Constraint Logic Programming using ECL^{i}PS^e

Krzysztof R. Apt
and Mark G. Wallace

CAMBRIDGE

CAMBRIDGE www.cambridge.org/9780521866286

This page intentionally left blank

Constraint logic programming lies at the intersection of logic programming, optimisation and artificial intelligence. It has proved a successful tool for application in a variety of areas including production planning, transportation scheduling, numerical analysis and bioinformatics. Its migration from academic discipline to software development has been due in part to the availability of software systems that realise the underlying methodology; ECL^{i}PS^e is one of the leading such systems. It is exploited commercially by Cisco, and is freely available and used for teaching and research in over 500 universities.

This book has a two-fold purpose. It's an introduction to constraint programming, appropriate for a one-semester course for upper undergraduate or graduate students of computer science or for programmers wishing to master the practical aspects of constraint programming. By the end of the book, the reader will be able to understand and write constraint programs that solve complex problems.

Second, it provides a systematic introduction to the ECL^{i}PS^e system through carefully chosen examples that guide the reader through the language, illustrate its power and versatility, and clarify the gain achieved by this approach, which, ultimately allows the reader to better understand the proper use of constraints in solving real-world problems.

Krzysztof R. Apt received his PhD in 1974 in mathematical logic from the University of Warsaw in Poland. He is a senior researcher at Centrum voor Wiskunde en Informatica, Amsterdam and Professor of Computer Science at the University of Amsterdam. He is the author of three other books: *Verification of Sequential and Concurrent Programs* (with E.-R. Olderog), *From Logic Programming to Prolog*, and *Principles of Constraint Programming*, and has published 50 journal articles and 15 book chapters.

He is the founder and the first editor-in-chief of the ACM Transactions on Computational Logic, and past president of the Association for Logic Programming. He is a member of the Academia Europaea (Mathematics and Informatics Section).

After completing a degree at Oxford in Mathematics and Philosophy, Mark Wallace joined the UK computer company ICL, who funded his PhD at Southampton University, which was published as a book: *Communicating with Databases in Natural Language*.

He has been involved in the ECL^{i}PS^e constraint programming language since its inception and has led several industrial research collaborations exploiting the power of constraint programming with ECL^{i}PS^e. Currently he holds a chair in the Faculty of Information Technology at the Monash University, Victoria, Australia and is involved in a major new constraint programming initiative funded by National ICT Australia (NICTA), and in the foundation of a Centre for Optimisation in Melbourne. He has published widely, chaired the annual constraint programming conference, and is an editor for three international journals.

Advance praise for *Constraint Logic Programming using ECL^{i}PS^e*

The strength of *Constraint Logic Programming using ECL^{i}PS^e* is that it simply and gradually explains the relevant concepts, starting from scratch, up to the realisation of complex programs. Numerous examples and ECL^{i}PS^e programs fully demonstrate the elegance, simplicity, and usefulness of constraint logic programming and ECL^{i}PS^e.

The book is self-contained and may serve as a guide to writing constraint applications in ECL^{i}PS^e, but also in other constraint programming systems. Hence, this is an indispensable resource for graduate students, practioners, and researchers interested in problem solving and modelling.

Eric Monfroy, Université de Nantes

ECL^{i}PS^e is a flexible, powerful and highly declarative constraint logic programming platform that has evolved over the years to comprehensively support constraint programmers in their quest for the best search and optimisation algorithms. However, the absence of a book dedicated to ECL^{i}PS^e has presented those interested in this approach to programming with a significant learning hurdle. This book will greatly simplify the ECL^{i}PS^e learning process and will consequently help ECL^{i}PS^e reach a much wider community.

Within the covers of this book readers will find all the information they need to start writing sophisticated programs in ECL^{i}PS^e. The authors first introduce ECL^{i}PS^e's history, and then walk the reader through the essentials of Prolog and Constraint Programming, before going on to present the principal features of the language and its core libraries in a clear and systematic manner.

Anyone learning to use ECL^{i}PS^e or seeking a course book to support teaching constraint logic programming using the language will undoubtedly benefit from this book.

Hani El-Sakkout, Cisco Systems, Boston, Massachusetts

It has been recognized for some years now within the Operations Research community that Integer Programming is needed for its powerful algorithms, but that logic is a more flexible modelling tool. The case was made in most detail by John Hooker, in his book *Logic-Based Methods for Optimization: Combining Optimization and Constraint Satisfaction*.

The ECL^{i}PS^e system is a highly successful embodiment of these ideas. It draws on ideas coming from logic programming, constraint programming, and the Prolog language. I strongly recommend this book as a systematic account of these topics. Moreover, it gives a wealth of examples showing how to deploy the power thus made available via the ECL^{i}PS^e system.

Maarten van Emden, University of Victoria, Canada

This is an impressive introduction to Constraint Logic Programming and the ECL^{i}PS^e system by two pioneers in the theory and practice of CLP. This book represents a state-of-the-art and comprehensive coverage of the methodology of CLP. It is essential reading for new students, and an essential reference for practioners.

Joxan Jaffar, National University of Singapore

CONSTRAINT LOGIC PROGRAMMING USING ECL^{i}PS^e

KRZYSZTOF R. APT AND MARK WALLACE

CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE UNIVERSITY PRESS
Cambridge, New York, Melbourne, Madrid, Cape Town, Singapore, São Paulo

Cambridge University Press
The Edinburgh Building, Cambridge CB2 8RU, UK
Published in the United States of America by Cambridge University Press, New York
www.cambridge.org
Information on this title: www.cambridge.org/9780521866286

First published in print format 2006

ISBN-13 978-0-511-34585-2 eBook (Adobe Reader)
ISBN-10 0-511-34585-2 eBook (Adobe Reader)

ISBN-13 978-0-521-86628-6 hardback
ISBN-10 0-521-86628-6 hardback

Contents

Introduction

The subject of this book is constraint logic programming, and we will present it using the open source programming system ECL^{i}PS^e, available at `http://www.eclipse-clp.org`. This approach to programming combines two programming paradigms: logic programming and constraint programming. So to explain it we first discuss the origins of these two programming paradigms.[1]

Logic programming

Logic programming has roots in the influential approach to automated theorem proving based on the resolution method due to Alan Robinson. In his fundamental paper, Robinson [1965], he introduced the resolution principle, the notion of unification and a unification algorithm. Using his resolution method one can *prove* theorems formulated as formulas of first-order logic, so to get a 'Yes' or 'No' answer to a question. What is missing is the possibility to *compute* answers to a question.

The appropriate step to overcome this limitation was suggested by Robert Kowalski. In Kowalski [1974] he proposed a modified version of the resolution that deals with a a subset of first-order logic but allows one to generate a substitution that satisfies the original formula. This substitution can then be interpreted as a result of a computation. This approach became known as ***logic programming***. A number of other proposals aiming to achieve the same goal, viz. to compute with the first-order logic, were proposed around the same time, but logic programming turned out to be the simplest one and most versatile.

In parallel, Alain Colmerauer with his colleagues worked on a program-

[1] In what follows we refer to the final articles discussing the mentioned programming languages. This explains the small discrepancies in the dateline.

ming language for natural language processing based on automated theorem proving. This ultimately led in 1973 to creation of ***Prolog***. Kowalski and Colmerauer with his team often interacted in the period 1971–1973, which explains the relation between their contributions, see Colmerauer and Roussel [1996]. Prolog can be seen as a practical realisation of the idea of logic programming. It started as a programming language for applications in natural language processing, but soon after, thanks to contributions of several researchers, it was successfully transformed into a general purpose programming language.

Colmerauer, when experimenting with Prolog, realised some of its important limitations. For example, one could solve in it equations between terms (by means of unification) but not equations between strings. Using the current terminology, Prolog supports only one constraint solver. This led Colmerauer to design a series of successors, Prolog II, Prolog III, and Prolog IV. Each of them represents an advance in the logic programming paradigm towards constraint programming. In particular Prolog III, see Colmerauer [1990], included a support in the form of solving constraints over Booleans, reals and lists and can be viewed as the first realisation of constraint logic programming.

Constraint programming

Let us turn now our attention to ***constraint programming***. The formal concept of a ***constraint*** was used originally in physics and combinatorial optimisation. It was first adopted in computer science by Ivan Sutherland in Sutherland [1963] for describing his interactive drawing system Sketchpad. In the seventies several experimental languages were proposed that used the notion of constraints and relied on the concept of constraint solving. Also in the seventies, in the field of artificial intelligence (AI), the concept of a ***constraint satisfaction problem*** was formulated and used to describe problems in computer vision. Further, starting with Montanari [1974] and Mackworth [1977], the concept of ***constraint propagation*** was identified as a crucial way of coping with the combinatorial explosion when solving constraint satisfaction problems using top-down search.

Top-down search is a generic name for a set of search procedures in which one attempts to construct a solution by systematically trying to extend a partial solution through the addition of constraints. In the simplest case, each such constraint assigns a value to another variable. Common to most top-down search procedures is ***backtracking***, which can be traced back to the nineteenth century. In turn, the ***branch and bound search***, a

top-down search concerned with optimisation, was defined first in the context of combinatorial optimisation.

In the eighties the first constraint programming languages of importance were proposed and implemented. The most significant were the languages based on the logic programming paradigm. They involve an extension of logic programming by the notion of constraints. The main reason for the success of this approach to constraint programming is that constraints and logic programming predicates are both, mathematically, relations; backtracking is automatically available; and the variables are viewed as unknowns in the sense of algebra. The latter is in contrast to the imperative programming in which the variables are viewed as changing, but each time known entities, as in calculus.

The resulting paradigm is called ***constraint logic programming***. As mentioned above, Prolog III is an example of a programming language realising this paradigm. The term was coined in the influential paper Jaffar and Lassez [1987] that introduced the operational model and semantics for this approach and formed a basis for the CLP($\mathcal{R}$) language that provided support for solving constraints on reals, see Jaffar et al. [1992].

Another early constraint logic programming language is CHIP, see Dincbas et al. [1988] and for a book coverage Van Hentenryck [1989]. CHIP incorporated the concept of a constraint satisfaction problem into the logic programming paradigm by using constraint variables ranging over user-defined finite domains. During the computation the values of the constraint variables are not known, only their current domains. If a variable domain shrinks to one value, then that is the final value of the variable. CHIP also relied on top-down search techniques originally introduced in AI.

The language was developed at the European Computer-Industry Research Centre (ECRC) in Munich. This brings us to the next stage in our historical overview.

ECL^{i}PS^e

ECRC was set up in 1984 by three European companies to explore the development of advanced reasoning techniques applicable to practical problems. In particular three programming systems were designed and implemented. One enabled complex problems to be solved on multiprocessor hardware, and eventually on a network of machines. The second supported advanced database techniques for intelligent processing in data-intensive applications. The third system was CHIP. All three systems were built around a common foundation of logic programming.

In 1991 the three systems were merged and ECL^{i}PS^e was born. The constraint programming features of ECL^{i}PS^e were initially based on the CHIP system, which was spun out from ECRC at that time in a separate company. Over the next 15 years the constraint solvers and solver interfaces supported by ECL^{i}PS^e have been continuously extended in response to users' requirements.

The first released interface to an external state-of-the-art linear and mixed integer programming package was in 1997. The integration of the finite domain solver and linear programming solver, supporting hybrid algorithms, came in 2000. In 2001 the `ic` library was released. It supports constraints on Booleans, integers and reals and meets the important demands of practical use: it is sound, scalable, robust and orthogonal. ECL^{i}PS^{e}also includes as libraries some constraint logic programming languages, for example CLP($\mathcal{R}$), that were developed separately. By contrast with the constraint solving facilities, the parallel programming and database facilities of ECL^{i}PS^e have been much less used, and over the years some functionality has been dropped from the system.

The ECL^{i}PS^e team was involved in a number of European research projects, especially the Esprit project CHIC – Constraint Handling in Industry and Commerce (1991–1994). Since the termination of ECRC's research activities in 1996, ECL^{i}PS^e has actively been further developed at the Centre for Planning and Resource Control at Imperial College in London (IC-Parc), with funding from International Computers Ltd (ICL), the UK Engineering and Physical Sciences Research Council, and the Esprit project CHIC-2 – Creating Hybrid Algorithms for Industry and Commerce (1996–1999). The Esprit projects played an important role in focussing ECL^{i}PS^e development. In particular they emphasised the importance of the end user, and the time and skills needed to learn constraint programming and to develop large scale efficient and correct programs.

In 1999, the commercial rights to ECL^{i}PS^e were transferred to IC-Parc's spin-off company Parc Technologies, which applied ECL^{i}PS^e in its optimisation products and provided funding for its maintenance and continued development. In August 2004, Parc Technologies, and with it the ECL^{i}PS^e platform, was acquired by Cisco Systems.

ECL^{i}PS^e is in use at hundreds of institutions for teaching and research all over the world, and continues to be freely available for education and research purposes. It has been exploited in a variety of applications by academics around the world, including production planning, transportation scheduling, bioinformatics, optimisation of contracts, and many others. It is also being used to develop commercial optimisation software for Cisco.

Overview of the book

In this book we explain constraint logic programming using ECL^{i}PS^e. Since ECL^{i}PS^e extends Prolog, we explain the latter first. The reader familiar with Prolog may skip the first five chapters or treat them as a short reference manual.

In Part I of the book we focus on the logic programming paradigm. We begin by introducing in **Chapter 1** a subset of Prolog called pure Prolog and explaining the underlying computation model. In **Chapter 2** we clarify the programming features implicitly supported by pure Prolog by discussing a toy procedural programming language and explaining how pure Prolog can be translated into it.

Part II is devoted to a short exposition of Prolog. In **Chapter 3** we explain the Prolog approach to arithmetic. In particular, arithmetic constraints are allowed, but only for testing. We call constraints used in this way ***passive***. In **Chapter 4** we discuss control in Prolog. Also, we explain there how Prolog supports meta-programming, i.e., the possibility of writing programs that use other programs as data. Next, in **Chapter 5** we explain how Prolog allows us to inspect, compare and decompose terms. This exposition of Prolog is incomplete: the standard Prolog has 102 built-ins and we cover only those we need later.[2]

Part III begins with a primer on constraint programming, provided in **Chapter 6**. We introduce here the concepts of constraints, constraint satisfaction problems (CSPs) and constraint optimisation problems (COPs), and discuss the basic techniques used to solve CSPs and COPs. Then in **Chapter 7** we return to ECL^{i}PS^e by introducing a large suite of iterators that allow us to write programs without recursion. They are extensively used in the remainder of the book. Next, in **Chapter 8** we study the top-down search in presence of passive constraints by considering search algorithms for both complete and incomplete search.

One of the limitations of Prolog is that arithmetic constraints, when selected too early cause a run-time error. In **Chapter 9** we introduce the `suspend` library of ECL^{i}PS^e that allows us to circumvent this problem by automatically delaying various types of constraints, including arithmetic constraints, until they reduce to tests.

Part IV forms the main part of the book. We begin by introducing in **Chapter 10** two ECL^{i}PS^e libraries, `sd` and `ic`, that treat constraints as ***active constraints***, by allowing their evaluation to affect their variables.

[2] For those wishing to acquire the full knowledge of Prolog we recommend Bratko [2001] and Sterling and Shapiro [1994].

This is achieved through the process of ***constraint propagation***. In **Chapter 11** we explain how to combine the constraint propagation with ECL^{i}PS^e facilities that support top-down search. Next, in **Chapter 12** we discuss the `branch_and_bound` library that allows us to solve COPs using ECL^{i}PS^e.

The last two chapters deal with constraints on continuous variables. In **Chapter 13** we explain how the approach to solving CSPs and COPs based on the constraint propagation and top-down search can be modified to deal with such constraints. Finally, in **Chapter 14** we consider linear and non-linear constraints over continuous and integer variables. In ECL^{i}PS^e they are solved using the `eplex` library that provides an interface to a package for solving mixed integer programming problems.

Resources on ECL^{i}PS^e

The main website of ECL^{i}PS^e is `www.eclipse-clp.org`. It provides a substantial body of documentation to aid users in getting the most out of the system.

There are a tutorial and three manuals: a User Manual, a Reference Manual and a Constraint Library Manual. There are documents about ECL^{i}PS^e embedding and interfacing, and about applications development in ECL^{i}PS^e. Also there are example ECL^{i}PS^e programs for solving a variety of puzzles and applications in planning and scheduling. Finally, there is a great deal of on-line help available, covering all (currently 2644) built-in predicates.

The programs presented in this book can be found at `www.cwi.nl/~apt/eclipse`.

Acknowledgements

Krzysztof Apt would like to thank his colleagues who during their stay at CWI used ECL^{i}PS^e and helped him to understand it better. These are: Sebastian Brand, Sandro Etalle, Eric Monfroy and Andrea Schaerf. Also, he would like to warmly thank three people who have never used ECL^{i}PS^e but without whose support this book would never have been written. These are: Alma, Daniel and Ruth.

Mark Wallace offers this book as a testament both to the ECRC CHIP team, who set the ball rolling, to Micha Meier at ECRC, and to the IC-Parc ECL^{i}PS^e team, including Andy Cheadle, Andrew Eremin, Warwick Harvey, Stefano Novello, Andrew Sadler, Kish Shen, and Helmut Simonis – of both the CHIP and ECL^{i}PS^e fame – who have designed, built and

maintained the functionality described herein. Last but not least, he thanks Joachim Schimpf, a great colleague, innovative thinker and brilliant software engineer, who has shaped the whole ECL^{i}PS^e system.

Since dragging his reluctant family halfway round the world to Australia, Mark has buried himself in his study and left them to it. With the completion of this book he looks forward to sharing some proper family weekends with Duncan, Tessa and Toby and his long-suffering wonderful wife Ingrid.

The authors acknowledge helpful comments by Andrea Schaerf, Joachim Schimpf, Maarten van Emden and Peter Zoeteweij. Also, they would like to thank the School of Computing of the National University of Singapore for making it possible for them to meet there on three occasions and to work together on this book. The figures were kindly prepared by Ren Yuan using the `xfig` drawing program.

To Alma, Daniel and Ruth and
to Duncan, Tessa, Toby and Ingrid

Part I

Logic programming paradigm

1

Logic programming and pure Prolog

1.1 Introduction

LOGIC PROGRAMMING (LP in short) is a simple yet powerful formalism suitable for computing and for knowledge representation. It provides a formal basis for ***Prolog*** and for ***constraint logic programming***. Other successful applications of LP include ***deductive databases***, an extension of relational databases by rules, a computational approach to machine learning called ***inductive logic programming***, and a computational account of various forms of reasoning, in particular of ***non-monotonic reasoning*** and ***meta-reasoning***.

The ***logic programming paradigm*** substantially differs from other programming paradigms. The reason for this is that it has its roots in automated theorem proving, from which it borrowed the notion of a deduction. What is new is that in the process of deduction some values are computed. When stripped to the bare essentials, this paradigm can be summarised by the following three features:

- any variable can stand for a number or a string, but also a list, a tree, a record or even a procedure or program,

- during program execution variables are constrained, rather than being assigned a value and updated,
- program executions include choice points, where computation can take alternative paths; if the constraints become inconsistent, the program backtracks to the previous open choice point and tries another path.

In this chapter we discuss the logic programming framework and the corresponding small subset of Prolog, usually called ***pure Prolog***. This will allow us to set up a base over which we shall define in the successive chapters a more realistic subset of Prolog supporting in particular arithmetic and various control features. At a later stage we shall discuss various additions to Prolog provided by ECL^{i}PS^e, including libraries that support constraint programming.

We structure the chapter by focussing in turn on each of the above three items. Also we clarify the intended meaning of pure Prolog programs.[1] Consequently, we discuss in turn

- the objects of computation and their syntax,
- the meaning of pure Prolog programs,
- the accumulation of constraints during program execution, and
- the creation of choice points during program execution and backtracking.

1.2 Syntax

Syntactic conventions always play an important role in the discussion of any programming paradigm and logic programming is no exception in this matter. In this section we discuss the syntax of Prolog.

Full Prolog syntactic conventions are highly original and very powerful. Their full impact is little known outside of the logic programming community. We shall discuss these novel features in Chapters 3 and 4.

By the 'objects of computation' we mean anything that can be denoted by a Prolog variable. These are not only numbers, lists and so on, but also compound structures and even other variables or programs.

Formally, the objects of computation are ***base terms***, which consists of:

- ***variables***, denoted by strings starting with an upper case letter or '_' (the underscore), for example `X3` is a variable,
- ***numbers***, which will be dealt with in Chapter 3,

[1] To clarify the terminology: ***logic programs*** are pure Prolog programs written using the logical and not Prolog notation. In what follows we rather focus on Prolog notation and, as a result, on pure Prolog programs.

- ***atoms***, denoted by sequences of characters starting with a lower case letter, for example `x4` is an atom. Any sequence of characters put between single quotes, even admitting spaces, is also an atom, for example `'My Name'`,

and ***compound terms***, which comprise a ***functor*** and a number of ***arguments***, each of which is itself a (base or compound) ***term***.

By a ***constant*** we mean a number or an atom. Special case of terms are ***ground terms*** which are terms in which no variable appears. In general, the qualification 'ground', which we will also use for other syntactic objects, simply means 'containing no variables'.

In Chapter 4 we shall explain that thanks to the ***ambivalent syntax*** facility of Prolog programs can also be viewed as compound terms. This makes it possible to interpret ***programs as data***, which is an important feature of Prolog.

In the standard syntax for compound terms the functor is written first, followed by the arguments separated by commas, and enclosed in round brackets. For example the term with functor `f` and arguments `a`, `b` and `c` is written `f(a,b,c)`. Similarly, a more complex example with functor `h` and three arguments:

(i) the variable `A`,
(ii) the compound term `f(g,'Twenty',X)`,
(iii) the variable `X`,

is written `h(A,f(g,'Twenty',X),X)`.

Some compound terms can be also written using an infix notation. Next, we define goals, queries, clauses and programs. Here is a preliminary overview.

- A program is made up of procedures.
- A procedure is made up of clauses, each terminated by the period '.'.
- A ***clause*** is either a fact or a rule.
- There is no special procedure such as `main` and a user activates the program by means of a query.

Now we present the details.

- First we introduce an ***atomic goal*** which is the basic component from which the clauses and queries are built. It has a ***predicate*** and a number of arguments. The predicate has a ***predicate name*** and a ***predicate arity***. The predicate name is, syntactically, an atom. The arguments are placed after the predicate name, separated by commas and surrounded by round brackets. The number of arguments is the arity of the predicate.

An example of an atomic goal is `p(a,X)`. Here the predicate name is `p` and its arity is 2. We often write `p/n` for the predicate with name `p` and arity n.

- A ***query*** (or a ***goal***) is a sequence of atomic goals terminated by the period '.'. A query is called ***atomic*** if it consists of exactly one atomic goal.
- A ***rule*** comprises a ***head***, which is an atomic goal, followed by '`:-`', followed by the ***body*** which is a non-empty query, and is terminated by the period '.'.

  An example of a rule is

  ```
  p(b,Y) :- q(Y), r(Y,c).
  ```

  A rule contributes to the definition of the predicate of its head, so this example rule contributes to the definition of `p/2`. We also call this a rule for `p/2`. Rule bodies may contain some atomic goals with a binary predicate and two arguments written in the infix form, for example `X = a`, whose predicate is `=/2`.
- A ***fact*** is an atomic goal terminated by the period '.'. For example

  ```
  p(a,b).
  ```

  is a fact. This fact also contributes to the definition of `p/2`.
- A sequence of clauses for the same predicate makes a ***procedure***. The procedure provides a ***definition*** for the predicate.

  For example, here is a definition for the predicate `p/2`:

  ```
  p(a,b).
  p(b,Y) :- q(Y), r(Y,c).
  ```

- A ***pure Prolog program*** is a finite sequence of procedures, for example

  ```
  p(a,b).
  p(b,Y) :- q(Y), r(Y,c).
  q(a).
  r(a,c).
  ```

  So a program is simply a sequence of clauses.

In what follows, when discussing specific queries and rules in a running text we shall drop the final period '.'.

Before leaving this discussion of the syntax of pure Prolog, we point out a small but important feature of variable naming. If a variable appears more than once in a query (or similarly, more than once in a program clause), then

each occurrence denotes the *same* object of computation. However, Prolog also allows so-called ***anonymous variables***, written as '_' (underscore). These variables have a special interpretation, because each occurrence of '_' in a query or in a clause is interpreted as a *different* variable. That is why we talk about the anonymous variable*s* and not about the anonymous variable. So by definition each anonymous variable occurs in a query or a clause only once.

Anonymous variables form a simple and elegant device and, as we shall see, their use increases the readability of programs in a remarkable way. ECL^{i}PS^e and many modern versions of Prolog encourage the use of anonymous variables by issuing a warning if a non-anonymous variable is encountered that occurs only once in a clause. This warning can be suppressed by using a normal variable that starts with the underscore symbol '_', for example `_X`.

Prolog has several (about one hundred) built-in predicates, so predicates with a predefined meaning. The clauses, the heads of which refer to these built-in predicates, are ignored. This ensures that the built-in predicates cannot be redefined. Thus one can rely on their prescribed meaning. In ECL^{i}PS^e and several versions of Prolog a warning is issued in case an attempt at redefining a built-in predicate is encountered.

So much about Prolog syntax for a moment. We shall return to it in Chapters 3 and 4 where we shall discuss several novel and powerful features of Prolog syntax.

1.3 The meaning of a program

Pure Prolog programs can be interpreted as statements in the first-order logic. This interpretation makes it easy to understand the behaviour of a program. In particular, it will help us to understand the results of evaluating a query w.r.t. to a program. In the remainder of this section we assume that the reader is familiar with the first-order logic.

Let us start by interpreting a simple program fact, such as `p(a,b)`. It contributes to the definition of the predicate `p/2`. Its arguments are two atoms, `a` and `b`.

We interpret the predicate `p/2` as a relation symbol p of arity 2. The atoms `a` and `b` are interpreted as logical constants a and b. A logical constant denotes a value. Since, in the interpretation of pure Prolog programs, different constants denote different values, we can think of each constant denoting itself. Consequently we interpret the fact `p(a,b)` as the atomic formula $p(a, b)$.

The arguments of facts may also be variables and compound terms. Consider for example the fact `p(a,f(b))`. The interpretation of the compound term `f(b)` is a logical expression, in which the unary function f is applied to the logical constant b. Under the interpretation of pure Prolog programs, the denotations of any two distinct ground terms are themselves distinct.[2] Consequently we can think of ground terms as denoting themselves, and so we interpret the fact `p(a,f(b))` as the atomic formula $p(a, f(b))$.

The next fact has a variable argument: `p(a,Y)`. We view it as a statement that for all ground terms t the atomic formula $p(a, t)$ is true. So we interpret it as the universally quantified formula $\forall Y.\, p(a, Y)$.

With this interpretation there can be no use in writing the procedure

```
p(a,Y).
p(a,b).
```

because the second fact is already covered by the first, more general fact.

Finally we should mention that facts with no arguments are also admitted. Accordingly we can assert the fact `p`. Its logical interpretation is the proposition p.

In general, we interpret a fact by simply changing the font from `teletype` to *italic* and by preceding it by the universal quantification of all variables that appear in it.

The interpretation of a rule involves a logical ***implication***. For example the rule

```
p :- q.
```

states that if q is true then p is true.

As another example, consider the ground rule

```
p(a,b) :- q(a,f(c)), r(d).
```

Its interpretation is as follows. If $q(a, f(c))$ and $r(d)$ are both true, then $p(a, b)$ is also true, i.e., $q(a, f(c)) \wedge r(d) \rightarrow p(a, b)$.

Rules with variables need a little more thought. The rule

```
p(X) :- q.
```

states that if q is true, then $p(t)$ is true for any ground term t. So logically this rule is interpreted as $q \rightarrow \forall X.\, p(X)$. This is equivalent to the formula $\forall X.\, (q \rightarrow p(X))$.

If the variable in the head also appears in the body, the meaning is the same. The rule

[2] This will no longer hold for arithmetic expressions which will be covered in Chapter 3.

```
p(X) :- q(X).
```

states that for any ground term t, if $q(t)$ is true, then $p(t)$ is also true. Therefore logically this rule is interpreted as $\forall X.\,(q(X) \rightarrow p(X))$.

Finally, we consider rules in which variables appear in the body but not in the head, for example

```
p(a) :- q(X).
```

This rule states that if we can find a ground term t for which $q(t)$ is true, then $p(a)$ is true. Logically this rule is interpreted as $\forall X.\,(q(X) \rightarrow p(a))$, which is equivalent to the formula $(\exists X.\,q(X)) \rightarrow p(a)$.

Given an atomic goal A denote its interpretation by $\tilde{A}$. Any ground rule H `:-` $B_1, \ldots, B_n$ is interpreted as the implication $\tilde{B_1} \wedge \ldots \wedge \tilde{B_n} \rightarrow \tilde{H}$. In general, all rules H `:-` $B_1, \ldots, B_n$ have the same, uniform, logical interpretation. If $\mathbf{V}$ is the list of the variables appearing in the rule, its logical interpretation is $\forall \mathbf{V}.\,(\tilde{B_1} \wedge \ldots \wedge \tilde{B_n} \rightarrow \tilde{H})$.

This interpretation of ',' (as $\wedge$) and '`:-`' (as $\rightarrow$) leads to so-called ***declarative interpretation*** of pure Prolog programs that focusses – through their translation to the first-order logic – on their semantic meaning.

The computational interpretation of pure Prolog is usually called ***procedural interpretation***. It will be discussed in the next section. In this interpretation the comma ',' separating atomic goals in a query or in a body of a rule is interpreted as the semicolon symbol ';' of the imperative programming and '`:-`' as (essentially) the separator between the procedure header and body.

1.4 Computing with equations

We defined in Section 1.2 the computational objects over which computations of pure Prolog programs take place. The next step is to explain how variables and computational objects become constrained to be equal to each other, in the form of the answers. This is the closest that logic programming comes to assigning values to variables.

1.4.1 Querying pure Prolog programs

The computation process of pure Prolog involves a program P against which we pose a query Q. This can lead to a successful, failed or diverging computation (which of course we wish to avoid).

A successful computation yields an *answer*, which specifies constraints on the query variables under which the query is true. In this subsection we

describe how these constraints are accumulated during query processing. In the next subsection we will describe how the constraints are checked for consistency and answers are extracted from them.

The constraints accumulated by Prolog are equations whose conjunction logically entails the truth of the query. To clarify the discussion we will use the following simple program:

```
p(X) :- q(X,a).
q(Y,Y).
```

Let us first discuss the answer to the atomic query

```
q(W,a).
```

The definition of the predicate `q/2` comprises just the single fact `q(Y,Y)`. Clearly the query can only succeed if `W = a`.

Inside Prolog, however, this constraint is represented as an equation between two atomic goals: `q(W,a) = q(Y1,Y1)`. The atomic goal `q(W,a)` at the left-hand side of the equation is just the original query. For the fact `q(Y,Y)`, however, a new variable `Y1` has been introduced. This is not important for the current example, but it is necessary in general because of possible variable clashes. This complication is solved by using a different variable each time. Accordingly, our first query succeeds under the constraint `q(W,a) = q(Y1,Y1)`.

Now consider the query

```
p(a).
```

This time we need to use a rule instead of a fact. Again a new variable is introduced for each use of the rule, so this first time it becomes:

```
p(X1) :- q(X1,a).
```

To answer this query, Prolog first adds the constraint `p(a) = p(X1)`, which constrains the query to match the definition of `p/1`. Further, the query `p(a)` succeeds only if the body `q(X1,a)` succeeds, which it does under the additional constraint `q(X1,a) = q(Y1,Y1)`. The complete sequence of constraints under which the query succeeds is therefore `p(a) = p(X1), q(X1,a) = q(Y1,Y1)`. Informally we can observe that these constraints hold if all the variables take the value `a`.

Consider now the query:

```
p(b).
```

Reasoning as before, we find the query would succeed under the constraints: `p(b) = p(X1), q(X1,a) = q(Y1,Y1)`. In this case, however, there are no possible values for the variables which would satisfy these constraints. `Y1` would have to be equal both to `a` and to `b`, which is impossible. Consequently the query fails.

Next consider a non-atomic query

```
p(a), q(W,a).
```

The execution of this query proceeds in two stages. First, as we already saw, `p(a)` succeeds under the constraints `p(a) = p(X1), q(X1,a) = q(Y1,Y1)`, and secondly `q(W,a)` succeeds under the constraint `q(W,a) = q(Y2,Y2)`. The complete sequence of constraints is therefore: `p(a) = p(X1), q(X1,a) = q(Y1,Y1), q(W,a) = q(Y2,Y2)`. Informally these constraints are satisfied if all the variables take the value `a`.

A failing non-atomic query is:

```
p(W), q(W,b).
```

Indeed, this would succeed under the constraints `p(W) = p(X1), q(X1,a) = q(Y1,Y1), q(W,b) = q(Y2,Y2)`. However, for this to be satisfied `W` would have to take both the value `a` (to satisfy the first two equations) and `b` (to satisfy the last equation), so the constraints cannot be satisfied and the query fails.

Operationally, the constraints are added to the sequence during computation, and tested for consistency immediately. Thus the query

```
p(b), q(W,b).
```

fails already during the evaluation of the first atomic query, because already at this stage the accumulated constraints are inconsistent. Consequently the second atomic query is not evaluated at all.

1.4.2 Most general unifiers

The Prolog system has a built-in algorithm which can detect whether a sequence of equations between atomic goals is consistent or not. In general, two outcomes are possible. Either the algorithm returns a failure, because some atomic goals cannot be made equal (for example `p(X,a) = p(b,X)`), or it yields a positive answer in the form of a ***substitution***, which is a set of equations of the form *Variable* = *Term* such that each variable occurs at most once at a left-hand side of an equation. The resulting set of equations is called a ***unifier***.

For example, the equation `p(X,f(X,Y),c) = p(V,W,V)` has a unifier {`V = c, X = c, W = f(c,Y)`}. Also {`V = c, X = c, W = f(c,a)`} is a unifier but the first one is *more general* in a sense that can be made precise. Such a most general substitution that subsumes all others is called a ***most general unifier*** (usually abbreviated to ***mgu***). If an equation between two atomic goals has a unifier, the atomic goals are called ***unifiable***.

The problem of deciding whether a sequence of equations between atomic goals has a unifier is called the ***unification problem***. The following result was established by A. Robinson.

Unification theorem *An algorithm exists, called a* ***unification algorithm****, that for a sequence of equations between atomic goals determines whether they are consistent and if they do, it produces their mgu.*

An ***answer*** to a query is then any most general unifier of the sequence of constraints accumulated during query evaluation.

An elegant unification algorithm was introduced by M. Martelli and U. Montanari. Given a term t we denote here the set of its variables by $Var(t)$.

Martelli–Montanari Algorithm

Non-deterministically choose from the set of equations an equation of a form below and perform the associated action.

(1)	$f(s_1, ..., s_n) = f(t_1, ..., t_n)$	*replace by the equations* $s_1 = t_1, ..., s_n = t_n$,
(2)	$f(s_1, ..., s_n) = g(t_1, ..., t_m)$ where $f \not\equiv g$	*halt with failure*,
(3)	$x = x$	*delete the equation*,
(4)	$t = x$ where t is not a variable	*replace by the equation* $x = t$,
(5)	$x = t$ where $x \notin Var(t)$ and x occurs elsewhere	*perform the substitution* $\{x/t\}$ *on all other equations*
(6)	$x = t$ where $x \in Var(t)$ and $x \not\equiv t$	*halt with failure.*

The algorithm starts with the original sequence of equations and terminates when no action can be performed or when failure arises. In case of success we obtain the desired mgu.

Note that in the borderline case, when $n = 0$, action (1) includes the case $c = c$ for every constant c which leads to deletion of such an equation. In addition, action (2) includes the case of two different constants and the cases where a constant is compared with a term that is neither a constant or a variable.

To illustrate the operation of this algorithm consider some examples. To facilitate the reading the selected equations are underlined.

(i) Consider the equation

$$\texttt{p(k(Z, f(X,b,Z))) = p(k(h(X), f(g(a),Y,Z))).}$$

Applying action (1) twice yields the set

$$\{\texttt{Z} = \texttt{h(X)}, \underline{\texttt{f(X, b, Z)} = \texttt{f(g(a), Y, Z)}}\}.$$

Choosing the second equation again action (1) applies and yields

$$\{\texttt{Z} = \texttt{h(X)}, \texttt{X} = \texttt{g(a)}, \underline{\texttt{b} = \texttt{Y}}, \texttt{Z} = \texttt{Z}\}.$$

Choosing the third equation action (4) applies and yields

$$\{\texttt{Z} = \texttt{h(X)}, \texttt{X} = \texttt{g(a)}, \texttt{Y} = \texttt{b}, \underline{\texttt{Z} = \texttt{Z}}\}.$$

Now, choosing the last equation action (3) applies and yields

$$\{\texttt{Z} = \texttt{h(X)}, \underline{\texttt{X} = \texttt{g(a)}}, \texttt{Y} = \texttt{b}\}.$$

Finally, choosing the second equation action (5) applies and yields

$$\{\texttt{Z} = \texttt{h(g(a))}, \texttt{X} = \texttt{g(a)}, \texttt{Y} = \texttt{b}\}.$$

At this stage no action applies, so $\{\texttt{Z} = \texttt{h(g(a))}, \texttt{X} = \texttt{g(a)}, \texttt{Y} = \texttt{b}\}$ is an mgu of the atomic goals `p(k(Z, f(X,b,Z)))` and `p(k(h(X), f(g(a),Y,Z)))`.

(ii) Consider now the equation

$$\texttt{p(k(h(a), f(X,b,Z))) = p(k(h(b), f(g(a),Y,Z))).}$$

Again applying action (1) twice yields the set

$$\{\underline{\texttt{h(a)} = \texttt{h(b)}}, \texttt{f(X, b, Z)} = \texttt{f(g(a), Y, Z)}\}.$$

Next, choosing the first equation action (1) applies again and yields

$$\{\underline{\texttt{a} = \texttt{b}}, \texttt{f(X, b, Z)} = \texttt{f(g(a), Y, Z)}\}.$$

Choosing again the first equation action (2) applies and a failure arises. So the atomic goals `p(k(h(a), f(X,b,Z)))` and `p(k(h(b), f(g(a),Y,Z)))` are not unifiable.

(iii) Finally, consider the equation

$$\texttt{p(k(Z, f(X,b,Z))) = p(k(h(X), f(g(Z),Y,Z))).}$$

Let us try to repeat the choices made in (i). Applying action (1) twice we get the set

$$\{\mathtt{Z} = \mathtt{h(X)}, \underline{\mathtt{f(X,b,Z)} = \mathtt{f(g(Z),Y,Z)}}\}.$$

Next, choosing the second equation action (1) applies again and yields

$$\{\mathtt{Z} = \mathtt{h(X)}, \mathtt{X} = \mathtt{g(Z)}, \underline{\mathtt{b} = \mathtt{Y}}, \mathtt{Z} = \mathtt{Z}\}.$$

Choosing the third equation action (4) applies and yields

$$\{\mathtt{Z} = \mathtt{h(X)}, \mathtt{X} = \mathtt{g(Z)}, \mathtt{Y} = \mathtt{b}, \underline{\mathtt{Z} = \mathtt{Z}}\}.$$

Now, choosing the fourth equation action (3) applies and yields

$$\{\mathtt{Z} = \mathtt{h(X)}, \underline{\mathtt{X} = \mathtt{g(Z)}}, \mathtt{Y} = \mathtt{b}\}.$$

Finally, choosing the second equation action (5) applies and yields

$$\{\underline{\mathtt{Z} = \mathtt{h(g(Z))}}, \mathtt{X} = \mathtt{g(Z)}, \mathtt{Y} = \mathtt{b}\}.$$

But now choosing the first equation action (6) applies and a failure arises. Hence the atomic goals `p(k(Z, f(X,b,Z)))` and `p(k(h(X),f(g(Z),Y,Z)))` are not unifiable.

1.4.3 Choice points and backtracking

Until now we did not fully explain what happens during a program execution when an equation, for instance `q(b,a) = q(Y,Y)`, fails. When such an equation is generated during the execution of a query, backtracking is triggered, a process we first illustrate using the following simple program:

```
p(X) :- X = a.
p(X) :- X = b.
```

Each of the two rules in this program has a body consisting of an atomic goal, with predicate `=/2` written in the infix form. The behaviour of this predicate is built into the underlying implementation and will be discussed in the next section. Its logical interpretation is, of course, equality. When we pose the query

```
p(a).
```

against this program, it therefore succeeds, returning the answer "`Yes`".

Next, consider what happens when we pose the query

```
p(b).
```

According to the computation process explained in Subsection 1.4.1, using the first rule we accumulate two equations, `p(b) = p(X1)`, `X1 = a`, which are inconsistent. This might suggest that the answer to the query `p(b)` is "`No`". However, this disregards the fact that in the program the predicate `p/2` is defined by means of two clauses.

When the first equation, `p(b) = p(X1)`, is considered a ***choice point*** is created, which consists of the second clause, `p(X1) :- X1 = b`, and the current state (in which values of all the variables are recorded). Now when the failure arises during the execution that starts with the selection of the first clause, the computation returns to the choice point, the state is restored to the one stored and the second clause is selected. So all the substitutions performed since the creation of this choice point are undone. Now we accumulate the equations `p(b) = p(X1)`, `X1 = b`, which are consistent. So the answer is "`Yes`".

Finally, when we pose the query

```
p(c).
```

against this program, we get the answer "`No`" because both `X1 = a` and `X1 = b` fail under the initial condition `X1 = c`.

In general, if a predicate is defined by n clauses with $n > 1$, then upon selection of the first clause all the other clauses are kept in a choice point. When during the execution which starts by selecting the first clause a failure arises, the computation returns to the last created choice point and resumes in the state stored there but with the next clause stored in this choice point. If the last clause is selected, the choice point is removed. So when a failure arises during the execution that involves the last clause, the computation resumes at the one before last choice point, if such a choice point exists and otherwise a failure results. This computation process is called ***backtracking***.

1.5 Prolog: the first steps

1.5.1 Prolog: how to run it

It is useful at this stage to get an idea how one interacts with a Prolog system. There are two ways of running a Prolog program using ECL^{i}PS^e. The first is using `tkeclipse`, which provides an interactive graphical user interface to the ECL^{i}PS^e compiler and system. The second is a more traditional command line interface accessed by invoking just `eclipse`.

To start TkECL^{i}PS^e, either type the command `tkeclipse` at an operating system command line prompt, or select TkECL^{i}PS^e from the program menu,