor stop/start
- Accept/reject mechanism
- Filter wheel
- Pan, tilt, zoom and focus of a camera
- High-level image processing operations
- Navigation of help files, including the catalogue of lighting-viewing methods [10]
- Administration and analysis of the Machine Vision Questionnaire (MVQ)
- System initialisation, configuration and testing
- Recall of performance statistics

When designing such a speech-controlled natural language system, it is important to remember that homophones can cause confusion. For example: *rows* and *rose* sound the same and therefore must be treated as the same entity when defining a set of grammar rules. While the attractions of hands-free speech-controlled operation of industrial vision systems are obvious, there are great difficulties when trying to use DCGs to describe complex and highly variable images. Matters are made somewhat easier in the context of Automated Visual Inspection, where we know quite precisely what kind of objects and faults to expect. Nevertheless, with our present capabilities, it is impractical to employ low-level programming of vision systems (e.g., specifying individual QT commands).

23.3 Part 3: Implementation of Prolog+

23.3.1 Implementation of PQT

Using Prolog to control an image processing sub-system was conceived by David H. Mott in the mid-1980s [11]. (Also see [8, 12].) He linked Prolog and an image processing package, similar to QT, running on the same computer. Since then, several other approaches have been followed (► [Table 23.3](#)). PQT is one of the most recent implementations of Prolog+ and was constructed by interconnecting SWI-Prolog [1] and the QT image processing software described in ► [Chap. 21](#). Communication between them is achieved using “glue” programs written in Java [13]. This allows Prolog and QT to run on the same or different computers, perhaps thousands of kilometres apart. While a Prolog program is able to control several QT “slave” modules, we shall concentrate on implementing Prolog+ using just one image processing engine. This does not limit the number of the cameras that can be used, nor their location. Another interface between MATLAB and SWI-Prolog is described in [14], although this was not designed specifically for image processing.

23.3.1.1 Structure of a PQT System

► [Figure 23.9](#) shows the overall structure of a system implementing PQT and which employs the following software components

- (a) QT, implemented using MATLAB’s *Image Processing Tool-box* [15].
- (b) SWI-Prolog [1]. This is available free of charge from the University of Amsterdam.
- (c) Java interfacing (“glue”) software between QT and SWI-Prolog (boxes labelled J1, . . . , J4 in ► [Fig. 23.9](#)).

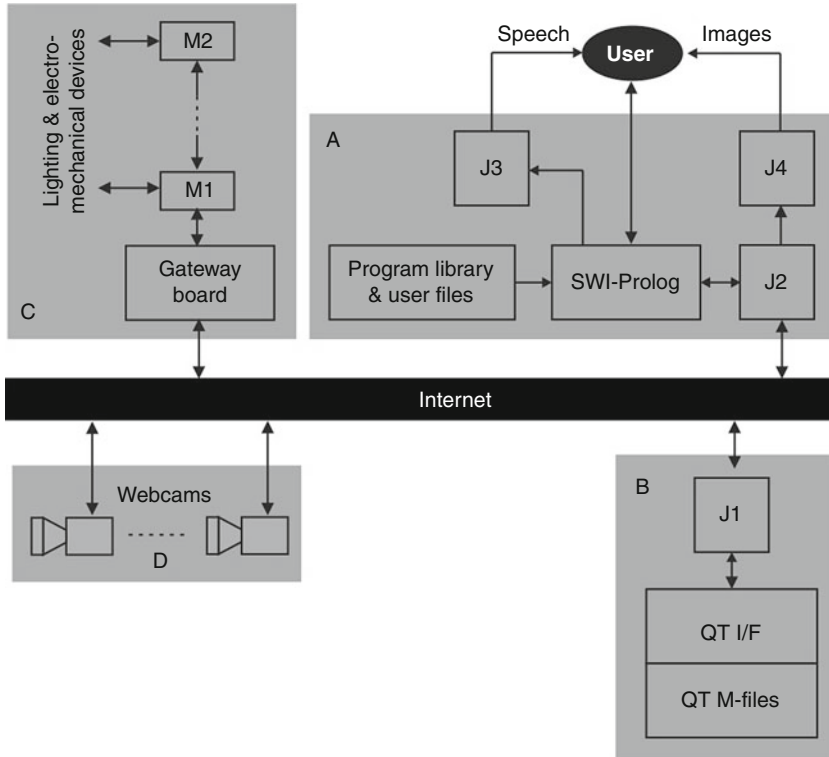
■ Table 23.3
Methods for implementing Prolog+

Prolog	Image processing	Interface	Remarks
Computer/ single-card processor	External hardware	Serial line (e.g., RS232)	Easy to build. Despite the low data rate across the Prolog-IP interface, this is surprisingly effective.
	Separate computer		
	Smart camera		Low cost. Simple to implement. IP command repertoire is fixed by the camera and may be somewhat limited compared to QT. Adequate for some of the less demanding shop-floor applications.
	External hardware (e.g., pipe-line or array processor), or plug-in card	Parallel bus	Fast enough for some shop-floor applications.
Same computer		Tight integration; direct connection between IP and Prolog programs	Can be difficult to maintain, as Prolog, the image processor and the operating system all evolve separately.
		Loose integration via operating system	Easy to build and maintain, as a standard interface is used.
Computer	Separate computer	Ethernet/Internet	Easy to build and maintain as a standard interface is used. Possible to implement multi-camera & multi-processor systems. Unpredictable delays are possible on public networks.

It is not necessary or appropriate to describe the minutiae of the Prolog-QT interface. Let it suffice to say that it involves several data-type conversions, involving MATLAB arrays and strings, Prolog strings (different delimiters), terms, atoms, lists and numbers.

The following general points should be noted:

- (i) Prolog and QT can be run on the same, or different, computers.
- (ii) Images from any web-enabled camera, or image file with a known URL, can be acquired, processed and analysed by the PQT system.
- (iii) A range of electro-mechanical devices can be operated, via a suitable device-control hardware unit. One supervisor module that the author has used is web friendly and responds to HTTP commands. It can operate up to 16 slave boards containing power switches for such devices as stepper motors, DC motors, LEDs, solenoids, etc. (Also see Appendix J.) Since the supervisor module is connected to the network, these devices could be located anywhere in the world. Effective communication can be achieved using standard wired, or wireless, Internet paths and protocols.
- (iv) Once the initialisation process has been completed, both SWI-Prolog and QT run continuously; there are no delays associated with re-starting either of them. This also means that we do not have to take any special measures to carry data from one query/command



■ Fig. 23.9

Implementing PQT using Java “glue” software. Box C contains hardware control devices. J1 – J4 are small Java programs

over to the next. (Keeping both QT and Prolog “alive”, particularly during error conditions, was one of the main factors that influenced the design of the “glue” software.)

- (v) The user interface has been enhanced by using a speech synthesiser controlled from Prolog. Since Prolog can generate natural language sentences using DCGs, it is possible to construct quite sophisticated user feed-back sequences. An earlier implementation of Prolog+ also used speech recognition hardware to provide hands-free user interaction [8]. This showed quite very clearly that speech control is potentially very useful - and is great fun!
- (vi) An infix Prolog operator (\sim) has been programmed to perform repeated operations. This allows iterated processes, such as image filtering and morphology, to be performed very easily. (This operator is listed later.)
- (vii) New functions can be written for PQT in either of two ways:
 - Adding new Prolog code
 - Adding MATLAB M-functions

23.3.1.2 Processing Prolog+ Goals

We shall now explain how Prolog goals incorporating image processing operations are processed

1. Prolog generates a command (C1) for QT. (The process of generating C1 from the original Prolog specified interactively by the user, or from a goal in a QT program, will be explained later.) C1 is expressed in the form of a Prolog string (S1) and sent via Java interface programs (J2 and J1 in [Fig. 23.7](#)) to MATLAB.
2. The command string S1 is received by a MATLAB interface M-function (*QT I/F*), which is responsible for all communication between QT and Prolog. Assuming that string S1 is correctly formatted, the appropriate command is executed by MATLAB and the results is passed back to *QT I/F*.
3. *QT I/F* standardises the data format for values that are to be returned to Prolog. First, a (MATLAB) array of size 7×1 elements is created, irrespective of how many values the QT M-function executed by S1 actually computes. Then, the array is converted to a string, S2. [Table 23.3](#) shows some examples which illustrate this. (This 7-element array allows for six values or fewer to be returned from QT, plus a variable indicating any error conditions that have occurred. Error 0 (zero) indicates no error. No current QT operator returns more than six values, although there is no fundamental reason why more values could not be calculated.)
4. String S2 is then despatched, via Java programs J1 and J2 to Prolog.
5. Program J2 reformats S2, to form a Prolog string (S3).
6. Prolog converts string S3 into a Prolog list, L1.
7. Prolog then analyses L1, to extract the values calculated by QT and ignoring the undefined (“padding”) variables. Several layers of Prolog code are required to do this. For the convenience of the user, Prolog-style operators have been implemented and are explained below.

23.3.1.3 Prolog Programs Implementing Prolog+

Step 6 in the sequence described above results in a 7-element Prolog list. The head of this list indicates how many of the numbers in its tail are valid (i.e., how many outputs the QT command S1 actually generated.) As we indicated earlier, analysing this list is made easier by using Prolog-style operators.

The `<-` Operator represents the first step towards implementing Prolog+, although it does not provide a complete solution.

Syntax

Prefix operator	<code><- command</code>
Infix operator	<code>[N,V1,V2,V3,V4,V5,V6] <- command</code>

Function

Performs the QT function *command*, if it exists

Returned values

N indicates how many of *V1,V2,V3,V4,V5,V6* are valid
Vi (*i* = 1,...,N) are the values calculated by the QT function *command*
 If N is negative an error condition has been found. MATLAB does not quit.

Example 1

Goal	<code>[N,V1,V2,V3,V4,V5,V6] <- cgr</code>
Results	N = 2; V1 and V2 are instantiated to the (X,Y)-coordinates of the centroid

Example 2

Goal $[N, V1, V2, V3, V4, V5, V6] <- \text{eul}(4)$
 Results $N = 1$; $V1$ is instantiated to the 4-neighbour Euler number

Example 3

Goal $<- \text{neg}$
 Results $[N, V1, V2, V3, V4, V5, V6] = [0, 0, 0, 0, 0, 0, 0]$

Example 4

Goal $<- \text{thr}(123, 234)$
 Results $[N, V1, V2, V3, V4, V5, V6] = [0, 0, 0, 0, 0, 0, 0]$

Example 5

Goal $[N_] <- \text{qwerty}$
 Results $N = -1$ (Error: QT command *qwerty* does not exist)

The # Operator is easier to use than $<-$ and takes us closer to Prolog+ see (● [Table 23.4](#)). # is a prefix operator and conveniently masks some quite sophisticated processing. To understand how # works, consider the Prolog goal

$\# \text{abc}(1, 2, 3, X, Y, Z).$

First, the variables are removed, leaving the Prolog term:

$\text{abc}(1, 2, 3)$

This is then converted into the following Prolog goal, which is then resolved in the manner described above:

$[N, V1, V2, V3, V4, V5, V6] <- \text{abc}(1, 2, 3).$

■ **Table 23.4**

Commands and data format used by MATLAB. The right-hand column shows the output *string* generated by the QT I/F program. This is then converted to a Prolog list that looks the same when printed. (Valid results in this list are shown in **bold-face type**; undefined padding variables are indicated by *italics*.)

Command (C1)	Operation	No. of output values calculated	String S2
neg	Negate	0	$[0, 0, 0, 0, 0, 0, 0]$
thr(123, 234)	Threshold	0	$[0, 0, 0, 0, 0, 0, 0]$
avg	Average intensity	1	$[1, \mathbf{123}, 0, 0, 0, 0, 0]$
cgr	Centroid	2	$[2, \mathbf{111}, \mathbf{222}, 0, 0, 0, 0]$
mar	Min. area rectangle	4	$[4, \mathbf{111}, \mathbf{222}, \mathbf{333}, \mathbf{444}, 0, 0]$
eul(8)	Euler number	1	$[1, \mathbf{4}, 0, 0, 0, 0, 0]$
nonsense	Not a QT command	Error!	$[-1, 0, 0, 0, 0, 0, 0]$

Let us assume that the QT command *abc(1,2,3)* has been completed satisfactorily and has returned three values, instantiating the variables in this list as follows:

```
N = 3
V1 = 444
V2 = 555
V3 = 666
```

The “padding” variables V4, V5 and V6 remain uninstantiated. The final action of the # operator is to match these variables with those specified in the original goal (C1):

```
X = V1
Y = V2
Z = V3
```

Hence, the goal succeeds with the following instantiations:

```
# abc(1,2,3,444,555,666).
```

There are three other cases to consider:

1. The command *abc(1,2,3)* returns fewer than three values. In this case, some of the variables specified in the original goal remain uninstantiated. For example, if $N = 1$ (i.e., V2–V6 have undefined values), only X is instantiated. The goal then succeeds as

```
# abc(1,2,3,123,_,_).
```

Notice that the last two arguments remain uninstantiated.
2. The command *abc(1,2,3)* returns more than three values. In this case, only the results V1, V2 and V3 are retained; V4, V5 and V6 are discarded. The goal succeeds as before

```
# abc(1,2,3,444,555,666).
```
3. If the command *abc(1,2,3)* generates a QT error, the Prolog goal C1 fails.

Completing the implementation of Prolog+ While the # operator takes us closer to implementing Prolog+, there is still a small gap remaining. To avoid the need to precede each QT command by the # symbol, it is expedient to define a library of simple rules like those following, for each QT image processing function:

```
neg :- # neg.           % Predicate neg/0 can now be used in lieu of #neg
thr(A,B) :- # thr(A,B). % Predicate thr/2 can now be used in lieu of thr(A,B)
thr(A) :- # thr(A,255). % Predicate thr/1 can now be used in lieu of #thr(A)
thr :- # thr(128,255).  % Predicate thr/0. Default values can be redefined in Prolog
cgr(X,Y) :- # cgr(X,Y). % Predicate cgr/2 can now be used in lieu of # cgr(X,Y)
cgr :- # cgr(A,Y), write(X), nl, write(Y), nl.
etc.
```

Writing rules like this for all QT commands is straightforward but rather tedious to do manually. To automate this task, the author devised a small Prolog program which builds a library of rules based on the total number of arguments (i.e., inputs plus outputs), needed for each QT function. (This “input” data was obtained from QT’s *hlp*, *HELP*, command.) Here is a small sample of the database generated automatically by the program:

```
.....
dor(V1,V2,V3,V4) :- # dor(V1,V2,V3,V4).
```

■ Table 23.5

Command formats in QT, with the <- and # operators and in PQT

QT	<- Operator (L is a 7-element list)	# Operator	Library rules	PQT (X, Y and Z are unstantiated)
neg	<- neg	# neg	neg :- # neg.	neg.
thr(12,34) thr(123) thr	<- thr(12,34) <- thr(123) <- thr	# thr(12,34) # thr(123) # thr	thr(A,B) :- # thr(A,B). thr(A) :- # thr(A). thr :- # thr(128,255).	thr(12,34). thr(123). thr.
x = avg avg	L <- avg	# avg(X)	avg(X) :- # avg(X). avg :- # avg.	avg(X). avg.
[a,b,c] = avg Fails - too many results expected	L <- avg	# avg(A,B,C)		avg(X,Y,Z).
x = eul(8) x = eul	L <- eul(4)	# eul(A,X)	eul(A,X) :- # eul(A,X). eul(A) :- # eul(8,X)	eul(8,X). eul(X).
[x,y] = cgr	L <- cgr	# cgr(X,Y)	cgr(X,Y) :- # cgr(X,Y).	cgr(X,Y).
x = cgr		# cgr(X)		cgr(X).

■ Table 23.6

Comparing the syntax of programs written in QT, PQT and using the # operator. A single PQT goal *[cbl(22)]* counts the blobs and checks that there are 22 of them. It performs the same function as *[cbl(N), N is 22]*. Similarly for *# cbl(22)*

QT	PQT	# Operator	Function
function [a,n] = analyse(b) enc thr big a = cwp; if a > 250 b = round(a/2); swi kgr(b) n = cbl; if n == 22, display('Accept') else display('Reject') end end	analyse(A,B,N) :- enc, thr, big, cwp(A), A > 250, B is A/2, swi, kgr(B), cbl(22), % Blank line here write('Accept '), nl, !. analyse_image(N) :- write('Reject'), nl.	analyse(A,B,N) :- # enc, # thr, # big, # cwp(A), A > 250, B is A/2, # swi, # kgr(B), # cbl(22), % Blank line here write('Accept '), nl, !. analyse_image(N) :- write('Reject'), nl. nl.	Enhance contrast Threshold at mid grey Isolate largest blob Count white pixels Test value of an integer Integer division Interchange images Discard small blobs Count blobs Display message Prolog: cut (!) Prolog: Display message

```

dor(V1,V2,V3) :- # dor(V1,V2,V3).
dor(V1,V2) :- # dor(V1,V2).
dor(V1) :- # dor(V1).
dor :- # dor.
dri(V1,V2) :- # dri(V1,V2).
dri(V1) :- # dri(V1).
dri :- # dri.
drp :- # drp.
....

```

The set of rules generated in this manner now forms part of the PQT library and is loaded automatically when SWI-Prolog is started.

► **Tables 23.5** compares Prolog+ notation with that of the `<-` and `#` operators, while ► **Table 23.6** compares a QT program, its Prolog+ equivalent and another using the `#` operator. However, remember that most Prolog+ programs cannot be translated directly into QT. Hence, comparisons like this should not be taken too far.

References

1. What is SWI-Prolog? University of Amsterdam, Netherlands. URL <http://www.swi-prolog.org/>. Accessed 31 Oct 2007
2. Gazdar G, Mellish C (1989) Natural language processing in prolog. Addison-Wesley, Wokingham
3. Coelho H, Cotta J (1988) Prolog by example. Springer, Berlin. ISBN 0 471 92141 6
4. Sterling L, Shapiro E (1994) The art of prolog, second edition: advanced programming techniques (logic programming). MIT Press, Cambridge. ISBN ISBN0-262-19338-8
5. Spenser C (ed) (1995) Prolog for industry Proceedings of the LPA Prolog Day RSA, Logic Programming Associates, London, ISBN 1 899754 00 8
6. Bratko I (2000) Prolog programming for artificial intelligence. Addison-Wesley, Wokingham
7. Clocksin WF, Mellish CS (2003) Programming in Prolog: using the ISO standard. Springer, Berlin
8. Batchelor BG (1991) Intelligent Image Processing in Prolog. Springer, Berlin. ISBN 978-3-540-19647-1. URL <http://www.springer.com/978-3-540-19647-1>
9. Batchelor BG (Apr 1979) Interactive image analysis as a prototyping tool for industrial inspection. Proc IEE Comput Digit Tech 2(2):61–69, Part E
10. Lighting-viewing methods, this book, Chapters 8 and 40
11. Mott DH (Oct 1985) Prolog-based image processing using Viking XA. In: Zimmerman N (ed) Proceedings of the international conference on robot vision & sensory controls. IFS, Amsterdam, pp 335–350. ISBN 0-903608-96-0
12. Bell B, Pau LF (1992) Context knowledge and search control issues in object-oriented prolog-based image understanding. Pattern Recognit Lett 13(4):279–290
13. Caton SJ (2010) Networked vision systems. Ph.D. thesis, School of Computer Science, Cardiff University, Cardiff, Wales, UK
14. Abdallah S, Rhodes C. Prolog-Matlab interface. Queen Mary College University of London. <http://www.elec.qmul.ac.uk/digitalmusic/downloads/index.html#plml>. Accessed 17 Feb 2011
15. MATLAB (2007) Image processing toolbox. Mathworks, Natick. URL <http://www.mathworks.com/>. Accessed 31 Oct 2007
16. Bell B, Pau LF (Sept 1990) Contour tracking and corner detection in a logic programming environment. PAMI 12(9):913–917