

Undergraduate Topics in Computer Science

Maurizio Gabbrielli  
Simone Martini

# Programming Languages: Principles and Paradigms

*Second Edition*



 Springer

---


# Undergraduate Topics in Computer Science


## Series Editor

Ian Mackie, University of Sussex, Brighton, UK


## Advisory Editors

Samson Abramsky , Department of Computer Science, University of Oxford, Oxford, UK

Chris Hankin , Department of Computing, Imperial College London, London, UK


Mike Hinchey , Lero – The Irish Software Research Centre, University of Limerick, Limerick, Ireland

Dexter C. Kozen, Department of Computer Science, Cornell University, Ithaca, NY, USA

Andrew Pitts , Department of Computer Science and Technology, University of Cambridge, Cambridge, UK

Hanne Riis Nielson , Department of Applied Mathematics and Computer Science, Technical University of Denmark, Kongens Lyngby, Denmark

Steven S. Skiena, Department of Computer Science, Stony Brook University, Stony Brook, NY, USA

Iain Stewart , Department of Computer Science, Durham University, Durham, UK

Joseph Migga Kizza, College of Engineering and Computer Science, The University of Tennessee-Chattanooga, Chattanooga, TN, USA

‘Undergraduate Topics in Computer Science’ (UTiCS) delivers high-quality instructional content for undergraduates studying in all areas of computing and information science. From core foundational and theoretical material to final-year topics and applications, UTiCS books take a fresh, concise, and modern approach and are ideal for self-study or for a one- or two-semester course. The texts are all authored by established experts in their fields, reviewed by an international advisory board, and contain numerous examples and problems, many of which include fully worked solutions.

The UTiCS concept relies on high-quality, concise books in softback format, and generally a maximum of 275–300 pages. For undergraduate textbooks that are likely to be longer, more expository, Springer continues to offer the highly regarded Texts in Computer Science series, to which we refer potential authors.

---

Maurizio Gabbrielli · Simone Martini

# Programming Languages: Principles and Paradigms

Second Edition

With a chapter by Saverio Giallorenzo

Maurizio Gabbrielli  
Department of Computer Science  
and Engineering  
University of Bologna  
Bologna, Italy

Simone Martini  
Department of Computer Science  
and Engineering  
University of Bologna  
Bologna, Italy

ISSN 1863-7310                      ISSN 2197-1781 (electronic)  
Undergraduate Topics in Computer Science  
ISBN 978-3-031-34143-4              ISBN 978-3-031-34144-1 (eBook)  
<https://doi.org/10.1007/978-3-031-34144-1>

1<sup>st</sup> edition: © Springer-Verlag London Ltd., part of Springer Nature 2010

2<sup>nd</sup> edition: © The Editor(s) (if applicable) and The Author(s), under exclusive license to Springer  
Nature Switzerland AG 2023

This work is subject to copyright. All rights are solely and exclusively licensed by the Publisher, whether the whole or part of the material is concerned, specifically the rights of reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors, and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG  
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

*To Francesca and Antonella,  
who will never want to read this book  
but who contributed to it being written.*

*To Costanza, Maria and Teresa, who will  
read it perhaps,  
but who have done everything to stop it being  
written.*

*In the end not only Francesca and Antonella  
never read the book,  
but also Costanza, Maria, and Teresa never  
got to do it.  
This second edition is dedicated also to  
Daniela,  
hoping that at least she will read it.*

---

## Foreword to the First Edition

With great pleasure, I accepted the invitation extended to me to write these few lines of Foreword. I accepted for at least two reasons. The first is that the request came to me from two colleagues for whom I have always had the greatest regard, starting from the time when I first knew and appreciated them as students and as young researchers.

The second reason is that the text by Gabbrielli and Martini is very near to the book that I would have liked to have written but, for various reasons, never have. In particular, the approach adopted in this book is the one which I myself have followed when organizing the various courses on programming languages I have taught for almost 30 years at different levels under various titles.

The approach, summarized in two words, is that of introducing the general concepts (either using linguistic mechanisms or the implementation structures corresponding to them) in a manner that is independent of any specific language; once this is done, “real languages” are introduced. This is the only approach that allows one to reveal similarities between apparently quite different languages (and also between paradigms). At the same time, it makes the task of learning different languages easier. In my experience as a lecturer, ex-students recall the principles learned in the course even after many years; they still appreciate the approach which allowed them to adapt to technological developments without too much difficulty.

The book by Gabbrielli and Martini has, as central reference point, an undergraduate course in Computer Science. For this reason, it does not have complex prerequisites and tackles the subject by finding a perfect balance between rigor and simplicity. Particularly appreciated and successful is the force with which they illuminate the connections with other important “areas of theory” (such as formal languages, computability, semantics) which the book rightly includes (a further justification for their inclusion being that these topics are no longer taught in many degree courses).

Pisa, Italy  
Fall 2005

Giorgio Levi

---

## Preface to the Second Edition

For this second English edition, we have extensively revised the translation of the first printing. As well as correcting errata, we have updated the existing material, clarified some tricky points, and discussed newer programming languages, with Python as the main reference. Almost every page of the previous edition has changed in some way.

We also added three final chapters, on topics that we felt are too important today to be omitted, even in an undergraduate level textbook.

The first regards constraint programming. It represents the third approach seen in this book to declarative programming, which has a wide range of applications in artificial intelligence. We discuss the main ideas behind the paradigm and we illustrate the two main ways to implement it: constraint logic programs and constraint programming in the context of an imperative language. For the latter approach, we use MinZinc as a reference language. Following the guiding principle of the book, we do not want to teach constraint logic programming, MinZinc or any other specific language, but we rather provide enough background for appreciating the potential that constraint programming has in many application areas.

The second new chapter concerns concurrent programming. Although our aim remains that of an introductory textbook, the absence of any reference to concurrency seemed to us too conspicuous a gap, given that a significant part of software today exploits concurrency, from the operating system level up to the Web services. An exhaustive treatment of this topic would require (at least) a volume by itself. The chapter illustrates only the main problems that arise when switching from sequential to concurrent programs, together with the relative solutions. We tried to offer an adequate panorama of the main techniques and linguistic constructs for realizing interaction mechanisms, synchronization, and communication between concurrent programs or processes. Following the guiding principle of the whole book, we have not referred to a specific language, even if we have tried to make some notions concrete by examining the case of Java.

The third new chapter, written by Saverio Giallorenzo, in a way, rounds up the tour we started in the chapter on concurrency. We focus on a specific form of the concurrent paradigm, distributed programming, and we introduce service orientation. Distribution (as in “processes not sharing memory”) presents both challenges and opportunities. We highlight how dedicated programming language constructs



have evolved to tackle these issues and seize their strengths to support the development of open-ended, complex distributed programs through principles such as loose coupling and interoperability. We also explore the fundamental concepts of services, their traits, the related language constructs, and the challenges and support for verifying the correctness of service-oriented programs. As done in the previous chapters, we never focus on a specific language, and we maintain an open perspective. We explore how languages from this landscape implemented the same concepts, introduced different solutions to common issues, or rather evolved following different interpretations of the same ideas. In the chapter, we also provide an overview of the role of service-oriented architectures and cloud computing in the development and evolution of the service-oriented paradigm.

---

## Use of the Book

The book is primarily a university textbook, but it is also suitable for the personal education of IT specialists who want to deepen their knowledge of the mechanisms behind the languages they use. The choice of themes and the presentation style are largely influenced by the experience of teaching the content as part of the bachelor's degree in Computer Science of the School of Sciences at *Alma Mater Studiorum*—Università di Bologna.

In our experience, the book can cover two modules of six credits, placed in the second or third year of the bachelor's degree. A first module can cover abstract machines and the fundamental aspects, say Chaps. 1–9. The other can cover (some of) the different paradigms. As the maturity of students increases it also increases the amount of material one can present.

---

## Acknowledgements

M. G. and S. M. are happily indebted to Saverio Giallorenzo for his prodding to finish this second edition. Besides writing Chap. 15 on service-oriented programming, he reviewed the chapter on concurrency, revised and extended the one on history, and made many more valuable contributions to the text of the whole book.

We would like to thank the people who have kindly helped us with the new chapters included in the second edition. Roberto Amadini read a draft of the chapter on constraint programming and provided useful insights and comments. Fabrizio Montesi and Florian Rademacher read drafts of the chapter on service orientation and helped to improve it with their generous feedback, insights, and errata.

Bologna, Italy  
Spring 2023

Maurizio Gabbrielli  
Simone Martini  
Saverio Giallorenzo

---

## Preface to the First Edition

*Facilius per partes in cognitionem totius adducimur*  
(Seneca, *Epist.* 89, 1)

Learning a programming language, for most students in computing, is akin to a rite of passage. It is an important transition, soon recognized as insufficient. Among the tools of the trade, there are many languages, so an important skill for a good IT specialist is to know how to move from one language to another (and how to learn new ones) with naturalness and speed.

This competence is not obtained by learning many different languages from scratch. Programming languages, like natural languages, have their similarities, analogies, cross-pollination phenomena, and genealogies that influence their characteristics. If it is impossible to learn tens of languages well, it is possible to deeply understand the mechanisms that inspire and guide the design and implementation of hundreds of different languages. This knowledge of the “parts” eases the understanding of the “whole” of a new language and, therefore, underpins a fundamental methodological competence in the professional life of IT specialists, at least as far as it allows them to anticipate innovations and outlive technologies that grow obsolete.

It is for these reasons that a course on the general aspects of programming languages is, throughout the world, a key step in the advanced education (university or professional) of IT specialists. The fundamental competences which an IT specialist must possess about programming languages are of at least four types:

- the proper linguistic aspects;
- how one can implement language constructs and what are the costs relative to a given implementation;
- the architectural aspects that influence implementation;
- translation techniques (compilation).

A single course rarely deals with all four of these aspects. In particular, description of the architectural aspects and compilation techniques are both topics that are sufficiently complex and elaborate to deserve independent courses. The remaining

aspects are primarily the content of a general course on programming languages and comprise the main subject of this book.

The literature is rich in texts dealing with these subjects and generations of students have used them in their education. All these texts, though, have in mind an advanced reader who already understands many programming languages, who has a more-than-superficial competence with the fundamental basic mechanisms, and who is not afraid when confronted with a fragment of code written in an unknown language (because they can understand it by looking at the analogies and differences with those they already know). We can see these textbooks as focused on the comparison among languages. These are long, deep, and stimulating books, but they are *too* long and deep (read: difficult) for the student who begins their career with a single programming language (or at most two) and who still has to learn the basic concepts in detail.

This book aims to fill this gap. Experts will see that the index of topics pursues in large measure classical themes. However, we treat these themes in an elementary fashion, trying to assume as prerequisites only the bare minimum. We also strive to not make the book a catalogue of the traits and differences of the existing programming languages. The ideal (or reference) reader knows one language (well) (for example, Pascal, C, C++, or Java); even better if they have had some exposure to another language or paradigm. We avoided references to languages that are now obsolete, and we rarely show code examples written in a specific programming language. Mainly, the text freely uses a sort of pseudo-language (whose concrete syntax was inspired by C and Java) and seeks, in this way, to describe the most relevant aspects of different languages.

Every so often, a “box” at the top of the page presents an in-depth discussion, a reminder of a basic concept or specific details about popular languages (C, C++, Java; ML, and LISP for functional languages; PROLOG for logic programming languages). The reader at the first reading can safely skip the material in boxes.

Every chapter contains a short sequence of exercises, intended as a way of testing the understanding of the material. There are no truly difficult exercises or requiring more than 10 min for their solution.

Chapter 3 (Foundations) deals with themes that are not usually present in a book on programming languages. It is natural, however, while discussing static semantics and comparing languages, to ask what are the limits of the static analysis of programs and whether one can apply the same techniques to different languages. Rather than pointing the reader to other textbooks, given the conceptual and pragmatic relevance of these questions, we decided to answer them directly. In the space of a few pages, we present, in an informal but rigorous manner, the undecidability of the halting problem and that all general-purpose programming languages express the same class of computable functions. This allows us to present to the student, who does not always have a complete course on the “fundamentals” in their curriculum, the main results related to the limitations of computational models, which we deem fundamental for their education.

Besides the principles, this book also introduces the main *programming paradigms*: object-oriented, functional and logic. The need to write an introductory

book and keep it within a reasonable length explains the exclusion of important topics, such as concurrency and scripting languages for example.

---

## Acknowledgements

Our thanks to Giorgio Levi go beyond the fact that he had the grace to write the Foreword. Both of us owe to him our first understanding of the mechanisms that underpin programming languages. His teaching appears in this book in a way that is anything but marginal.

We have to thank the many people who kindly helped us in the realization of the book. Ugo Dal Lago drew the figures using METAPOST and Cinzia Di Giusto, Wilmer Ricciotti, Francesco Spegni, and Paolo Tacchella read and commented on the drafts of some chapters. The following people pointed out misprints and errors: Irene Borra, Ferdinanda Camporesi, Marco Comini, Michele Filannino, Matteo Friscini, Stefano Gardenghi, Guido Guizzunti, Giacomo Magisano, Flavio Marchi, Fabrizio Massei, Jacopo Mauro, Maurizio Molle, Mirko Orlandelli, Marco Pedicini, Andrea Rappini, Andrea Regoli, Fabiano Ridolfi, Giovanni Rosignoli, Giampiero Travaglini, and Fabrizio Giuseppe Ventola.

We gladly acknowledge the support of the Department of Computer Science of Università di Bologna towards the English translation of the first edition of the book.

Bologna, Italy  
Fall 2005

Maurizio Gabbrielli  
Simone Martini

---

# Contents

<b>1</b>	<b>Abstract Machines</b>	<b>1</b>
1.1	The Concepts of Abstract Machine and the Interpreter	1
1.1.1	The Interpreter	2
1.1.2	An Example of an Abstract Machine: The Hardware Machine	5
1.2	Implementation of a Language	9
1.2.1	Implementation of an Abstract Machine	9
1.2.2	Implementation: The Ideal Case	11
1.2.3	Implementation: The Real Case and the Intermediate Machine	16
1.3	Hierarchies of Abstract Machines	20
1.4	Summary	22
1.5	Bibliographical Notes	23
1.6	Exercises	23
	References	24
<b>2</b>	<b>Describing a Programming Language</b>	<b>25</b>
2.1	Levels of Description	25
2.2	Grammar and Syntax	27
2.2.1	Context-Free Grammars	28
2.3	Contextual Syntactic Constraints	37
2.4	Compilers	39
2.5	Semantics	43
2.6	Pragmatics	50
2.7	Implementation	50
2.8	Summary	50
2.9	Bibliographical Notes	51
2.10	Exercises	51
	References	52
<b>3</b>	<b>Foundations</b>	<b>53</b>
3.1	The Halting Problem	53
3.2	Undecidable Problems	55
3.2.1	The Standard Model	56
3.2.2	More Undecidable Problems	57

3.3	Formalisms for Computability .....	57
3.4	There Are More Functions Than Algorithms .....	59
3.5	Summary .....	60
3.6	Bibliographical Notes .....	61
3.7	Exercises .....	61
	References .....	62
<b>4</b>	<b>Names and the Environment .....</b>	<b>63</b>
4.1	Names and Denotable Objects .....	63
4.1.1	Denotable Objects .....	65
4.2	Environments and Blocks .....	66
4.2.1	Blocks .....	67
4.2.2	Types of Environment .....	68
4.2.3	Operations on Environments .....	71
4.3	Scope Rules .....	73
4.3.1	Static Scope .....	74
4.3.2	Dynamic Scope .....	76
4.3.3	Some Scope Problems .....	78
4.4	Summary .....	80
4.5	Bibliographical Notes .....	81
4.6	Exercises .....	81
	References .....	85
<b>5</b>	<b>Memory Management .....</b>	<b>87</b>
5.1	Techniques for Memory Management .....	87
5.2	Static Memory Management .....	89
5.3	Dynamic Memory Management Using Stacks .....	90
5.3.1	Activation Records for In-Line Blocks .....	92
5.3.2	Activation Records for Procedures .....	93
5.3.3	Stack Management .....	96
5.4	Dynamic Management Using a Heap .....	97
5.4.1	Fixed-Length Blocks .....	98
5.4.2	Variable-Length Blocks .....	99
5.5	Implementation of Scope Rules .....	102
5.5.1	Static Scope: The Static Chain .....	102
5.5.2	Static Scope: The Display .....	107
5.5.3	Dynamic Scope: Association Lists and CRT .....	109
5.6	Summary .....	113
5.7	Bibliographical Notes .....	114
5.8	Exercises .....	114
	References .....	116
<b>6</b>	<b>Control Structure .....</b>	<b>117</b>
6.1	Expressions .....	118
6.1.1	Expression Syntax .....	118
6.1.2	Semantics of Expressions .....	120
6.1.3	Evaluation of Expressions .....	123

6.2	The Concept of Command .....	127
6.2.1	The Variable .....	128
6.2.2	Assignment .....	129
6.3	Sequence Control Commands .....	134
6.3.1	Commands for Explicit Sequence Control .....	135
6.3.2	Conditional Commands .....	137
6.3.3	Iterative Commands .....	141
6.4	Structured Programming .....	148
6.5	Recursion .....	151
6.5.1	Tail Recursion .....	153
6.5.2	Recursion or Iteration? .....	157
6.6	Summary .....	158
6.7	Bibliographical Notes .....	159
6.8	Exercises .....	159
	References .....	161
<b>7</b>	<b>Control Abstraction</b> .....	163
7.1	Subprograms .....	164
7.1.1	Functional Abstraction .....	166
7.1.2	Parameter Passing .....	167
7.2	Higher-Order Functions .....	176
7.2.1	Functions as Parameters .....	177
7.2.2	Functions as Results .....	182
7.2.3	Anonymous Functions: Lambda Expressions .....	185
7.3	Exceptions .....	187
7.3.1	Implementing Exceptions .....	192
7.4	Summary .....	193
7.5	Bibliographical Notes .....	194
7.6	Exercises .....	195
	References .....	197
<b>8</b>	<b>Structuring Data</b> .....	199
8.1	Data Types .....	199
8.1.1	Types as Support for Conceptual Organisation .....	200
8.1.2	Types for Correctness .....	201
8.1.3	Types and Implementation .....	202
8.2	Type Systems .....	203
8.2.1	Static and Dynamic Checking .....	204
8.3	Scalar Types .....	205
8.3.1	Booleans .....	206
8.3.2	Characters .....	207
8.3.3	Integers .....	207
8.3.4	Reals .....	207
8.3.5	Fixed Point .....	208
8.3.6	Complex .....	209
8.3.7	Unit .....	209

8.3.8	Enumerations .....	210
8.3.9	Intervals .....	210
8.3.10	Ordered Types .....	211
8.4	Composite Types .....	211
8.4.1	Records .....	211
8.4.2	Unions .....	213
8.4.3	Tagged Unions .....	215
8.4.4	Arrays .....	216
8.4.5	Sets .....	221
8.4.6	Pointers .....	222
8.4.7	Sequences .....	226
8.4.8	Recursive Types .....	228
8.4.9	Functions .....	230
8.5	Equivalence .....	231
8.5.1	Equivalence by Name .....	232
8.5.2	Structural Equivalence .....	232
8.6	Compatibility and Conversion .....	234
8.7	Polymorphism .....	238
8.7.1	Overloading .....	239
8.7.2	Universal Parametric Polymorphism .....	240
8.7.3	Subtype Universal Polymorphism .....	242
8.7.4	Remarks on the Implementation .....	243
8.8	Type Checking and Inference .....	245
8.9	Safety: An Assessment .....	247
8.10	Avoiding Dangling References .....	248
8.10.1	Tombstone .....	248
8.10.2	Locks and Keys .....	250
8.11	Garbage Collection .....	251
8.11.1	Reference Counting .....	252
8.11.2	Mark and Sweep .....	254
8.11.3	Interlude: Pointer Reversal .....	255
8.11.4	Mark and Compact .....	256
8.11.5	Copy .....	257
8.12	Summary .....	260
8.13	Bibliographical Notes .....	260
8.14	Exercises .....	261
	References .....	264
<b>9</b>	<b>Data Abstraction .....</b>	<b>267</b>
9.1	Abstract Data Types .....	268
9.2	Information Hiding .....	270
9.2.1	Representation Independence .....	273
9.3	Modules .....	274
9.4	Summary .....	276
9.5	Bibliographical Notes .....	276



---

9.6	Exercises .....	278
	References .....	278
<b>10</b>	<b>Object-Oriented Paradigm .....</b>	<b>279</b>
10.1	The Limits of Abstract Data Types .....	280
10.1.1	A First Review .....	283
10.2	Fundamental Concepts .....	283
10.2.1	Objects .....	284
10.2.2	Classes .....	286
10.2.3	Encapsulation .....	289
10.2.4	Subtypes .....	289
10.2.5	Inheritance .....	295
10.2.6	Dynamic Method Lookup .....	300
10.3	Implementation Aspects .....	305
10.3.1	Single Inheritance .....	306
10.3.2	The Problem of Fragile Base Class .....	308
10.3.3	Dynamic Method Dispatch in the JVM .....	309
10.3.4	Multiple Inheritance .....	312
10.4	Polymorphism and Generics .....	318
10.4.1	Subtype Polymorphism .....	318
10.4.2	Generics in Java .....	320
10.4.3	Implementation of Generics in Java .....	325
10.4.4	Generics, Arrays and Subtype Hierarchy .....	325
10.4.5	Covariant and Contravariant Overriding .....	327
10.5	Summary .....	330
10.6	Bibliographical Notes .....	331
10.7	Exercises .....	331
	References .....	334
<b>11</b>	<b>Functional Programming Paradigm .....</b>	<b>335</b>
11.1	Computing Without State .....	335
11.1.1	Expressions and Functions .....	336
11.1.2	Computation as Reduction .....	338
11.1.3	The Fundamental Ingredients .....	340
11.2	Evaluation .....	341
11.2.1	Values .....	341
11.2.2	Capture-Free Substitution .....	342
11.2.3	Evaluation Strategies .....	342
11.2.4	Comparison of the Strategies .....	345
11.3	Programming in a Functional Language .....	347
11.3.1	Local Environment .....	347
11.3.2	Interactiveness .....	347
11.3.3	Pattern Matching .....	348
11.3.4	Programming in a Functional Style .....	349
11.3.5	Types .....	350
11.3.6	Infinite Objects .....	352

11.3.7	Imperative Aspects .....	353
11.4	An Assessment .....	355
11.5	Implementation: The SECD Machine .....	356
11.6	Fundamentals: The $\lambda$ -Calculus .....	359
11.7	Summary .....	365
11.8	Bibliographical Note .....	366
11.9	Exercises .....	366
	References .....	367
<b>12</b>	<b>Logic Programming Paradigm .....</b>	<b>369</b>
12.1	Deduction as Computation .....	369
12.1.1	An Example .....	370
12.2	Syntax .....	373
12.2.1	The Language of First-Order Logic .....	373
12.2.2	Logic Programs .....	375
12.3	Theory of Unification .....	377
12.3.1	The Logic Variable .....	377
12.3.2	Substitution .....	378
12.3.3	Most General Unifier .....	381
12.3.4	A Unification Algorithm .....	383
12.4	The Computational Model .....	386
12.4.1	The Herbrand Universe .....	386
12.4.2	Declarative and Procedural Interpretation .....	387
12.4.3	Procedure Calls .....	388
12.4.4	Control: Non-determinism .....	392
12.4.5	Some Examples .....	394
12.5	Extensions .....	397
12.5.1	Prolog .....	397
12.5.2	Logic Programming and Databases .....	402
12.6	Advantages and Disadvantages of the Logic Paradigm .....	403
12.7	Summary .....	404
12.8	Bibliographical Notes .....	405
12.9	Exercises .....	406
	References .....	407
<b>13</b>	<b>Constraint Programming Paradigm .....</b>	<b>409</b>
13.1	Constraint Programming .....	409
13.1.1	Types of Problems .....	411
13.2	Constraint Logic Programs .....	413
13.2.1	Syntax and Semantics of CLP .....	414
13.3	Generate and Test Versus Constraint and Generate .....	417
13.3.1	A Further Example .....	420
13.4	MiniZinc .....	422
13.4.1	A MiniZinc CSP Model .....	422
13.4.2	A MiniZinc COP Model .....	425
13.5	Summary .....	427

13.6	Bibliographical Notes .....	427
13.7	Exercises .....	428
	References .....	431
<b>14</b>	<b>Concurrent Programming .....</b>	<b>433</b>
14.1	Threads and Processes .....	433
14.2	A Brief Historical Overview .....	434
14.3	Types of Concurrent Programming .....	436
14.3.1	Communication Mechanisms .....	438
14.3.2	Synchronisation Mechanisms .....	440
14.4	Shared Memory .....	441
14.4.1	Busy Waiting .....	442
14.4.2	Scheduler-Based Synchronisation .....	445
14.5	Message Exchange .....	451
14.5.1	Naming Mechanisms .....	451
14.5.2	Asynchronous Communication .....	453
14.5.3	Synchronous Communication .....	456
14.5.4	Remote Procedure Call and Rendez-Vous .....	457
14.6	Non-determinism and Parallel Composition .....	459
14.6.1	Parallel Composition .....	462
14.7	Concurrency in Java .....	463
14.7.1	Creation of Threads .....	463
14.7.2	Scheduling and Termination of Threads .....	464
14.7.3	Synchronisation and Communication Between Threads .....	466
14.8	Summary .....	469
14.9	Bibliographical Notes .....	469
14.10	Exercises .....	470
	References .....	471
<b>15</b>	<b>Service-Oriented Programming Paradigm .....</b>	<b>473</b>
15.1	Towards Services .....	473
15.1.1	Distributed Programs .....	473
15.1.2	Open Systems .....	474
15.1.3	Loose Coupling and Interoperability .....	475
15.1.4	Approaching Services .....	476
15.2	Elements of Service-Oriented Programming .....	477
15.2.1	Services, a Layered View .....	477
15.2.2	Data Types .....	481
15.2.3	Messaging Patterns .....	487
15.2.4	Operations and Interfaces .....	489
15.2.5	Behaviour .....	494
15.3	Service-Oriented Architectures .....	509
15.3.1	The First Generation: Web Services .....	510
15.3.2	Second Generation: Microservices and Serverless ....	513
15.4	Summary .....	515

---

15.5	Bibliographical Note .....	515
15.6	Exercises .....	516
	References .....	517
<b>16</b>	<b>Short Historical Perspective .....</b>	<b>519</b>
16.1	Beginnings .....	519
16.2	Factors in the Development of Languages .....	522
16.3	1950s and 1960s .....	523
16.4	1970s .....	527
16.5	1980s .....	532
16.6	1990s .....	536
16.7	2000s .....	542
16.8	2010s .....	544
16.9	Summary .....	547
16.10	Bibliographical Notes .....	548
	References .....	548
<b>Index</b>	.....	<b>551</b>

## 12.1 Deduction as Computation

A famous slogan by R. Kowalski, accurately captures the concepts underpinning the activity of programming:  $\text{Algorithm} = \text{Logic} + \text{Control}$ . According to this “equation”, the specification of an algorithm, and therefore its formulation in programming languages, can be separated into two parts. On the one side, the logic of the solution is specified—that is, the “what” must be done is defined. On the other, those aspects related to control are specified, and therefore the “how” of finding the desired solution is clarified. The programmer who uses a traditional imperative language must take account of both these components. Logic programming, on the other hand, was originally defined with the idea of separating these two aspects. The programmer is only required, at least in principle, to provide a logical specification. Everything related to control is delegated to the abstract machine. Using a computational mechanism based on a particular deduction rule (resolution), the interpreter searches through the space of possible solutions for the one specified by the “logic”, defining in this way the sequence of operations necessary to reach the final result.

The basis for this view of computation as logical deduction can be traced back to the work of K. Gödel and J. Herbrand in the 1930s. In particular, Herbrand anticipated, albeit incompletely, some ideas about the process of unification, which, as we shall see, is the basic computational mechanism of logic programming languages.

It was not until the 1960s that a formal definition (due to A. Robinson) of this process appeared. Then, ten years later (in the early 1970s) it was realised that formal automatic deduction of a particular kind could be interpreted as a computational mechanism. The first programming languages of this paradigm were created, among which PROLOG (the name is an acronym for PROgramming in LOGic; for more information on history, see Sect. 16.4).

Today there are many implemented versions of PROLOG, and there are several other languages in this paradigm (as far as applications are concerned, those including constraints are of particular interest). All these languages (including PROLOG) also

provide constructs for the explicit specification of control, for the sake of efficiency. Since these constructs do not have a direct logical interpretation, they make the semantics of the language more complicated and cause the loss of part of the purely declarative nature of the logic paradigm. This notwithstanding, we still call them logic programming languages—they require the programmer to do little more than formulate (or declare) the specification of the problem to be solved. In some cases, the resulting programs are really surprising in their brevity, simplicity and clarity, as we will see in the next section.

### Terminological Note

We should distinguish between Logic programming, which is the theoretical foundational formalism, from PROLOG, a language that has different implementations. The concepts we introduce below, if not otherwise stated, are valid for both. The important differences will be explicitly pointed out. The programming examples which we include (usually in `this` font) are all PROLOG code that could be run on some implementation of the language. The theoretical concepts, on the other hand, even when they are valid also for PROLOG, use mathematical concepts and are rendered in *italic*. Moreover, we will follow the PROLOG convention that we always write variables beginning with an upper-case letter. As mentioned before, in the logic programming paradigm there exist several other languages, different from PROLOG. We call all of them logic programming languages, or logic languages, for short.

#### 12.1.1 An Example

We start with a fairly informal example since we did not yet introduce either the syntax or the semantics of logic languages.

Let us consider the following problem. We want to arrange three 1s, three 2s, . . . , three 9s in a list (which therefore will consist of 27 numbers) such that, for each  $i \in [1, 9]$ , there are exactly  $i$  numbers between two successive occurrences of the number  $i$ . Therefore, 1, 2, 1, 8, 2, 4, 6, 2 could be a part of the final solution, while 1, 2, 1, 8, 2, 4, 2, 6 is not (because there is only a single number between the last two occurrences of 2). The reader is invited to try to write a program that solves this problem using her preferred imperative programming language. It is not a difficult exercise but does require some care, because even if the “what” must be done is clear, the “how” of the desired solution is not trivial. For example, in a naive (and inefficient) way, we may generate all possible permutations of a list of 27 numbers containing three 1s, three 2s, three 3s, . . . , three 9s, to see if one of them satisfies the required propriety. Even this solution (probably the conceptually simplest one), however, requires the specification in detail of the aspects of control to generate the permutations of a list.

Reasoning in a declarative fashion, on the other hand, we can proceed as follows. First, we need a list which we will call `LS`. This list will have to contain 27 elements,

something that we can specify using a unary predicate (that is, a relation symbol)<sup>1</sup> `list_of_27`. If we write `list_of_27(Ls)`, we mean, therefore, that `Ls` must be a list of 27 elements. In other words, `list_of_27` defines a unary relation (a subset of all possible finite strings) formed by all the lists of 27 elements. For this, we can define `list_of_27` as follows:

```
list_of_27(Ls) :-
    Ls = [_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_].
```

where the part to the left of the `:-` symbol denotes what is being defined, while the part to the right of `:-` indicates the definition. The `=` denotes an equality concept whose definition, for the time being, is left to intuition (it has nothing to do with an assignment, however). Anticipating the PROLOG notation for lists, here we write `[X1, X2, ..., Xn]` to denote the list that contains the  $n$  variables `X1`, `X2`, ..., `Xn` (see Sect. 12.4.5 for more details about lists). We use `_` for the *anonymous* variable, that is a variable whose name is of no interest and which is distinct from all other variables present, including all other anonymous variables.

In order to satisfy our specification, the list, `Ls`, in addition to being composed of 27 elements, must contain a sublist<sup>2</sup> in which the number 1 appears followed by any other number, by another occurrence of the number 1, then another number and finally a last occurrence of the number 1. Such a sublist can be specified as

```
[1, X, 1, Y, 1]
```

where `X` and `Y` are variables. Exploiting again the anonymous variable, we may write:

```
[1, _, 1, _, 1]
```

Assuming that we have available a binary predicate `sublist(X, Y)` whose meaning is that the first argument (`X`) is a sublist of the second (`Y`),<sup>3</sup> our requirement therefore can be expressed by writing:

```
sublist([1, _, 1, _, 1], Ls)
```

Moving on to number 2 and reasoning in a similar way, we obtain that the list `Ls` must also contain the sublist:

```
[2, _, _, 2, _, _]
```

and therefore the following must also be true:

```
sublist([2, _, _, 2, _, _], Ls)
```

Repeating this reasoning for all the other numbers up to 9, the `sol` program that we want to produce can be described as follows:

<sup>1</sup> Recall that unary means that it has a single argument while  $n$ -ary means that there are  $n$  arguments.

<sup>2</sup> Recall that `Li` is a sublist of `Ls` if `Ls` is obtained by concatenating a (possibly empty) list with `Li` and with another list (itself also possibly empty).

<sup>3</sup> The definition of the relation `sublist` is given in Sect. 12.4.5.





1, 9, 1, 2, 1, 8, 2, 4, 6, 2, 7, 9, 4, 5, 8, 6, 3, 4, 7, 5, 3, 9, 6, 8, 3, 5, 7.

Successive calls to the same procedure allow obtaining the other solutions to the problem.

With this procedural interpretation, one has a true programming language that allows us to express, in a compact and relatively simple way, programs that solve even complex problems. For this power, the language pays the penalty of efficiency. In the preceding program, despite its apparent simplicity, the computation performed by the language's abstract machine is complex, given that the interpreter must try the various combinations of possible sublists until it finds the one that satisfies all the given conditions. In these search processes, a *backtracking* mechanism is used. When the computation reaches at a point at which it cannot proceed, the computation that has been performed so far is undone so that a decision point can be reached, if it exists, at which an alternative is chosen that is different from the previous one (if this alternative does not exist, the computation terminates in failure). It is not difficult to see that, in general, this search process can have exponential complexity.

---

## 12.2 Syntax

Logic programs are sets of logic formulæ of a particular form. We begin therefore with some basic notions for defining the syntax.

The logic of interest here is first-order logic, also called *predicate calculus*. Symbols are used to express properties of (or, in a more old-fashioned terminology, to “predicate on”) elements of a fixed domain of discourse,  $\mathcal{D}$ . Higher-order logics (second, third, etc., order) also permit predicates whose arguments are more complicated objects such as sets and functions over  $\mathcal{D}$  (second order), sets of these functions (third order), etc., in addition to elements of  $\mathcal{D}$ .

### 12.2.1 The Language of First-Order Logic

The language of first-order logic consists of three components:

1. An *alphabet*.
2. *Terms* defined over this alphabet.
3. *Well-formed formulæ* defined over this alphabet.

#### Alphabet

The alphabet is a set<sup>5</sup> of symbols, partitioned into two disjoint subsets: the set of *logical symbols* (common to all first-order languages) and the set of *non-logical* (or *extra-logical*) symbols (which are specific to a domain of interest). For example, all

---

<sup>5</sup> All the sets we consider here are finite or denumerable.

first-order languages will use a (logical) symbol to denote conjunction. If we are considering orderings on a set, we will probably also have the  $<$  symbol as one of the non-logical symbols.

The set of *logical symbols* contains the following elements:

- The logical connectives  $\wedge$  (conjunction),  $\vee$  (disjunction),  $\neg$  (negation),  $\rightarrow$  (implication) and  $\leftrightarrow$  (double implication).
- The propositional constants *true* and *false*.
- The quantifiers  $\exists$  (exists) and  $\forall$  (forall).
- Some punctuation symbols such as brackets “(” and “)” and comma “,”.
- An (denumerably) infinite set  $V$  of *variables*, written  $X, Y, Z, \dots$

The *non-logical* (or *extra-logical*) symbols are defined by a *signature with predicates*  $(\Sigma, \Pi)$ . This is a pair in which the first element,  $\Sigma$  is the *function signature*, that is a set of function symbols, each considered with its own arity.<sup>6</sup> The second element of the pair,  $\Pi$ , is the *predicate signature*, a set of predicate symbols together with their arities. Functions of arity 0 are said to be *constants* and are denoted by the letters  $a, b, c, \dots$ . Function symbols of positive arity are, as usual, denoted by  $f, g, h, \dots$ , while predicate symbols are denoted by  $p, q, r, \dots$ . Let us assume that the sets  $\Sigma$  and  $\Pi$  have an empty intersection and are also disjoint from the logical symbols. The difference between function and predicate symbols is that the former must be interpreted as functions, while the latter must be interpreted as relations. This distinction will become clearer when we discuss formulæ.

## Terms

The concept of term, which is fundamental to mathematical logic and Computer Science, is used implicitly in many contexts. For example, an arithmetic expression is a term obtained by applying (arithmetic) operators to operands. Other types of construct, too, such as strings, binary trees, lists, and so on, can be conveniently seen as terms which are obtained using appropriate constructors. Semantically, a term denotes an element of the domain of discourse,  $\mathcal{D}$ .

Syntactically, in the simplest case a term is obtained by applying a function symbol to as many variables and constants as required by its arity. For example, if  $a$  and  $b$  are constants,  $X$  and  $Y$  are both variables and  $f$  and  $g$  have arity 2, then  $f(a, b)$  and  $g(a, X)$  are terms. Nothing prevents the use of terms as the arguments to a function, provided that the arity is respected. We can, for example, write  $g(f(a, b), Y)$  or  $g(f(a, f(X, Y)), X)$  and so on.

In the most general case, we can define terms as follows.

**Definition 12.1** (*Term*) The terms over a signature  $\Sigma$  (and over the set,  $V$ , of variables) are defined inductively<sup>7</sup> as follows:

<sup>6</sup> Recall that, as stated above, the arity denotes the number of arguments of a function or relation.

<sup>7</sup> See the box on Sect. 6.4 for inductive definitions.

- A variable (in  $V$ ) is a term.
- If  $f$  (in  $\Sigma$ ) is a function symbol of arity  $n$  and  $t_1, \dots, t_n$  are terms, then  $f(t_1, \dots, t_n)$  is a term.

As a particular case of the second point, a constant is a term. According to the letter of the definition, a term which corresponds to a constant must be written with parentheses:  $a()$ ,  $b()$ ,  $\dots$ . Let us establish, for ease of reading, that in the case of function symbols of arity 0, parentheses are omitted. Terms without variables are said to be *ground* terms. Terms are usually denoted by the letters  $s, t, u, \dots$ . Note that predicates do not appear in terms; they appear in formulæ (to express properties of terms).

## Formulæ

The *well-formed formulæ* (or *formulæ* for short) of the language allow us to express the properties of terms. Semantically, therefore, a formula states that some elements of our domain,  $\mathcal{D}$ , enjoy a property. For example, if we have the predicate  $>$  (interpreted as usual as a partial order relation on  $\mathcal{D}$ ), writing  $>(3, 2)$  we express that the term “3” corresponds to a value (an element of  $\mathcal{D}$ ) which is greater than the value associated with the term “2”. More complex formulæ can be constructed using logical symbols. For example, the formula  $>(X, Y) \wedge >(Y, Z) \rightarrow >(X, Z)$  expresses the transitivity of  $>$ —if  $>(X, Y)$  is true and  $>(Y, Z)$  is also true then, it is the case that  $>(X, Z)$ .

Wanting to define formulæ precisely, we have first atomic formulæ (or atoms), constructed by the application of a predicate to the number of terms required by its arity. For example, if  $p$  has arity 2, using the two terms introduced above, we can write  $p(f(a, b), f(a, X))$ . Using logical connectives and quantification, we can construct complex formulæ from atomic ones. As usual, we have an inductive definition (or, equivalently, a free grammar—see Exercise 1).

**Definition 12.2** (*Formula*) The (well-formed) formulæ over the signature with terms  $(\Sigma, \Pi)$  are defined inductively as follows:

1. If  $t_1, \dots, t_n$  are terms over the signature  $\Sigma$  and  $p \in \Pi$  is a predicate symbol of arity  $n$ , then  $p(t_1, \dots, t_n)$  is a formula.
2. *true* and *false* are formulæ.
3. If  $F$  and  $G$  are formulæ, then  $\neg F$ ,  $(F \wedge G)$ ,  $(F \vee G)$ ,  $(F \rightarrow G)$  and  $(F \leftrightarrow G)$  are formulæ.
4. If  $F$  is a formula and  $X$  is a variable, then  $\forall X.F$  and  $\exists X.F$  are formulæ.

### 12.2.2 Logic Programs

A formula constructed according to the last definition can have a highly complex structure. This complexity often determines also the effort required to find a proof. In

automatic theorem proving, and in logic programming, particular classes of formulæ have been identified, which lend to more efficient manipulation. One of this classes is the one of the *clauses*, which can be manipulated with a special inference rule called *resolution*. We are interested in a restriction of the notion of clause, called *definite clause*, as well as in a restricted forms of resolution—*SLD resolution*, see Sect. 12.4.3. Using these two notions, the procedure for seeking a proof is not only particularly simple but also allows the explicit calculation of the values of the variables necessary for the proof. These values can be considered as the result of the computation, giving way to an interesting model of computation based on logical deduction. We will see this model in more detail below, for now we will concentrate on syntactic aspects.

**Definition 12.3** (*Logic Program*) Let  $H, A_1, \dots, A_n$  be atomic formulæ. A definite clause (for us simply a “clause”) is a formula of the form:

$$H : -A_1, \dots, A_n.$$

If  $n = 0$ , the clause is said to be a *unit*, or a fact, and the symbol  $: -$  is omitted (but not the final full stop). A logic program is a set of clauses, while a pure PROLOG program is an ordered sequence of clauses. A query (or goal) is a sequence of atoms  $A_1, \dots, A_n$ .

Let us clarify some points about this definition. First, the symbol  $: -$  which we did not include in our alphabet, is just (reversed) implication ( $\leftarrow$ ).<sup>8</sup>

The commas in a clause or in a query should be interpreted as logical conjunction. The notation “ $H : -A_1, \dots, A_n$ .” is therefore an abbreviation for “ $H \leftarrow A_1 \wedge \dots \wedge A_n$ .” Note that the full stop is part of the notation and is important because it tells a potential interpreter or compiler that the clause has terminated.

The part on the left of  $: -$  is the *head* of the clause; that on the right is the *body*. A fact is therefore a clause with an empty body. A program is a set of clauses in the case of the theoretical formalism. In the case of PROLOG, on the other hand, a program is considered as a sequence because, as we will see, the order of clauses is relevant. Here, we used the simplified terminology found in many recent texts (for example, [1]). For more precise terminology, see the next box. The set of clauses containing the predicate symbol  $p$  in their head is said to be the *definition* of  $p$ . Variables occurring in the body of a clause and not in the head are said to be *local* variables.

---

<sup>8</sup>  $: -$  instead of  $\leftarrow$  is used for practical reasons. When logic languages were being introduced, it was much easier to type  $: -$  rather than a left arrow, on most keyboards.

---

## Clause

The reader familiar with first-order logic will have recognised the notion of clause given in Definition 12.3 as being a particular case of the one used in logic. A clause, in the general sense, is indeed a formula of the form:

$$\forall X_1, \dots, X_m (L_1 \vee L_2 \vee \dots \vee L_n)$$

where  $L_1 \dots L_n$  are *literals* (atoms or negated atoms) and  $X_1, \dots, X_m$  are all the variables that occur in  $L_1 \dots L_n$ . For clarity, we may separate the negated atoms from the others, therefore writing a clause as:

$$\forall X_1, \dots, X_m (A_1 \vee A_2 \vee \dots \vee A_m, \neg B_1 \vee \neg B_2 \vee \dots \vee \neg B_k).$$

Using well-known logical equivalences, we can express this formula in the following equivalent form;

$$A_1, \dots, A_m \leftarrow B_1, \dots, B_k. \quad (12.1)$$

A *program clause*, also called a *definite clause*, is a clause that has only one un-negated atom. In the form shown in (12.1), a definite clause always has  $m = 1$ , which is the notion introduced directly in Definition 12.3. A *fact* is therefore a definite clause containing no negated atoms. Finally, a *negative clause*, also called a *query* or *goal*, is a clause of the form (12.1) in which  $m = 0$ .

---

## 12.3 Theory of Unification

The fundamental computational mechanism in logic programming is the solution of equations between terms using the unification procedure. In this procedure, substitutions are computed so that variables and terms can be bound (or instantiated). The composition of the different substitutions obtained in the course of a computation provides the result of the calculation. Before seeing the computational model for logic programming in detail, we must analyse unification in a little detail, a task we undertake in this section.

### 12.3.1 The Logic Variable

Let us clarify, first, that the concept of variable we are considering is different from that seen in Sect. 6.2.1. Here, we consider the so-called *logic variable*, an unknown which can assume values from a predetermined set. In our case, this set is that of definite terms over the given alphabet. This fact, together with the use that logic programming makes of logic variables, gives rise to three important differences between logic variables and modifiable variables of imperative languages.

1. The logic variable can be bound only once, in the sense that if a variable is bound to a term, this binding cannot be destroyed (but the term might be modified, as will be explained below). For example, if we bind the variable  $X$  to the constant  $a$  in a logic program, the binding cannot later be replaced by another which binds  $X$  to the constant  $b$ . Clearly, this is possible in imperative languages using assignment. The fact that the binding of a variable cannot be eliminated does

not mean, however, that it is impossible to modify the value of the variable. This apparent contradiction is dealt with in the next point.

2. The value of a logic variable can be partially defined (or can be undefined), to be specified later. This is because a term that is bound to a variable can contain, in its turn, other logic variables. For example, if the variable  $X$  is bound to the term  $f(Y, Z)$ , successive bindings of the variables  $Y$  and  $Z$  will also modify the value of the variable  $X$ : if  $Y$  is bound to  $a$  and  $Z$  is bound to  $g(W)$ , the value of  $X$  will become the term  $f(a, g(W))$ . The process could continue by modifying the value of  $W$ . This mechanism for specifying the value of a variable by successive approximations, so to speak, is typical of logic languages and is somewhat different from the corresponding one encountered in imperative languages, where a value assigned to a value cannot be partially defined.
3. A third important difference concerns the bidirectional nature of bindings for logic variables. If  $X$  is bound to the term  $f(Y)$  and later we are able to bind  $X$  to the term  $f(a)$ , the effect so produced is that of binding the variable  $Y$  to the constant  $a$ . This does not contradict the first point, given that the binding of  $X$  to the term  $f(Y)$  is not destroyed, but the value of  $f(Y)$  is specified through the binding of  $Y$ . Therefore, we can not only modify the value of a variable by modifying the term to which it is bound, but we can also modify this term by providing another binding for that variable. Clearly, this second binding must be consistent with the first, that is if  $X$  is bound to the term  $f(Y)$ , we cannot try to bind  $X$  to a term of the form  $g(Z)$ .

The last point is fundamental, since it allows us to use a single logic program in quite different ways, as will be seen below. Essentially, we are talking about a property that derives from the unification mechanism, which we will discuss shortly. First, however, we must introduce the concept of substitution.

### 12.3.2 Substitution

The connection between variables and terms is made in terms of the concept of substitution, which, as its name tells us, allows the “substitution” of a variable by a term. A substitution, usually denoted by the greek letters  $\vartheta, \sigma, \rho, \dots$ , can be defined as follows.

**Definition 12.4** (*Substitution*) A substitution is a function from variables to terms such that the number of variables which are not mapped to themselves is finite. We denote a substitution  $\vartheta$  by the notation:

$$\vartheta = \{X_1/t_1, \dots, X_n/t_n\}$$

where  $X_1, \dots, X_n$  are different variables,  $t_1, \dots, t_n$  are terms and where we assume that  $t_i$  is different from  $X_i$ , for  $i = 1, \dots, n$ .

In the preceding definition, a pair  $X_i/t_i$  is said to be a *binding*.<sup>9</sup> In the case in which all the  $t_1, \dots, t_n$  are ground terms, then  $\vartheta$  is said to be a *ground substitution*. We write  $\epsilon$  for the empty substitution. For  $\vartheta$  represented as in Definition 12.4, we define the domain, codomain and variables of a substitution as follows:

$$\text{Domain}(\vartheta) = \{X_1, \dots, X_n\},$$

$$\text{Codomain}(\vartheta) = \{Y \mid Y \text{ a variable in } t_i, \text{ for some } t_i, 1 \leq i \leq n\}.$$

A substitution can be applied to a term, or, more generally, to any syntactic expression, to modify the value of the variables present in the domain of the substitution. More precisely, if we consider an expression,  $E$  (which could be a term, a literal, a conjunction of atoms, etc.), the result of the application of  $\vartheta = \{X_1/t_1, \dots, X_n/t_n\}$  to  $E$ , denoted by  $E\vartheta$ , is obtained by simultaneously replacing every occurrence of  $X_i$  in  $E$  by the corresponding  $t_i$ , for all  $1 \leq i \leq n$ . For example, if we apply the substitution  $\vartheta = \{X/a, Y/f(W)\}$  to the term  $g(X, W, Y)$ , we obtain the term  $g(X, W, Y)\vartheta$ , that is  $g(a, W, f(W))$ . Note that the application is simultaneous. For example, if we apply the substitution  $\sigma = \{Y/f(X), X/a\}$  to the term  $g(X, Y)$ , we obtain  $g(a, f(X))$  (and not  $g(a, f(a))$ ).

The *composition*,  $\vartheta\sigma$ , of two substitutions  $\vartheta = \{X_1/t_1, \dots, X_n/t_n\}$  and  $\sigma = \{Y_1/s_1, \dots, Y_m/s_m\}$  is defined as the substitution obtained by removing from the set

$$\{X_1/t_1\sigma, \dots, X_n/t_n\sigma, Y_1/s_1, \dots, Y_m/s_m\}$$

the pairs  $X_i/t_i\sigma$  such that  $X_i$  is equal to  $t_i\sigma$  and the pairs  $Y_i/s_i$  such that  $Y_i \in \{X_1, \dots, X_n\}$ . Composition is associative and it is not difficult to see that, for any expression,  $E$ , it is the case that  $E(\vartheta\sigma) = (E\vartheta)\sigma$ . The effect of the application of a composition is the same as it is obtained by successively applying the two substitutions that we want to compose.

For example, composing

$$\vartheta_1 = \{X/f(Y), W/a, Z/X\} \quad \text{and} \quad \vartheta_2 = \{Y/b, W/b, X/Z\}$$

we obtain the substitution

$$\vartheta = \vartheta_1\vartheta_2 = \{X/f(b), W/a, Y/b\}.$$

If we apply the latter to the term  $g(X, Y, W)$ , we obtain the term  $g(f(b), b, a)$ . The same term is obtained first by applying  $\vartheta_1$  to  $g(X, Y, W)$ , then applying  $\vartheta_2$  to the result. Note that, in the result  $\vartheta$  of the composition, the  $Y$  in  $\vartheta_1$  is instantiated to  $b$  because of the binding occurring in  $\vartheta_2$ . The  $X$  occurring in  $Z/X$  is instantiated to  $Z$  using the binding  $X/Z$  in  $\vartheta_2$ , after which the binding  $Z/Z$  is eliminated from the resulting substitution (because it is the identity). The bindings  $W/b$  and  $X/Z$

<sup>9</sup> We are using here an “opposite” notation of the one we used in the context of the  $\lambda$ -calculus, Sect. 11.6. There, according to the prevalent tradition in functional languages, we wrote  $N/X$  for the substitution of the term  $N$  in place of the variable  $X$ . Here such a binding is written  $X/N$  (i.e., backwards). In this chapter we will use the latter notation, which is the one most commonly used in logic programming.

present in  $\vartheta_2$  finally disappear from  $\vartheta$  because both  $W$  and  $Z$  appear in the domain (or, on the left of a binding) in  $\vartheta_1$ .

A particular type of substitution is formed from those which simply rename their variables. For example, the substitution  $\{X/W, W/X\}$  does nothing more than change the names of the variables  $X$  and  $W$ . Substitutions like this are called *renamings* and can be defined as follows.

**Definition 12.5** (*Renaming*) A substitution  $\rho$  is a renaming if its inverse substitution  $\rho^{-1}$  exists and is such that  $\rho\rho^{-1} = \rho^{-1}\rho = \epsilon$ .

Note that the substitution  $\{X/Y, W/Y\}$  is not a renaming. Indeed, it not only changes the names of the two variables, but also makes the two variables equal, while they were previously distinct.

Finally, it will be useful to define a preorder,  $\leq$ , over substitutions, where  $\vartheta \leq \sigma$  is read as:  $\vartheta$  is more general than  $\sigma$ . Let us therefore define  $\vartheta \leq \sigma$  if (and only if) there exists a substitution  $\gamma$  such that  $\vartheta\gamma = \sigma$ . Analogously, given two expressions,  $t$  and  $t'$ , we define  $t \leq t'$  ( $t$  is more general than  $t'$ ) if and only if there exists a  $\vartheta$  such that  $t\vartheta = t'$ . The relation  $\leq$  is a preorder and the equivalence induced by it<sup>10</sup> is called the *variance*;  $t$  and  $t'$  are therefore variants if  $t$  is an instance of  $t'$  and, conversely,  $t'$  is an instance of  $t$ . It is not difficult to see that this definition is equivalent to saying that  $t$  and  $t'$  are variants if there exists a renaming  $\rho$  such that  $t$  is syntactically identical to  $t'\rho$ . These definitions can be extended to any expression in an obvious fashion. Finally, if  $\vartheta$  is a substitution that has as domain the set of variables  $V$ , and  $W$  is a subset of  $V$ , the *restriction* of  $\vartheta$  to the variables in  $W$  is the substitution obtained by considering only the bindings for variables in  $W$ , that is the substitution defined as follows:

$$\{Y/t \mid Y \in W \text{ and } Y/t \in \vartheta\}.$$

A comparison with the imperative paradigm can help in better understanding the concepts under consideration. As we saw in Sect. 2.5, in the imperative paradigm, the semantics can be expressed by referring to a concept of state that associates every variable with a value.<sup>11</sup> An expression containing variables is evaluated with respect to a state to obtain a value that is completely defined.

In the logic paradigm, the association of values with variables is implemented through substitutions. The application of a substitution to a term (or to a more complex expression) can be seen as the evaluation of the terms, an evaluation that returns another term, and therefore, in general, a partially defined value.

<sup>10</sup> Given a preorder,  $\leq$ , the equivalence relation *induced* by  $\leq$  is defined as  $t = t'$  if and only if  $t \leq t'$  and  $t' \leq t$ .

<sup>11</sup> Wishing to be precise, as we have seen in the box on Sect. 6.3, in real languages, this association is implemented using two functions, environment and memory. This however does not alter the import of what we are saying.



### 12.3.3 Most General Unifier

The basic computation mechanism for the logic paradigm is the evaluation of equations of the form  $s = t$ ,<sup>12</sup> where  $s$  and  $t$  are terms and “=” is a predicate symbol interpreted as syntactic equality over the set of all ground terms; this set is called the *Herbrand Universe*.<sup>13</sup> We will attempt better to clarify this equality.

If we write  $X = a$  in a logic program, we mean that the variable  $X$  must be bound to the constant  $a$ . The substitution  $\{X/a\}$  therefore is a solution to this equation since, by applying the substitution to the equation, we obtain  $a = a$  which is a syntactical identity. You should not be deceived by the syntactic analogy with assignment in imperative languages, for it deals with a completely different concept. Indeed, unlike in an imperative language, here we can also write  $a = X$  instead of  $X = a$  and the meaning does not change (the equality that we are considering is symmetric, as are all equality relations). Also the analogy with the equality of arithmetic expressions can be, in some ways, misleading, as illustrated by the following example.

Let us assume that we have a binary function symbol  $+$  which, intuitively, expresses the sum of two natural numbers, and consider the equation  $3 = 2 + 1$ , where, for simplicity, we use infix notation for  $+$  and we represent in the usual fashion the natural numbers. Given that the equation  $3 = 2 + 1$  does not contain variables, it can be either true (or, solved) or false (that is, insoluble). Contrary to what arithmetic intuition would suggest to us, in a (pure) logic program this equation cannot be solved. This is because, as we have said, the symbol  $=$  is interpreted as syntactic equality over the set of ground terms (the Herbrand universe). It is clear that, from the syntactic viewpoint, the constant 3 is different from the term  $2 + 1$  and since they are treated as ground terms (that is completely instantiated terms) there is no way that they can be made syntactically equal. Analogously, the equation  $f(X) = g(Y)$  has no solutions (it is not solvable) because however the variables  $X$  and  $Y$  are instantiated, we cannot make the two different function symbols,  $f$  and  $g$ , equal. Note that the equation,  $f(X) = f(g(X))$ , also has no solutions, because the variable  $X$  in the left-hand term must be instantiated with  $g(X)$  and therefore the possible solution must contain the substitution  $\{X/g(X)\}$ . The application of this substitution to the term on the right would instantiate  $X$ , producing the term  $f(g(g(X)))$ , so the  $X$  in the right-hand term would have to be instantiated to  $g(g(X))$  rather than to  $g(X)$  and so on, without ever reaching a solution.<sup>14</sup> In general, therefore, the equation  $X = t$  cannot be solved if  $t$  contains the variable  $X$  (and  $t$  is different from  $X$ ).

<sup>12</sup> We draw attention to the notation that, as usual, overloads the “=” symbol. By writing  $\vartheta = \{X_1/t_1, \dots, X_n/t_n\}$ , we mean that  $\vartheta$  is the substitution  $\{X_1/t_1, \dots, X_n/t_n\}$ , while writing  $s = t$ , we mean an equation.

<sup>13</sup> The symbol “=” is usually written in infix notation for increased readability. Different logic languages can have different syntactic readings for it and use different equality symbols, each with a different meaning. Here, we refer to “pure” logic programming.

<sup>14</sup> By admitting infinite terms, we can find a solution, however.

On the other hand, the equation  $f(X) = f(g(Y))$  is solvable. One solution is the substitution  $\vartheta = \{X/g(Y)\}$  because if this is applied to the two terms in the equation, it makes them syntactically equal. Indeed,  $f(X)\vartheta$  is identical to  $f(g(Y))$  which is identical to  $f(g(Y))\vartheta$ . In more formal terms, we can say that the substitution  $\vartheta$  *unifies* the two terms of the equation— $\vartheta$  is a *unifier*. We have said “a solution” because there are many substitutions (an infinite number of them) that are unifiers of  $X$  and  $g(Y)$ . It is sufficient to instantiate  $Y$  in the definition of  $\vartheta$ . So, for example, the substitution  $\{X/g(a), Y/a\}$  is also an unifier, as are  $\{X/g(f(Z)), Y/f(Z)\}$ ,  $\{X/g(f(a)), Y/f(a)\}$ , and so on. All these substitutions are, however, *less general* than  $\vartheta$  according to the preorder that we defined above. Each of them, that is, can be obtained by composing  $\vartheta$  with some other, appropriate, substitution. For example,  $\{X/g(a), Y/a\}$  is equal to  $\vartheta\{Y/a\}$  (remember that we denote the composition of substitutions with juxtaposition). In this sense, we say that  $\vartheta$  is the most general unifier (or m.g.u.) of  $X$  and  $g(Y)$ .

Before moving to general definitions, note one last important detail: the process of solving an equation (the unification process), can create bidirectional bindings, that is the direction in which the associations must be realised is not specified. For example, a solution of the equation  $f(X, a) = f(b, Y)$  is given by the substitution  $\{X/b, Y/a\}$ , where a variable on the left and one on the right of the  $=$  symbol are bound. Instead, in the equation,  $f(X, a) = f(Y, a)$ , to find a solution, we can choose whether to bind the variable on the left (using  $\{X/Y\}$ ) or the one on the right (using  $\{Y/X\}$ ) or even both (using  $\{X/Z, Y/Z\}$ ).

This aspect, to which we will return, is important because it allows the implementation of bidirectional parameter-passing mechanisms. It allows also a unique characteristic of the logic paradigm—to use the same program in different ways, turning input arguments into outputs and vice versa, without modifying the program.

We may now give the formal definition.

**Definition 12.6** (*M.g.u.*) Given a set of equations  $E = \{s_1 = t_1, \dots, s_n = t_n\}$ , where  $s_1, \dots, s_n$  and  $t_1, \dots, t_n$  are terms, the substitution,  $\vartheta$ , is a unifier for  $E$  if the sequence  $(s_1, \dots, s_n)\vartheta$  and  $(t_1, \dots, t_n)\vartheta$  are syntactically identical. A unifier of  $E$  is said to be the *most general unifier* (m.g.u.) if it is more general than any other unifier of  $E$ ; that is, for every other unifier,  $\sigma$ , of  $E$ ,  $\sigma$  is equal to  $\vartheta\tau$  for some substitution,  $\tau$ .

The preceding concept of unifier can be extended to other syntactic objects in an obvious fashion. In particular, we say that  $\vartheta$  is a unifier of two atoms  $p(s_1, \dots, s_n)$  and  $p(t_1, \dots, t_n)$  if  $\vartheta$  is a unifier of  $\{s_1 = t_1, \dots, s_n = t_n\}$ .

### 12.3.4 A Unification Algorithm

An important result, due to Robinson in 1965, shows that the problem of determining whether a set of equations of terms can be unified is decidable. The proof is constructive, in the sense that it provides a *unification algorithm* which, for every set of equations, produces their m.g.u. if the set is unifiable and returns a failure in the opposite case.<sup>15</sup>

It can also be proved that a m.g.u. is unique up to renaming. The unification algorithm which we will now see is not Robinson's original but is Martelli and Montanari's from 1982 and it makes use of ideas present in Herbrand's thesis of 1930.

#### Martelli and Montanari's Unification Algorithm

Given a set of equations

$$E = \{s_1 = t_1, \dots, s_n = t_n\},$$

the algorithm produces either failure or a set of equations in the following, so called, solved form:

$$\{X_1 = r_1, \dots, X_m = r_m\},$$

where  $X_1, \dots, X_m$  are distinct variables not appearing in the terms  $r_1, \dots, r_m$ . The set of equations is equivalent to the starting set,  $E$ , and from it we can obtain a m.g.u. for  $E$  simply interpreting every equality as a binding. Therefore, the m.g.u. that we seek is the substitution:

$$\{X_1/r_1, \dots, X_m/r_m\}.$$

The algorithm is non-deterministic in the sense that when there are more possible actions, one is chosen in an arbitrary fashion, with no priority between the various actions.<sup>16</sup> The algorithm is given by the following steps.

1. Nondeterministically select one equation from the set  $E$ .
2. According to the type of equation chosen, execute, if possible, one of the specific operations as follows (where on the left of the ":" we indicate the type of equation and, on the right, the associated action):
  - a.  $f(l_1, \dots, l_k) = f(m_1, \dots, m_k)$ : eliminate this equation from the set  $E$  and add to  $E$  the equations  $l_1 = m_1, \dots, l_k = m_k$ .
  - b.  $f(l_1, \dots, l_k) = g(m_1, \dots, m_k)$ : if  $f$  is different from  $g$ , terminate with failure.

<sup>15</sup> Robinson's original algorithm considers the unification of just two terms but this, obviously, is not reductive given that the unification of  $\{s_1 = t_1, \dots, s_n = t_n\}$  can be seen as the unification of  $f(s_1, \dots, s_n)$  and  $f(t_1, \dots, t_n)$ .

<sup>16</sup> We will return to the non-determinism briefly when we discuss the operational semantics of logic programs.

- c.  $X = X$ : eliminate this equation from the set  $E$ .
  - d.  $X = t$ : if  $t$  does not contain the variable  $X$  and this variable appears in another equation in the set  $E$ , apply the substitution  $\{X/t\}$  to all the other equations in the set  $E$ .
  - e.  $X = t$ : if  $t$  contains the variable  $X$ , terminate with failure.
  - f.  $t = X$ : if  $t$  is not a variable, eliminate this equation from the set  $E$  and add to  $E$  the equation  $X = t$ .
3. If none of the preceding operations is possible, terminate with success ( $E$  contains the solved form). If, on the other hand, an operation different from termination with failure has been executed, go to (1).

This is a simple algorithm, but it is worth discussing in detail its steps.

In the first case the two terms agree on the function symbol. To unify the two terms, it is necessary to unify the arguments, so we replace the original equation with the equations obtained by equating the arguments in each position. Note that this case also includes equivalence between constants of the form  $a = a$  (where  $a$  is a function symbol of arity 0) which are eliminated without adding anything.

The second case produces a failure given that, when  $f$  and  $g$  are different, the two terms cannot be unified.

The equation  $X = X$  is eliminated using the identity substitution which therefore produces no change in the other equations.

The fourth case is the most interesting. An equation  $X = t$  is already in solved form (because, by assumption,  $t$  does not contain  $X$ ). In other words, the substitution  $\{X/t\}$  is the m.g.u. of this very equation. We need to combine the effect of such an m.g.u. with those produced by other equations, so we apply the substitution  $\{X/t\}$  to all the other equations in  $E$ .

On the other hand, in the case in which  $t$  contains the variable  $X$ , as we have already seen, the equation has no solution and therefore the algorithm terminates with failure. It is important to note that this check, called the *occurs check*, is removed from many implementations of PROLOG for reasons of efficiency. Therefore, many PROLOG implementations use an incorrect unification algorithm!

The last case, finally, serves only to obtain a result form in which the variables appear on the left and the terms on the right of the  $=$  symbol.

It is easy to convince oneself that the algorithm terminates, given that the depth of the input terms is finite. It is, moreover, possible to prove that the algorithm produces an m.g.u. which is obtained by the interpretation as substitutions of the final result form of the equations.

By a careful consideration of the algorithm, it can be seen that the computation of the most general unifier occurs in an incremental fashion by solving ever simpler equations until a result form is encountered. It is also possible to express this process in terms of substitution compositions as happens in the operational model of logic languages. We will exemplify this point with an example. To simplify, we will always choose the leftmost equation (the final result, anyway, does not depend upon such an assumption; any other selection rule would lead to the same result up to renaming).

Let us consider the set of equations:

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

Choosing the first equation on the left, using the operation described in (a), the set  $E$  is transformed into:

$$E_1 = \{X = g(Y), b = W, h(X, Y) = h(Z, W)\}.$$

Using operation (d), we therefore obtain:

$$E_2 = \{X = g(Y), b = W, h(g(Y), Y) = h(Z, W)\}.$$

Using (f) and then (d) again on the second equation, we finally obtain:

$$E_3 = \{X = g(Y), W = b, h(g(Y), Y) = h(Z, b)\}.$$

This already contains the result of the first equation in the set  $E$ . In fact, the substitution:

$$\vartheta_1 = \{X/g(Y), W/b\}$$

is an m.g.u. for  $f(X, b) = f(g(Y), W)$ .

Continuing with the second equation from the original set, suitably instantiated by the substitutions already computed, using operation (a), we obtain:

$$E_4 = \{X = g(Y), W = b, g(Y) = Z, Y = b\};$$

then, by (f), we obtain:

$$E_5 = \{X = g(Y), W = b, Z = g(Y), Y = b\}.$$

Finally, using (d) applied to the last equation, we have:

$$E_4 = \{X = g(b), W = b, Z = g(b), Y = b\},$$

which is the result form for the set  $E$  and therefore also provides the m.g.u. of the initial set in the form of the substitution

$$\vartheta = \{X/g(b), W/b, Z/g(b), Y/b\}.$$

Two important observations can be made. First, note how the value of some variables can be partially specified first, and then later refined. For example,  $\vartheta_1$  (m.g.u. of the first equation in  $E$ ) tells us that  $X$  has  $g(Y)$  as its value and only by solving the second equation do we see that  $Y$  has  $b$  as its value. Therefore it can be seen that  $g(b)$  is the value of  $X$  (as, indeed, it results in the final m.g.u.).

Moreover, we can see that if we consider  $\{h(g(X), Y) = h(Z, W)\}\vartheta_1$  (the second equation in  $E$  instantiated using the m.g.u. of the first), we obtain the equation

$$h(g(Y), Y) = h(Z, b)$$

for which the substitution

$$\vartheta_2 = \{Z = g(b), Y = b\}$$

is a m.g.u. Using the definition of composition of substitutions, it is easy to check that  $\vartheta = \vartheta_1\vartheta_2$ . The m.g.u. of the set  $E$  can therefore be obtained by composing the first equation's m.g.u. with that of the second (to which the first m.g.u. has already been applied). This, as we have already said, is what, indeed, normally happens in implementations of logic languages, where, instead of accumulating all the equations and then solve them, an m.g.u. is computed on each step of the computation and is composed with the ones that were previously obtained.

## 12.4 The Computational Model

The logic paradigm, implementing the idea of “computation as deduction”, uses a computational model that is substantially different from all the others that we have seen so far. Wishing to synthesise, we can identify the following main differences from the other paradigms:

1. The only possible values, at least in the pure model, are terms over a given signature.
2. Programs can have a declarative reading which is entirely logical, or a procedural reading of an operational kind.
3. Computation works by instantiating the variables appearing in terms (and therefore in goals) to other terms using the unification mechanism.
4. Control, which is entirely handled by the abstract machine (except for some possible annotations in PROLOG) is based on the process of automatic backtracking.

Below, we analyse these four points. We will explicitly discuss the differences between logic programming and PROLOG.

### 12.4.1 The Herbrand Universe

In logic programming, terms are a fundamental element. The set of all possible terms over a given signature is called the *Herbrand Universe* and is the domain over which computation in logic programs is performed. It has some characteristics that must be understood.

- The alphabet over which programs are defined is not fixed but can vary as far as non-logical symbols are concerned.
- As a (partial) consequence of the previous point, unlike what happens in imperative languages, no predefined meaning is assigned to the (non-logical) symbols of the alphabet. For example, a program can use the  $+$  symbol to denote addition, while another program can use the same symbol to denote string concatenation. The exceptions are the (binary) equality predicate and some other predefined (“built-in”) symbols in PROLOG.<sup>17</sup>
- As a final consequence, no type system is present in logic languages (at least in the classic formalism). The only type that is present is that of terms with which we can represent arithmetic, list, expressions, etc.

---

<sup>17</sup> As well as predefined predicates in constraint languages, as we will see in the next chapter.

From the theoretical stance, the fact that there are no types and that computation occurs in the Herbrand Universe is not limiting, rather it permits the highly elegant and, all considered, simple expression of the formal semantics of logic programs. For example, with only two function symbols, 0 (constant zero) and  $s$  (successor, of arity 1), we can express the natural numbers using the terms 0,  $s(0)$ ,  $s(s(0))$ ,  $s(s(s(0)))$ , etc. With a little effort, we can express the normal arithmetic operations in terms of this two-symbol representation.

From the practical view point, on the other hand, the lack of types is a serious problem. In fact, in PROLOG, as in other, more recent, logic languages, some primitive types have been introduced (for example, integers and associated arithmetic operations). The languages of this paradigm are therefore always somewhat lacking as far as types are concerned.

### 12.4.2 Declarative and Procedural Interpretation

As we have just hinted, a clause, and therefore a logic program, can have two different interpretations: one *declarative* and one *procedural*.

From the *declarative* viewpoint, a clause  $H : -A_1, \dots, A_n$  is a formula which expresses that if  $A_1$  and  $A_2$  and  $\dots$  and  $A_n$  are true, then  $H$  is also true. A query (or goal) is also a formula for which we want to prove, provided that it is appropriately instantiated, that it is a logical consequence of the program, and is, therefore, true in all interpretations in which the program is true.<sup>18</sup> This interpretation can be developed using the methods of logic (in particular, some elementary concepts of model theory) in such a way as to give a meaning to a program in purely declarative terms without referring at all to a computational process. For this interpretation, while interesting, we refer the reader to the specialist literature cited at the end of the chapter.

The *procedural* interpretation, on the other hand, allows us to read a clause such as:

$$H : -A_1, \dots, A_n$$

as follows. To prove  $H$ , it is necessary first to prove  $A_1, \dots, A_n$ , or rather to compute  $H$ , it is necessary first to compute  $A_1, \dots, A_n$ . From this, we can view a predicate as the name of a procedure, whose defining clauses constitute its body. In this interpretation, we can read an atom in the body of a clause, or in a goal, as a procedure call. A logic program is therefore a set of declarations and a goal is no more than the equivalent of “main” in an imperative program, given that it contains all the calls to the procedures that we want to evaluate. The comma in the body of clauses and in goals, in PROLOG (but not in other pure logic programming languages), can be read as the analogue of “;” in imperative languages.

---

<sup>18</sup> Here we will content ourselves with an intuitive idea of this concept. The interested reader can consult any text on logic for more information.

Precise correspondence theorems allow us to reconcile the declarative and procedural views, proving that the two approaches are equivalent.

From a formal viewpoint, the procedural interpretation is supported by so-called SLD resolution, a logical inference rule which we will discuss in the box on Sect. 12.4.3. It is also possible to describe the procedural interpretation in a more informal manner, using only the analogy with procedure calls and parameter passing which we have just outlined. This is the approach we will use below.

### 12.4.3 Procedure Calls

Let us consider for now a simplified definition of clause in which we assume that, in the head, all the arguments of the predicate are distinct variables. An arbitrary clause of this type therefore has the form:

$$p(X_1, \dots, X_n) : -A_1, \dots, A_m$$

and, as we have anticipated, it can be seen as the declaration of the procedure  $p$  with  $n$  formal parameters,  $X_1, \dots, X_n$ . An atom  $q(t_1, \dots, t_n)$  can be seen as a call to the procedure  $q$  with  $n$  actual parameters,  $t_1, \dots, t_n$ . In the definition of  $p$ , therefore, the body is formed from the calls to  $m$  procedures which constitute the atoms  $A_1, \dots, A_m$ .

In accordance with this view, and analogously to what happens in imperative languages, the evaluation of the call  $p(t_1, \dots, t_n)$  causes the evaluation of the body of the procedure after parameter passing has been performed. Parameter passing uses a technique similar to call by name, replacing the formal parameter  $X_i$  with the corresponding actual parameter  $t_i$ . Moreover, given that the variables appearing in the body of the procedure are to be considered as logical variables, they can be considered to be distinct from all other variables. In block-structured languages, this happens implicitly given that the body of the procedure is considered as a block with its own local environment. Here, on the other hand, the concept of block is absent, so, in order to avoid conflicts between variable names, we assume that, before using a clause, all the variables appearing in it are systematically renamed (so that they do not conflict with any others).

Using more precise terms, we can say that the evaluation of the call  $p(t_1, \dots, t_n)$ , with the definition of  $p$  seen above, causes the evaluation of the  $m$  calls:

$$(A_1, \dots, A_m)\vartheta$$

present in the body of  $p$ , appropriately instantiated by the substitution

$$\vartheta = \{X_1/t_1, \dots, X_n/t_n\}$$

which performs parameter passing. In the case in which the body of the clause is empty (or that  $m = 0$ ), the procedure call terminates immediately. Otherwise, the



computation proceeds with the evaluation of the new calls. Using logic programming terminology, this can be expressed by saying that the evaluation of the goal  $p(t_1, \dots, t_n)$  produces the new goal:

$$(A_1, \dots, A_m)\{X_1/t_1, \dots, X_n/t_n\}$$

which, in its turn, will have to be evaluated. When all the calls generated by this process have been evaluated (provided there has been no failure), the computation terminates with success and the final result is composed of the substitution that associates the values computed in the course of the computation with the variables present in the initial call ( $X_1, \dots, X_n$ , in our case). This substitution is said to be the *computed answer* for the initial goal in the given program. We will see a more precise definition of this concept below. For now, we will see a simple example. Let us consider the procedure `list_of_2` defined below and identical to the procedure `list_of_27` of Sect. 12.1.1, except for the lesser number of anonymous variables:

```
list_of_2(Ls):- Ls = [_,_].
```

The evaluation of the call `list_of_2(LXs)` causes the evaluation of the body of the clause, instantiated by the substitution  $\{Ls/LXs\}$ , that is:

```
LXs = [_,_]
```

This is a particular call because `=` is a predefined predicate which, as we saw in the previous section, is interpreted as syntactic equality over the Herbrand Universe and, operationally, corresponds to the unification operation. The previous call therefore reduces to the attempt (performed by the language interpreter) to solve the equation using unification. In our case, clearly, this attempt succeeds and produces the m.g.u.  $\{LXs/[_,_]\}$  which is the result of the computation. The previous substitution is therefore the computed answer for the goal `list_of_2` in the logic program defined by the single line (1) above.

## Evaluation of a Non-atomic Goal

The view presented in the previous subsection must be generalised and made more precise to clarify the logic-programming computational model. Below, in order to conform with current terminology, we will talk of atomic goals and goals rather than procedure calls and sequences of calls. The analogy with the conventional paradigm remains valid, though.

In the case in which a non-atomic goal must be evaluated, the computational mechanism is analogous to the one seen above, except that now we must choose one of the possible calls using some *selection rule*. While in the case of pure logic programming no such rule is specified, PROLOG adopts the rule that the leftmost atom is always chosen. It is however possible to prove that whatever rule is adopted, the results that are computed are always the same (see also Exercises 13 and 14 at the end of the chapter).

Assuming, for simplicity, that we adopt the PROLOG rule, we can describe the progress of the evaluation process. Let

$$B_1, \dots, B_k$$

with  $k \geq 1$ , be the goal to be evaluated. We distinguish the following cases according to the form of the selected atom,  $B_1$ :

1. If  $B_1$  is an equation of the form  $s = t$ , try to compute a m.g.u. (using the unification algorithm). There are two possibilities:

- a. If the m.g.u. exists and is the substitution  $\sigma$ , then the result of the evaluation is the goal:

$$(B_2, \dots, B_k)\sigma$$

obtained from the previous one by eliminating the chosen atom and applying the m.g.u. thus computed. If  $k = 1$  (and therefore  $(B_2, \dots, B_k)\sigma$  is empty), the computation terminates in success.

- b. If the m.g.u. does not exist (or the equation has no solutions), then we have failure.

2. If, on the other hand,  $B_1$  has the form  $p(t_1, \dots, t_n)$ , we have the following two cases:

- a. If, in the program, there exists a clause of the form:

$$p(X_1, \dots, X_n) : -A_1, \dots, A_m$$

(which we consider renamed to avoid variable capture), then the result of evaluation is a new goal:

$$(A_1, \dots, A_m)\vartheta, B_2, \dots, B_k$$

where  $\vartheta = \{X_1/t_1, \dots, X_n/t_n\}$ . If  $k = 1$  (then we have an atomic goal) and  $m = 0$  (the body of the clause is empty), then the computation terminates with success.

- b. If, in the program, there exists no clause defining the predicate  $p$ , we have failure.

To be able exactly to define the results of the computation (the computed answers), we need to clarify some aspects of control, which we will do in Sect. 12.4.4.

## SLD Resolution

Definite clauses allow a natural procedural reading based on *resolution*, an inference rule which is complete for sets of clauses and which was introduced by Robinson and used in automated deduction. In logic programming, we have *SLD resolution*, that is linear resolution guided by a clause selection rule (SLD is an acronym for Selection rule-driven Linear resolution for Definite clauses).

This rule can be described as follows. Let  $G$  be the goal  $B_1, \dots, B_k$  and let  $C$  be the (definite) clause  $H : -A_1, \dots, A_n$ . We say that  $G'$  is *derived* from  $G$  and  $C$  using  $\vartheta$  or, equivalently,  $G'$  is a *resolvent* of  $G$  and  $C$  if (and only if) the following conditions are met:

1.  $B_m$ , with  $1 \leq m \leq k$ , is a *selected* atom from those in  $G$ .
2.  $\vartheta$  is the m.g.u. of  $B_m$  and  $H$ .
3.  $G'$  is the goal  $(B_1, \dots, B_{m-1}, A_1, \dots, A_n, B_{m+1}, \dots, B_k)\vartheta$ .

Note that, unlike Sect. 12.4.3, here it is necessary also to apply  $\vartheta$  to the other atoms occurring in the goal  $G$  because the heads of the clauses contain arbitrary terms and therefore  $\vartheta$  could instantiate variables that are also in the goal.

Given a goal,  $G$ , and a logic program,  $P$ , an *SLD derivation* of  $P \cup G$  consists of a (possibly infinite) sequence of goals  $G_0, G_1, G_2, \dots$ , of a sequence  $C_1, C_2, \dots$  of clauses in  $P$  which have been renamed in such a way as to avoid variable capture and a sequence  $\vartheta_1, \vartheta_2, \dots$ , of m.g.u. s such that  $G_0$  is  $G$  and, for  $i \geq 1$ , every  $G_i$ , is derived from  $G_{i-1}$  and  $C_i$  using  $\vartheta_i$ . An *SLD refutation* of  $P \cup G$  is a finite SLD derivation of  $P \cup G$  which has the empty clause as the last resolvent of the derivation. If  $\vartheta_1, \vartheta_2, \dots$  are the m.g.u. s used in the refutation of  $P \cup G$ , we say that the substitution  $\vartheta_1 \vartheta_2 \dots \vartheta_n$  restricted to the variables occurring in  $G$  is the *computed answer substitution* for  $P \cup G$  (or for the goal  $G$  in the program  $P$ ).

Classic results, due to K. L. Clark, show that this rule is sound and complete with respect to the traditional first-order logical interpretation.

Indeed, it can be proved that if  $\vartheta$  is the *computed answer substitution* for the goal  $G$  in program  $P$ , then  $G\vartheta$  is a logical consequence of  $P$  (soundness). Moreover, if  $G\vartheta$  is a logical consequence of  $P$ , then no matter which selection rule is used, there exists a SLD refutation of  $P \cup G$  with computed answer  $\sigma$  such that  $G\sigma$  is more general than  $G\vartheta$  (strong completeness).

## Heads with Arbitrary Terms

So far, we have assumed that the heads of clauses contain just distinct variables. We have made this choice to preserve the similarity with procedures in traditional languages. However, real logic programs also use arbitrary terms as arguments to predicates in heads, as we have seen in the example in Sect. 12.1.1. The box on Sect. 12.4.3 provides the evaluation rule for such a general case. Note, however, that our assumption is in no way limiting (apart, perhaps, from textual convenience). Indeed, as it is clear from what was said in the box and from the preceding treatment, a clause of the form:

$$p(t_1, \dots, t_n) : -A_1, \dots, A_m$$

can be seen as an abbreviation of the clause:

$$p(X_1, \dots, X_n) : -X_1 = t_1, \dots, X_n = t_n, A_1, \dots, A_m.$$

In the following examples, we will often use the notation with arbitrary terms in heads.

#### 12.4.4 Control: Non-determinism

In the evaluation of a goal, we have two degrees of freedom: the selection of the atom to evaluate and the choice of the clause to apply.

For the first, we have said that we can fix a selection rule, without influencing the final results of the computation that terminate in success. For this reason, the non-determinism arising from the selection of the atom is called “*don’t care*”.

For the selection of clauses, on the other hand, the matter is more delicate. Given that a predicate can be defined by more than one clause, and that we have to use only one of them at a time, we could think about fixing some rule for choosing clauses, analogous to the one used to choose atoms. The following program reveals however a problem. Let  $\mathcal{P}a$  be the program:

$p(X) :- p(X).$	1
$p(X) :- X=a.$	2

and let us assume that we choose clauses from top to bottom, following the textual order of the program. It is easy to see that by adopting this rule, the evaluation of  $p(Y)$  never terminates and therefore we do not have a computed answer by the program. Indeed, using clause (1) and the substitution  $\{X/Y\}$ , the initial goal, after one computation step, becomes the goal  $p(Y)$ , which is again the starting goal. According to the clause-selection rule, we must choose clause (1) and apply it to this (new) goal, and so on. It is however clear that using clause (2) we would immediately obtain a terminating computation, producing the substitution  $\{Y/a\}$  as result. Note that to fix another order, for example, from bottom to top, would not in general solve this problem.

In the light of this example, let us carefully reconsider the evaluation rule for a goal seen in the previous subsection and, in particular, let us fix on point 2(a), where we wrote “if there exists a clause” without specifying how this can be chosen. By doing this, we have therefore introduced into the computation model a form of non-determinism: in the case in which there is more than one clause for the same predicate, we can choose one in a non-deterministic fashion, without adopting any particular rule. This form of non-determinism is called “*don’t know*” non-determinism, because we do not know which the “right” clause will be that allows the termination of the computation with success. The theoretical model of logic programming keeps this non-determinism when it considers all possible choices of clause and therefore all possible results of the various computations that are produced as a consequence of this choice. The result of the evaluation of a goal  $G$  in the program  $P$  is therefore a set of computed answers, where each of these answers is the substitution obtained by the composition of all the m.g.u.s which are encountered in a specific computation (with specific choices of clauses), restricted to the variables present in  $G$ . For a more precise definition of this idea, as well as of the whole process of evaluating a goal, see the box on Sect. 6.4.

Turning to our previous example, the only computed answer for the goal  $p(Y)$  in program  $\mathcal{P}a$  is the substitution  $\{Y/a\}$  while, for the same program, the goal  $p(b)$  has no computed answer given that all its computations either terminate with failure (when the second clause is used) or do not terminate (when the first clause is used).

### Backtracking in PROLOG

When moving from the theoretical model to an implemented language, such as PROLOG, non-determinism must, at some level, be transformed into determinism, given that the physical computing machine is deterministic. This can obviously be done in various ways, and in principle does not cause loss of solutions. For example, we could think of starting  $k$  parallel computations when there are  $k$  possible clauses for a predicate<sup>19</sup> and therefore consider the results of all the possible computations.

In PROLOG, however, for reasons of simplicity and implementation efficiency, the strategy we first saw is employed: clauses are used according to the textual order in which they occur in the program (top-to-bottom). We saw in the previous example that this strategy is *incomplete*, given that it does allow us to find all possible computed answers. This limit, however, is adjustable by the programmer who, knowing this property of PROLOG, can order the clauses in the program in the most convenient way (typically putting first those relating to terminal cases and then the inductive steps). Note, though, that this trick, at least in principle, eliminates some of the declarative nature of the language, because the programmer has specified an aspect of control.

In addition to infinite computations, there is a second, more important aspect to be considered by adopting the deterministic model of PROLOG and deals with the handing of failures. Let us first see an example. Let us consider the program  $\mathcal{P}b$ :

$p(X) :- X = f(a).$	1
$p(X) :- X = g(a).$	2

Let us consider the evaluation of the goal  $p(g(Y))$ . By the PROLOG strategy, clause (1) is chosen, which, using the substitution  $\{X/g(Y)\}$ , produces the new goal  $g(Y) = f(a)$ . This fails, given that the two terms in the equation cannot be unified. However, given that there is still one clause to use, it would not be acceptable to terminate the computation by returning a failure. “Backtracking” then occurs, returning to the choice of the clause for  $p(g(Y))$  and therefore continues the computation by trying clause (2). In this way, the computation achieves success with the computed answer  $\{Y/a\}$ .

In general, therefore, when arriving at a failure, the PROLOG abstract machine “backtracks” to a previous choice point at which there are other choices; that is, where there are other clauses to test. In this backtracking process, the bindings that might have been computed in the previous computation must be undone. Once a choice point has been reached, a new clause is tried and the computation continues in the way we have seen. If the previous choice point contains no possibilities for computations,

---

<sup>19</sup> Clearly, on a machine with one processor, the  $k$  parallel computations must be appropriately “scheduled” so they can be executed in a sequential manner, analogous to what happens with processes in a multitasking operating system.

an older choice point is sought and, if there are no more, the computation terminates in failure. Note that all of this is handled directly by the PROLOG abstract machine and is completely invisible to the programmer (except the use of particular constructs such as the cut which we will introduce in Sect. 12.5.1).

It is also easy to see how this procedure, which uses a search corresponding to a depth-first search through a tree representing the possible computations, can be computationally demanding. The solution to the problem seen in Sect. 12.1.1, for example, requires extensive backtracking and is fairly wasteful in computation time.

Let us now see another example. Consider the program `Pc`:

```

p ( X ) : -   X = f ( Y ) ,   q ( X ) .           1
p ( X ) : -   X = g ( Y ) ,   q ( X ) .           2
q ( X ) : -   X = g ( a ) .                       3
q ( X ) : -   X = g ( b ) .                       4

```

Let us analyse the goal  $p(Z)$ . Using clause (1), we obtain the goal  $Z = f(Y), q(Z)$ . The evaluation of the equation produces the m.g.u.  $\{Z/f(Y)\}$  and we obtain the goal  $q(f(Y))$ . Using clause (3), we obtain the goal  $f(Y) = g(a)$ , which fails. At this point we must turn to the last choice point, that is to the point where the choice of clause for the predicate  $q$  occurred. In this case, there are no bindings to undo, and therefore we try clause (4), obtaining therefore the goal  $f(Y) = g(b)$  which also fails. We return again to the choice point for  $q$  and we see that there are no other possible clauses and therefore we return to the previous choice point (the one for predicate  $p$ ). Doing this, we have to undo the binding  $\{Z/f(Y)\}$  calculated by clause (1) and therefore we return to the initial situation, where variable  $Z$  is not instantiated. Using clause (2), we obtain the goal  $Z = g(Y), q(Z)$  and therefore by the evaluation of the equation, we obtain the m.g.u.  $\{Y/a\}$  and the new goal  $q(g(Y))$ . At this point, using clause (3), we obtain the goal  $g(Y) = g(a)$  which succeeds and produces the m.g.u.  $\{Y/a\}$ . Given that there remain no more goals to evaluate, the computation terminates with success. The result of the computation is produced by composing the computed m.g.u.  $s\{Z/g(Y)\}\{Y/a\}$  and restricting the substitution  $\{Z/g(a), Y(a)\}$  obtained by this composition to the single variable present in the initial goal. Therefore, the computed answer obtained is  $\{Z/g(a)\}$ . Note that there is another computed answer,  $\{Z/g(b)\}$ , which we can obtain using clause (4) rather than (3). In PROLOG implementations, answers subsequent to the first can be obtained using the “;” command. Finally, the reader can easily check that the goal  $p(g(c))$  in program `Pc` terminates with failure, a result that is obtained after having proved all four of the possible clause combinations.

### 12.4.5 Some Examples

In this subsection, we will focus on the PROLOG language, and we will make use of its syntax. The notation  $[h|t]$  is used to denote the list which has  $h$  as its head and  $t$  as its tail. Let us remember that the head is the first element in the list, while the tail is what remains of the list when the head has been removed. The empty list is written  $[]$ , while  $[a,b,c]$  is an abbreviation for

`[a | [b | [c | [] ] ] ]` (the list composed of elements `a`, `b` and `c`). Note that in PROLOG, as in pure logic programming language, there exists no list type, for which the binary function symbol `[ | ]` can be used for terms that are not lists. For example, we can also write `[a | f(a)]` which is not a list (because `f(a)` is not a list).

As the first example, let us consider the following program `member` which checks whether an element belongs to a given list:

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

The declarative reading of the program is immediate: Clause (1) consists of the terminal case in which the element that we are looking for (the first argument of the `member` predicate) is the head of the list (the second argument to the `member` predicate). Clause (2) instead provides the inductive case and tells us that `X` is an element of the list `[Y|Xs]` if it is an element of the list `Xs`.

Formulated in this way, the `member` program is similar to the one that we could write in any language that supports recursion. However, let us note that, unlike in the imperative and functional paradigms, this program can be used in two different ways.

The more conventional mode is the one in which it is used as a test. In a PROLOG system, once the previous program has been input, we have:

```
?- member(hewey, [dewey, hewey, louie]).
Yes
```

where `?-` is the abstract machine's prompt, to which we have added the goal whose evaluation we require. The next line contains the interpreter's answer. In this case, we have a simple "boolean" answer which expresses the existence of a successful computation of our goal. Moreover, as we know, we can also use the program to compute. For example, we can ask for the evaluation of:

```
?- member(X, [dewey, hewey, louie]).
X = dewey
```

The abstract machine returns `{X/dewey}` as the computed answer. We can also obtain more answers using the `“;”` command. When there are no more answer, the system replies `“no”`.

Finally, even if it is used in a rather unnatural fashion, we can use the first argument to instantiate the list that appears as the second argument. For example:

```
?- member(dewey, [X, hewey, louie]).
X = dewey
```

This possibility of using the same arguments as inputs or as outputs according to the way in which they are instantiated is unique to the logic paradigm and is due to the presence of unification in the computational model.

We can clarify this point further by considering the following `append` program which allows us to concatenate lists:

```
append([], Ys, Ys).                                     1
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).      2
```

In this case, too, the declarative reading is immediate. If the first list is empty, the results is the second list (clause(1)). Otherwise, (inductively), if  $Zs$  is the result of the concatenation of  $Xs$  and  $Ys$ , the result of the concatenation of  $[X|Xs]$  and  $Ys$  is obtained by adding  $X$  to the head of the list  $Zs$ , as indicated by  $[X|Zs]$ .

The normal use of the `this` program is illustrated by the following goal:

```
?- append([dewey, hewey], [louie, donald], Zs).
Zs = [dewey, hewey, louie, donald]
```

Moreover, we can use `append` also to know how to subdivide a list into sublists, something that is not possible in a functional or imperative program:

```
?- append(Xs, Ys, [dewey, hewey]).
Xs = []
Ys = [dewey, hewey];
Xs = [dewey]
Ys = [hewey];
Xs = [dewey, hewey]
Ys = [];
no
```

Here, as before, the “;” command causes the computation of new solutions.

As a third example, we give the definition of the `sublist` predicate (we used this in Sect. 12.1.1). If  $Xs$  is a sublist of  $Ys$ , then there exist another two (possibly empty) lists  $As$  and  $Bs$  such that  $Ys$  is the concatenation of  $As$ ,  $Xs$  and  $Bs$ . We note that this means that  $Xs$  is the suffix of a prefix of  $Ys$ . Hence using `append`, we can define `sublist` as follows:

```
sublist(Xs, Ys) :- append(As, XsBs, Ys), append(Xs, Bs, XsBs).
```

This program, combined with the `append` program and the `sequence` program of Sect. 12.1.1 allow us to solve the problem stated in Sect. 12.1.1. Note the conciseness and simplicity of the resulting program. Clearly other definitions of `sublist` are possible (see Exercise 8).

As a last example, let us consider a program that solves the classical problem of the Towers of Hanoi. We have a tower (in Computer Science terms, we would say a stack) composed of  $n$  perforated discs (of different diameters), arranged on a pole in order of decreasing diameter and we have 2 free poles. The problem consists of moving the tower to another pole, recreating the initial order of the discs. The following rules apply. The disks can be moved only from one pole to another. Only one disc at a time can be moved from one pole to another and the top disc must be taken from a pole. A disc cannot be put onto a smaller disc.

According to legend, this problem was assigned by the Divinity to the monks of a monastery near to Hanoi. There were three poles and 64 discs of gold. The solution to the problem would have signalled the end of the world. Given that the optimal solution requires time exponential in the number of discs, even if the legend comes true, we can still remain calm for a while:  $2^{64}$  is a large enough number.

The following program solves the problem for an arbitrary number of disc,  $N$ , and three poles called  $A$ ,  $B$  and  $C$ . It uses the coding of the natural numbers in terms of 0 and successor that we saw above.



---

```
hanoi(s(0), A, B, C [move(A,B)]).
```

```
hanoi(s(N), A, B, C, Moves):-
    hanoi(N, A, C, B, Moves1),
    hanoi(N, C, B, A, Moves2),
    append(Moves1, [move(A,B)|Moves2], Moves).
```

The call `hanoi(n, A, B, C, Move)`, where `n` is a (term which represents a) natural number, solves the problem of moving the tower of `n` discs from `A` to `B` using `C` as an auxiliary pole. The solution is contained in the `Move` variable which, when the computation terminates, is instantiated with the list containing move which contribute to the solution. Every move is represented by a term of the form `move(X, Y)` to indicate the move of the top disc of pole `X` to pole `Y`.

This amazingly simple program demonstrates the power of logic programming and recursive reasoning. Let us give its declarative interpretation. The first clause is clear: if we have just one disc, all we have to do is to move it from `A` from `B`. The reading of the second clause also is quite intuitive. If `Moves1` is the list of moves that solves the problem of moving a tower of `N` discs from `A` to `C` using `B` as an auxiliary pole and `Moves2` is the list of moves needed to move the tower of `N` disc from `C` to `B` using `A` as an auxiliary pole, to solve our problem in the case of `N+1` (or `s(N)`) discs, we have to do the following: first, execute all the moves in `Moves1`, then move from `A` to `B` (with the move `move(A, B)`) the `N+1`st disc in `A` which, being the largest, must be the last (at the bottom) in `B` as well. Finally, execute all the moves in `Moves2`. This is what is done by the `append` predicate which therefore will use the variable `Moves` to provide the solutions to this problem.

---

## 12.5 Extensions

Until now, we have seen pure logic programming and some, highly partial, aspects of PROLOG. In this last section, we will briefly outline some of the numerous extensions to the pure formalism which are used in real implemented logic languages. We defer to the next chapter an important extension of logic programming which is nowadays used in most PROLOG languages, namely constraint logic programming. In the bibliographical notes we have included some references for those wishing to learn more.

### 12.5.1 Prolog

PROLOG is a language that is much richer than it might appear from what has been said so far. Let us recall that this language differs from logic programming in its adoption of precise rules for selecting the next atom to be rewritten (left to right) and for selecting the next clause to use (top to bottom, according to the program text).

In addition to these, there are other important differences with the theoretical formalism. We will list some of these without pretending to be exhaustive.

## Arithmetic

A real language cannot allow itself the luxury of using a completely symbolic arithmetic with no predefined operators. In PROLOG, there exist therefore integers and reals (as floating point) as predefined data structures and various operators for manipulating them. These include:

- The usual arithmetic operators  $+$ ,  $-$ ,  $*$  and  $//$  (integer division).
- Arithmetic comparison operators such as  $<$ ,  $<=$ ,  $>=$ ,  $==$  (equal),  $= \setminus =$  (different).
- An evaluation operator called `is`.

Unlike other (function and predicate) symbols used in PROLOG, these symbols use infix notation for ease of use. There are however numerous delicate aspects that should be known to avoid simple mistakes.

First, comparison operators always require that their operands are ground arithmetic expressions (they cannot contain variables). Thus, while we have<sup>20</sup>:

```
?- 3*2 == 1+5
Yes
? 4 > 5+2
no
```

if we use terms that are not arithmetic expressions or which contain variables, error will be signalled, as in:

```
?- 3 > a
error in arithmetic expression: a is not a number
?- X == 3+5
instantiation fault.
```

The last example is particularly disturbing. The evaluation of the arithmetic expression  $3+5$  produces the value 8 which we might expect to be bound to the variable  $X$ . This cannot be done using  $==$ , nor can it be done using syntactical equality between terms as considered in the previous paragraph. If we write  $X = 3+5$ , indeed, the effect is that of binding  $X$  to the term  $3+5$  (rather than the value 8). To avoid this problem, the expression evaluation operator `is` is introduced. This operator allows us to obtain the effect we desire; `s is t` indeed *unifies* `s` with the *value* of the ground arithmetic expression `t` (if `t` is not a ground arithmetic expression, we have an error). The following are some examples of the use of `is`:

---

<sup>20</sup> At the prompt `?-` we find the query that we want to evaluate; the following line is the PROLOG interpreter's reply.

```

?- X is 3+5;
X = 8
?- 8 is 3 + 5
Yes
?- 6 is 3 * 3
no
?- X is Y*2
error in arithmetic expression: Y*2 is not a number

```

The use of `is`, necessary to be able to evaluate an expression, makes the use of arithmetic in PROLOG rather complicated. For example, given the following program:

```

evaluate(0,0).
evaluate(s(X), Val+1) :- evaluate(X, Val)

```

The goal `evaluate(s(s(s(0))), X)` does not compute the value 3 for `X`, as perhaps would have been expected, but the term `0+1+1+1` (see Exercise 9 at the end of this chapter, too).

## Cut

PROLOG provides various constructs for interaction with the abstract machine's interpreter so that the normal flow of control can be modified. Among these, one of the more important (and most discussed) is *cut*. This is an argument-free predicate, written as an exclamation mark, which allows the programmer to eliminate some of the possible alternatives produced during evaluation, with the aim of increasing execution efficiency. It is used when we are sure that, when a condition is satisfied, the other clauses in the program are no longer useful. For example, the following program computes the minimum of two value. If a comparison condition is true, the other is necessarily false, so we can use `!` to express the fact that once the first clause has been used, there is no need to consider the second.

```

minimum(X, Y, X) :- X < Y, !.
minimum(X, Y, Y) :- X > Y.

```

Alternatively, if we are interested only in testing whether a value appears at least once in a list, we can use the following modification to the `member` program:

```

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

```

In general, the meaning of `cut` is the following. Suppose that we have  $n$  clauses defining the predicate  $p$

```

p(S1) :- A1.
...
p(Sk) :- B, !, C.
...
p(Sn) :- An.

```

If, during the evaluation of the goal  $p(t)$ , we find the  $k$ th clause in the list being used, we have the following cases:

1. If an evaluation of  $B$  fails,<sup>21</sup> then we proceed by trying the  $k + 1$ st clause.
2. If, instead, the evaluation of  $B$  succeeds, then  $!$  is evaluated. It succeeds (it always does) and the evaluation proceeds with  $C$ . In the case of backtracking, however, all the alternative ways of computing  $B$  are eliminated, as well as are eliminated all the alternatives provided by the clauses from the  $k$ th to the  $n$ th to compute  $p(\tau)$ .

There is no need to say that cut, in addition to not being easy to use, eliminates a good part of the declarativeness of a program.

## Disjunction

If we want to express the disjunction of two goals,  $G1$  and  $G2$ , that is the fact that it is sufficient that at least one of the two succeeds, we can use two clauses of the form:

```
p(X) :- G1.
p(X) :- G2.
```

with the goal  $p(X)$ . The same effect can be obtained writing  $G1 ; G2$ , where we have used the predefined predicate “;”, which represents disjunction.

## If-then-else

The traditional construct from imperative languages:

```
If B then C1 else C2
```

is provided in PROLOG as a built-in with syntax  $B \rightarrow C1 ; C2$ . This construct is implemented using cut as follows:

```
if_then_else(B, C1, C2) :- B, !, C1.
if_then_else(B, C1, C2) :- C2.
```

## Negation

Thus far, we have seen that in the body of a clause, only “positive” atomic formulae can be used; in other words, they cannot be negated.<sup>22</sup> Often, however, it can also be useful to use negated atomic formulae. For example, let us consider the following program, flights:

```
direct_flight(bologna, paris).
direct_flight(bologna, amsterdam).
direct_flight(paris, bombay). 2
```

<sup>21</sup> Here,  $B$ , just like  $A_i$  and  $C$ , is considered as an arbitrary goal and not necessarily an atomic one.

<sup>22</sup> Here, we refer to the  $H : \neg A_1, \dots, A_n$  notation. If, instead, we consider a clause as a disjunction of literals, the atoms in the body are negated, given that the preceding representation is equivalent to  $\neg A_1 \vee \dots \vee \neg A_n \vee H$ .

```

direct_flight(amsterdam, moscow).           4
flight(X, Y):- direct_flight(X, Y).
flight(X, Y):- direct_flight(X, Z), flight(Z, Y). 6

```

Here, we have a series of facts defining the `direct_flight` predicate, which denotes the existence of a connection without stopover between two destinations. Then, we have the `flight` predicate which defines a flight, possibly with intermediate stops, between two locations. This will be a direct flight (clause 5) or rather a direct flight for a stopover different from the one that is the destination, followed by a flight from this stopover to the destination (clause 6). With this program, we can check the existence of a flight, or we can ask which destinations can be reached from a given airport:

```

?- flight(bologna, bombay)
Yes
?- direct_flight(bologna, moscow)
no
?- flight(bologna, X)
X = paris

```

However, we fail to express the fact that there exists only non-direct flights. Indeed, there is no way we can express that there *does not* exist a direct flight. To do this, we need negation. If we write:

```
indirect_flight(X, Y) :- flight(X, Y), not direct_flight(X, Y).
```

we mean to say that there exists an indirect flight between `X` and `Y` if there exists a flight (between `X` and `Y`) and there exist no direct flight. With this definition, we have:

```

?- indirect_flight(bologna, bombay)
Yes
?- indirect_flight(bologna, paris)
no

```

These results can be explained as follows. The PROLOG interpreter evaluates a goal such as `not G` by trying to evaluate the un-negated goal `G`. If the evaluation of this goal terminates (possibly after backtracking) with failure, then the goal `not G` succeeds. If, on the other hand, the goal `G` has a computation that terminates with success, then that of `not G` fails. Finally, if the evaluation of `G` does not terminate, then `not G` fails to terminate. This type of negation is called “negation as failure”,<sup>23</sup> in that it interprets the negation of a goal in terms of failure of the un-negated goal. Note that, because of infinite computations,<sup>24</sup> this type of negation differs from the negation in classical logic, given that the lack of success of `G` is not equivalent to the success of the negated version `not G`. For example, neither the goal `p` nor the goal `not p` succeed in the program `p :- p`.

<sup>23</sup> To be more precise, negation as failure, as defined in the theoretical model of logic programming, has a behaviour that is slightly different from that described because of the incompleteness of the PROLOG interpreter.

<sup>24</sup> And also non-ground goals, which we will not consider here.

## 12.5.2 Logic Programming and Databases

The `flight` program above indirectly indicates a possible application of logic programming in the context of databases.

A set of unit clauses, such as those defining the `direct_flight` predicate, is indeed, to all effects, the explicit (or extensional) definition of the relation denoted by this predicate. In this sense, this set of unit clauses can be seen as the analogue of a relation in the relational database model. To express a query, while in the relational model we would use relational algebra with the usual operations of selection, projection, join, etc., or more conveniently, a data manipulation language such as SQL, in the logic paradigm we can use its usual computational mechanisms. For example, to find out if there is a direct flight to Paris, we can use the query `direct_flight(X,paris)`. The predicates defined by the non-unit clauses such as `flight`, define relations in an implicit (or intensional) manner. They, indeed, allow us to compute new relations which are not explicitly stored, through the mechanism of inference. Using database terminology, we will say that these predicates define “views” (or virtual relations).

Note, moreover, that in the `flights` program, function symbols are not used. Indeed, if we want only to manipulate relations, as is the case in relational algebra, function symbols are not needed.

These considerations were crystallised in the definition of *Datalog*, a logic language for databases. In its simplest form, it is a simplified version of logic programming in which:

1. There are no function symbols.
2. Extensional and intensional predicates are distinguished, as are comparison predicates.
3. Extensional predicates cannot occur in the head of clauses with a non-empty body.
4. If a variable occurs in the head, it must occur in the body of a clause.
5. A comparison predicate can occur only in the body of a clause. The variables occurring in such a predicate must occur also in another atom in the body of the same clause.

The last two conditions deal with the specific evaluation rule which Datalog uses<sup>25</sup> and will here will be ignored. The distinction between extensional and intensional predicates corresponds to the intuitive idea that has already been discussed. Extensional predicates, as condition (3) implies, can be defined only by facts.

The `flights` program can therefore be considered as a Datalog program.

The reader who knows SQL or relational algebra will have no difficulty in understanding how the presence of recursion increases the expressive power of Datalog beyond that of these other formalisms. Our program, `flights`, provides us with

---

<sup>25</sup> Datalog adopts a kind of bottom-up evaluation mechanism for goals. This differs from the top-down one we saw for logic programming. The results obtained are, however, the same.

an example in this sense: the query `flight(bologna, X)` allows us to find all the destinations that can be reached from Bologna with an arbitrary number of intermediate stops. This query is not expressible in relational algebra or in SQL (at least, in its initial version) since the number of joins that we must create depends on the number of intermediate stops and, in general, not knowing how many of them there are (we are making no assumptions about how relations are structured), we cannot define such a number *a priori*.

---

## 12.6 Advantages and Disadvantages of the Logic Paradigm

The few examples given in this chapter should be sufficient to show that logic languages require a programming style that is significantly different from that of traditional languages. They also have a different expressive power. Clearly, since both PROLOG and the conventional programming languages are Turing-complete formalisms, we are not saying that logic languages allow “to compute more things” than common imperative languages. The expressive power to which we refer is of a pragmatic kind, as we will make clearer in this section.

First, it is worth repeating that logic languages, in a way analogous to functional languages, allow us express solutions even to very complex problems in a “purely declarative” way, and therefore in a simple and compact way. In our specific case, these aspects are further reinforced by three unique properties of the logic paradigm: (i) the ability to use a program in more than one way by transforming input arguments into outputs and viceversa; (ii) the possibility of obtaining a complex relationship between variables as a result, which implicitly expresses an infinite number of solutions (given by all the values of the variables that satisfy the relation); (iii) the possibility of reading a program directly as a logical formula.

The first characteristic derives from the fact that the basic computational mechanism, unification, is intrinsically bi-directional. This does not happen either with assignment in imperative languages, or with pattern matching in functional languages because both of these mechanisms presuppose a direction in variable bindings. This flexibility clearly allows us to use every program to its best, avoiding the duplication of code for modifications that are often only of a syntactic nature.

The second property is particularly important to all application areas in which a single solution, understood as a set of specific values of interesting variables, either cannot be obtained (because the data do not contain enough information), or is not interesting, or is particularly difficult to obtain computationally. This property will be more evident in the next chapter, when we will discuss constraint logic programs.

The third aspect, finally, at least in principle, makes it easier to verify the correctness of logic programs than imperative ones, where a logical reading of programs is possible but requires much more expressive (and complicated) tools. As already emphasised in Sect. 11.4, this aspect is essential to be able to obtain formal correctness-verifying tools that are easily usable on programs of significant size.

These three aspects, which we can summarise in the principle of “computation as deduction”, allow us to express in a natural fashion forms of reasoning that we encounter in artificial intelligence and knowledge representation (for example, in the context of the Semantic Web). The fact that control is handled entirely by the abstract machine through backtracking, allows to easily express search problems in a space of solutions.

Logic languages can also profitably be used as tools for the rapid prototyping of systems. Using PROLOG, it is possible to describe in a very short period and in a compact way, even very complex systems, with the advantage, as far as specification languages are concerned, that it can be executed.

Thus far, advantages. Beside them, however, we can record various negative aspects of logic languages, at least in their classical versions.

First, as already noted in this chapter, the control mechanism based on backtracking is of limited efficiency. Various expedients can be used to improve this aspect, be it at the level of constructs used in programs, or at the implementation level. For example, abstract machines specific to logic languages such as the WAM (Warren Abstract Machine) have been defined and have been variously optimised. However, the efficiency of a PROLOG program remains fairly limited.

The absence of types or modules is a second bad point, even if in this sense some more recent languages have proposed partial solutions (for example, the logic language, Mercury). From what was said in Chap. 8, it should be clear that the lack of an adequate type system makes program correctness checking relatively difficult, despite the possibility of declarative reading.

Also, the arithmetic constructs and, in general, built-ins in PROLOG certainly do not facilitate the correctness of programs. This is because their use is not easy due to various semantic subtleties. This is particularly evident in the control constructs, which frequently force us to choose between a clear program that is not efficient and an efficient one that is, however, hard to understand.

Finally, and for many reasons (not just a limited commercial interest), logic languages almost always have a programming environment that is rather deficient, where there are few of the characteristics available in the sophisticated environments for object-oriented programming.

---

## 12.7 Summary

In this chapter, we have presented the primary characteristics of the logic programming paradigm, a relatively recent paradigm (which has developed since the mid-1970s), which implements the old aspiration of seeing computation as a process that is entirely governed by logical laws. With only one chapter available to us, we have had to limit ourselves to introductory aspects, however what has been presented has enough formal precision to illustrate the computational model for logic programming languages. In particular, we have seen:



- Some syntactic concepts in first-order logic which are necessary for the definition of clauses and therefore logic programs.
- The process of unification which comprises the basic computational mechanism. This required a few concepts relating to terms and substitutions. We have also seen a specific algorithm for unification.
- How computation in a logic language works using a rule of deduction called SLD Resolution. In the text, we have chosen an approach that was inspired by a procedural reading of the clauses that demonstrate the similarity with “normal” procedures in imperative languages. Two boxes more formally introduced the concepts.

After some examples, we saw some important extensions to the pure formalism, in particular:

- Some specific characteristics of the PROLOG language. We did not cover meta-programming, higher-order programming or constructs that manipulate the program at runtime, all of which are very important.
- The idea of the use of logic languages in connection with databases.

For more information on those aspects of PROLOG that we have omitted or for a better introduction to Datalog, the reader is recommended to the literature listed below. On the other hand Constraint logic programs, an important extension, will be considered in the next chapter.

---

## 12.8 Bibliographical Notes

Herbrand’s ideas on unification are in his doctoral dissertation from 1930 [2]. The definition of the resolution rule and the first formalisation of the unification algorithm are in A. Robinson’s work [3]. The unification algorithm that we have presented was introduced in [4]. Major details of theory of unification can be obtained from various articles and texts, among which is [5].

The “historic” paper by R. Kowalski which introduced SLD Resolution and therefore the theory of logic programming is [6]. K. L. Clark’s results on correctness and completion are in [7].

There are many texts both on the theory of logic programming and on programming in PROLOG. Among the former, there is the text by Lloyd [8] and the more recent and detailed treatment by Apt [1]. For programming in PROLOG and for numerous (non-trivial) examples of programming, we advise the reader to consult the classic text by Sterling and Shapiro [9], from which we have taken the program that solves the problem of the Towers of Hanoi; there is also the book by Coelho and Cotta [10], from which we have taken the program in the example presented in Sect. 12.1.1.

Finally, as for Datalog and, more generally, the use of logic programs with databases, [11] can be consulted.

## 12.9 Exercises

1. State a context-free grammar which defines propositional formulæ (that is those obtained by considering only predicates of arity 0 and without quantifiers).
2. Assume that we have to represent the natural numbers using 0 for zero and  $s(n)$  for the successor of  $n$ . State what is the computed answer obtained by evaluating the goal  $p(s, t, X)$ , in the following logic program, where  $s$  and  $t$  are terms that represent natural numbers:

```
p(0, X, X).
p(s(Y), X, s(Z)):- p(Y, X, Z).
```

3. State what are the computed answers obtained by evaluating the goal  $p(X)$  in the following logic program:

```
p(0).
p(s(X)):- q(X).
q(s(X)):- p(X).
```

4. Given the following logic program:

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

State what is the result of evaluating the goal:

```
member(f(X), [1, f(2), 3]).
```

5. Given the following PROLOG program (recall that  $X$  and  $Y$  are variables, while  $a$  and  $b$  are constants):

```
p(b):- p(b).
p(X):- r(b).
p(a):- p(a).
r(Y).
```

State whether the goal  $p(a)$  terminates or not; justify your answer.

6. Consider the following logic program:

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

State whether the goal  $p(b)$  terminates or not; justify your answer.

7. Assuming that the natural numbers are represented using 0 for zero and  $s(n)$  for the successor of  $n$  and using a primitive `write(x)` that writes the term  $t$ , write a logic program that prints all the natural numbers.
8. Define the `sublist` predicate in a direct fashion without using `append`.
9. Write in PROLOG a program that computes the length (understood as the number of elements) of a list and returns this value in numeric form. (Hint: consider an inductive definition of length and use the `is` operator to increment the value in the inductive case.)

10. List the principle differences between a logic program and a PROLOG program.
11. If in a logic program, the order of the atoms in the body of a clause is changed, is the semantics of the program altered? Justify your answer.
12. If in PROLOG, the clause-selection rule were changed (for example, always selecting the lowest instead of the highest) would it change the semantics of the language? Justify your answer.
13. Give an example of a logic program,  $P$ , and of a goal,  $G$ , such that the evaluation of  $G$  in  $P$  produces a different effect when two different selection rules are used. (Suggestion: since for computed answers there is no difference in the use of different selection rules, consider what happens to computations that do not terminate and that fail.)
14. Informally describe a selection rule that allows us to obtain the least possible number of computations that do not terminate. (Suggestion: computations that do not terminate, by changing the selection rule, would become finite failures. Consider a rule that guarantees that all atoms in the goal are evaluated.)

---

## References

1. K.R. Apt, *From Logic Programming to Prolog* (Prentice Hall, 1997)
2. J. Herbrand, *Logical Writings* (Reidel, Dordrecht, 1971)
3. J.A. Robinson, A machine-oriented logic based on the resolution principle. *J. ACM* **12**(1), 23–41 (1965)
4. A. Martelli, U. Montanari, An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.* **4**, 258–282 (1982)
5. E. Eder, Properties of substitutions and unifications. *J. Symb. Comput.* **1**, 31–46 (1985)
6. R.A. Kowalski, Predicate logic as a programming language. *Inf. Process.* **74**, 569–574 (1974)
7. K.L. Clark, Predicate logic as a computational formalism. Technical Report Res. Rep. DOC 79/59, Imperial College (Department of Computing, London, 1979)
8. J.W. Lloyd, *Foundations of Logic Programming*, 2nd edn. (Springer, 1987)
9. L. Sterling, E. Shapiro, *The Art of Prolog* (MIT Press, 1986)
10. H. Coelho, J.C. Cotta, *Prolog by Example* (Springer, 1988)
11. S. Ceri, G. Gottlob, L. Tanca, *Logic Programming and Databases* (Springer, 1989)

## 13.1 Constraint Programming

In the previous two chapters we have seen, with the two declarative paradigms there discussed, two major steps toward the “holy grail of programming: the user states the problem, the computer solves it” [1]. Constraint programming provides a further, significative step in this direction.

Indeed, as we have already seen in the previous chapters, the very idea of declarative programming is that the programmer specifies *what* has to be done, and the programming language system autonomously discovers *how*. An essential component of this declarative approach is the linguistic tool that the programmer can use to describe the *what*, that is, the problem to be solved. With functional languages this tool is the function. This is certainly a very general concept useful for describing many kind of computational problems, however it suffers the limitation of “directionality”: consider simply the very definition of a function as a rule which associates to an element of the domain at most one element of the codomain; from this definition it is clear that if we want to change the direction of application we need to use another function (which, of course, is the inverse). As an example, consider the famous definition of speed by Galileo Galilei described by the equation

$$v = \frac{d}{t} \quad (13.1)$$

where  $v$  is speed,  $d$  is distance and  $t$  is time. If, for a fixed speed  $v$ , we want to know the distance which is covered in time  $t$  we can derive from the previous equation the function

$$Distance(t) = v \times t.$$

However, if we want to derive (for the same fixed speed  $v$ ) the time needed to cover the distance  $d$  we need a different function

$$Time(d) = \frac{d}{v}$$

which is the inverse of the function *Distance*. Both these two functions are contained in Eq. 13.1, however if we use a functional language we cannot have a unique construct for denoting both of them.

The same problem is shared by imperative languages, since also assignment has a directional nature: if we want to compute the distance we need the assignment

$$D = V * T$$

(where  $D$ ,  $V$  and  $T$  are variables with the obvious meaning) while if we want to compute the time we need a different assignment command:

$$T = D / V.$$

The key observation here is that Eq. 13.1 expresses a *relation*<sup>1</sup> or, in other terms, a *constraint* on the quantities  $v$ ,  $d$  and  $t$ , which relates their values by a specific mathematical law. In order to overcome the limitation shown by the previous example we would then need a language which allows us to use relations, or constraints, as “first class citizens”, meaning that relations can be written, manipulated, computed and provided as results of computation: this is exactly what a constraint programming language provides. By using such a language one could simply write the constraint

$$D == V * T$$

where  $==$  means equality (and not assignment), and then we could obtain the value of one variable once the values of the two others are provided. But, more than that, from such a constraint one could also obtain as the result of the computation a new constraint (i.e. a new relation) rather than a single value. For example, if we add to the previous constraint the following one

$$V == 2$$

from these two constraints we could obtain as the result of the computation the constraint

$$D == 2 * T$$

which, being a relation, provides a result describing an (infinite) set of values rather than a single value.

This ability to manipulate relations and obtain them as results of computation is essential in many applications, especially in artificial intelligence: many problems have a combinatorial nature which allow us to easily express them in terms of constraints stating different relations among different objects. Of course, once the problem is expressed as a set of constraints, we need a suitable tool which allows us to simplify them and to perform some computations in order to obtain a solution. Such a tool is called *constraint solver* (or solver for short) and its nature depends on the specific domain on which the constraints are defined. For example, if we consider conjunctions of linear equations as constraints and real numbers as the domain of

---

<sup>1</sup> For the forgetful reader we recall that an  $n$ -ary relation on  $n$  sets  $D_1, D_2, \dots, D_n$  is any subset of the Cartesian product  $D_1 \times D_2 \times \dots \times D_n$ .

computation, then we could use the Gauss-Jordan method as the constraint solver.

The careful reader has surely noticed that, in the previous chapter, we have already seen a constraint programming paradigm where relations can be used, manipulated and computed. Indeed, if in a logic programming language we write the equations

$$X = t, Y = f(a)$$

we are considering a particular class of constraints, namely equalities over the Herbrand Universe.<sup>2</sup> More precisely, in the previous equations, the predicate symbol  $=$  is interpreted as syntactic equality over the Herbrand Universe, while the comma indicates logical conjunction. These specific constraints then refer to a specific solver, the unification algorithm, which, as we have seen, is integrated in the logic programming language.

The general idea of constraint programming is to consider further domains beyond the Herbrand Universe, both numerical and symbolic ones, with the related solvers, and to integrate them with existing languages.

The development of appropriate, efficient constraint solvers is a central topic in constraint programming which, however, is beyond the scope of this book. We only notice here that they are based on some *domain specific methods* which are related to the specific domain for the values used by the constraint (for example, methods for solving linear equations, packages for linear programming, unification algorithm, etc.). These specific methods are integrated with *general methods* which aim to reduce the search space (for reaching a solution) and use both *constraint propagation* techniques based on *local consistency* and specific search methods.

Constraint propagation is a key technique for constraints on finite domains. Its idea is to use existing constraints to perform some inference in order to reduce the number of admissible values for the existing variables. In modern constraint programming systems, these techniques are thoroughly integrated with the features offered by the host programming language, thus allowing the programmer to model the problem to be solved at a very high level, while the underlying solving machinery provide powerful tools for solving the problem. In most cases, the main difficulty of the programmer in solving a concrete problem is then finding an appropriate description, or model, for it.

As previously mentioned, in the following, we will consider the integration of constraints both with logic programming and with imperative languages, since these offer the mainly used systems today.

### 13.1.1 Types of Problems

Constraints arise in many real life applications and have been used to model and solve a variety of problems since the '60s. The following is an incomplete list which

---

<sup>2</sup> Recall that this is the set of ground terms.

includes both research and commercial applications of constraint programming and should provide an idea of the practical relevance of this technology:

- analysis, simulation, verification, diagnosis of software, hardware, and industrial processes (e.g., circuit verification in major producers of electronic components);
- configuration (e.g., software configuration for mobile phones);
- design (e.g., aircraft cabin layouts);
- financial applications (e.g., investment portfolio management and optimisation for banks);
- decision support systems (e.g., systems for diagnosis in medicine);
- planning (e.g., harbour resource planner, short-term production planning in car manufacturer, parcel delivery in logistics);
- research in molecular biology, biochemistry, and bio-informatics (e.g., protein folding, genomic sequencing);
- risk management (e.g., systems for dams management);
- scheduling and timetabling (crew and aircraft scheduling in airlines, timetables of railways).

All these applications involve different domains, such as blocks, booleans, finite domains, integer numbers, intervals, real numbers, strings, terms, trees, temporal intervals and others.

Particularly relevant for many applications are finite domains which, despite the apparent simplicity—being finite the possible values, variables, and constraints, the solution can always be found in a finite time—often involve problems which are computationally very difficult—many problems are NP-complete; just consider Boolean satisfiability.

Some practical problems which can be formulated by means of constraints require us to find just one solution, no matter how “good” it is. This class of problems is usually formalised by using the notion of Constraint satisfaction problem (CSP for short) as follows.

**Definition 13.1** (CSP) A Constraint Satisfaction Problem (CSP) consists of

1. a finite set  $V = \{x_1, \dots, x_n\}$  of variables;
2. a set  $D = \{D_1, \dots, D_n\}$  of domains where, for  $i \in [1, n]$ , the variable  $x_i$  assumes values in the domain  $D_i$ ;
3. a set  $C$  of constraints which specify the allowed values for variables (i.e., relations on the domains).

A *solution* is a tuple  $(d_1, \dots, d_n)$  of values which satisfies all the constraints in  $C$  where, for  $i \in [1, n]$ ,  $d_i \in D_i$  is the value of  $x_i$ .

Clearly different languages and systems provide different syntax for expressing constraints, and a precise notion of satisfiability can be defined on the basis of such a syntax (in the next section we will see a notion based on first order logic). However,

we can easily define what it means for a tuple of values to satisfy a constraint by using its definition as a relation: if the constraint  $c$  is a relation on  $D_1, \dots, D_n$  then  $(d, \dots, d_n) \in D_1 \times \dots \times D_n$  satisfies  $c$  iff  $(d_1, \dots, d_n) \in c$ . So, for example, the pair of values (10,5) for the pair of variables  $(x, y)$  satisfies the constraint  $x > y$  and does not satisfy the constraint  $x = y$ .

**Example 13.1** A classic CSP problem is the Map coloring. Consider a map of central Italy describing the regions Toscana, Marche, Umbria, Lazio, Abruzzo. The task is to color each region either red, green or blue in such a way that neighboring regions have different colors. The problem can be represented by using the variables  $V = \{T, M, U, L, A\}$  for representing the regions and the domain  $D = \{\text{red}, \text{green}, \text{blue}\}$ . If we look at the map, the constraints requiring neighboring regions having different colors can then be simply expressed as

$$C = \{T \neq M, T \neq U, T \neq L, M \neq L, M \neq U, M \neq A, U \neq L, L \neq A\}.$$

For some applications finding just one solution is not enough: among all the possible solutions to our constraints, we want to find one which maximises (or minimises) a given cost function. Consider for example the famous travelling salesperson problem: among all the different routes between two destinations, we want to find one which minimises the distance. This class of problems, which involve an objective (or cost) function which should be optimized, are called Constraint optimisation problems (COP for short) and can be defined as follows.

**Definition 13.2 (COP)** A Constraint Optimisation Problem (COP) consists of a CSP defined on the variables  $\{x_1, \dots, x_n\}$ , the constraints  $C$  and the domains  $\{D_1, \dots, D_n\}$ , together with an objective function  $f : D_1 \times \dots \times D_n \rightarrow D$ .<sup>3</sup>

A solution to the constraint optimisation problem is a solution to the CSP which optimize the function  $f$ .

We will consider constrained optimisation problem later in Sect. 13.4, when discussing constraints in the setting of imperative languages.

---

## 13.2 Constraint Logic Programs

We now discuss *Constraint logic programming*, an extension of logic programming which adds sophisticated constraint-satisfaction to the mechanisms we have seen in the previous chapter. The resulting paradigm is very interesting for practical application and *Constraint logic programs*, or CLP, today are used in many different areas, particularly in Artificial Intelligence. It is worth noting that most of the current Prolog

---

<sup>3</sup> Typically  $D$  is the set of real numbers.



implementations include an extension which allows for constraints, so most Prolog systems today are CLP languages.

A constraint can be seen as a particular first-order logic formula (normally we use a conjunction of atomic formulæ) which uses only predefined predicates. As previously mentioned, logic programming uses a particular kind of constraints, namely conjunctions of equations over the Herbrand Universe.

The idea of constraint logic programming is that of replacing the Herbrand Universe by another computational domain which usually is symbolic, but can also be arithmetic in some cases, and which is suitable for the application area of interest. Constraints, rather than equations between ground terms, define specific relations over the values in the new domain under consideration. Correspondingly, the basic computational mechanism will no longer be the solution of equations between terms (using unification) but will use a constraint-solving mechanism, that is an appropriate algorithm for determining solutions to the constraints to be solved.

For example, if we are interested in linear inequalities over real numbers, we could use the predicate (constraint) symbols  $\geq$  and  $\leq$  with the expected meaning. It is important to observe that this meaning, being the “expected” one, does not need to be defined by specific clauses in our program. Indeed, it is rather fixed a priori by a suitable logical theory that provides the correct axiomatisation and that we assume given. This also means that, when it comes to solving a constraint which uses the predicates  $\geq$  and  $\leq$ , it is not sufficient to use the rules of our program, but we need to use also specific algorithms borrowed from mathematics which allow us to manipulate and solve inequalities according to the meaning provided by the chosen axiomatisation. This point will be further clarified in the next subsection, when defining the semantics of CLP.

The principal advantage of this approach should be clear: we can integrate into the logic paradigm (in a semantically clean fashion) very powerful computational mechanisms that were developed in other contexts for specific domains (such as linear programming, operation research, etc.)

### 13.2.1 Syntax and Semantics of CLP

To provide some more precise details on Constraint logic programming we now define its syntax and semantics.

Let us start by defining a syntax for constraints.

**Definition 13.3** (*Constraint*) Assume given a signature  $(\Sigma, \Pi)$  augmented with constraint symbols, where the set of predicates  $\Pi$  contains a set  $\Pi_c$  of constraint symbols, which are different from the other predicate symbols. Assume also that  $\Pi_c$  contains  $=$ . The constraints over the signature with terms  $(\Sigma, \Pi)$  are defined as follows:

1. if  $t_1, \dots, t_n$  are terms over the signature  $\Sigma$  and  $c \in \Pi_c$  is a constraint symbol of arity  $n$ , then  $c(t_1, \dots, t_n)$  is a constraint;
2. *true* and *false* are constraints;
3. if  $C$  and  $D$  are constraints then  $(C \wedge D)$  (also written  $C, D$ ) is a constraint.<sup>4</sup>

The syntax of CLP is a straightforward modification of that one of logic programming, provided in Definition 12.3, to take into account constraints.

**Definition 13.4** (*Constraint logic program, CLP*) Let us assume given a signature  $(\Sigma, \Pi)$  augmented with constraint symbols, where  $\Pi_c \subseteq \Pi$  is the set of constraint symbols. Let  $H, A_1, \dots, A_n$  be atomic formulae defined on the signature  $(\Sigma, \Pi \setminus \Pi_c)$  and let  $C$  be a constraint on  $(\Sigma, \Pi_c)$ .

A CLP clause is a formula of the form:

$$H :- C, A_1, \dots, A_n.$$

If  $n = 0$ , the clause is said to be a *unit*, or a fact, and the symbol  $:-$  is omitted (but not the final full stop). A constraint logic program is a set of CLP clauses. A query (or goal) is a sequence of constraints and atoms  $C, A_1, \dots, A_n$ .

As it appears from the previous definition, a CLP program is simply a logic program where each clause can contain a constraint (note that the constraint  $C$  in the clause can be omitted, syntactically represented by choosing for  $C$  the *true* constraint).

The idea of a CLP computation is that constraints, introduced by clauses, are accumulated during the computation and are then simplified and solved by suitable derivation steps which use an underlying constraint solver. This is analogous to the case of logic programming, provided that rather than explicit substitutions one uses equation on terms, which are then solved by the unification algorithm.

To formalise this intuitive notion of semantics, we deviate from the previous chapter, and we use the notion of *transition system* which has been introduced in Chap. 2 as a general tool for defining the operational semantics of programming languages.

We first assume given a consistent first-order constraint theory (CT) which defines the meaning of the constraint predicate symbols. Such a theory includes Clark equality theory (CET) defining the meaning of  $=$  as syntactic equality over Herbrand Universe.<sup>5</sup> Hence, we introduce a specialised transition systems for CLP, defined as follows.

<sup>4</sup> Here, we have considered only conjunction, following the practice of modern CLP languages. In principle, one could consider also other logical connectives.

<sup>5</sup> In some cases we will overload the symbol  $=$ , and we will use it also with a different meaning (for example, equality on numbers). These different uses will be explicitly mentioned.

**Definition 13.5** (CLP transition system) A state is defined as a pair  $\langle G; C \rangle$  where  $G$  is a goal and  $C$  is a constraint.<sup>6</sup>

A CLP transition system (in the following transition system for short) for the program  $P$  is a pair  $(S; \mapsto)_P$  where  $S$  is a set of states and  $\mapsto$  is a binary relation over states, also called *transition relation*, defined inductively by the following transition rules:

**Unfold** If  $H :- C, A_1, \dots, A_n$  is a fresh variant of a clause in  $P$  and

$$CT \models \exists ((A = H) \wedge C)$$

then

$$\langle A \wedge G; D \rangle \mapsto \langle G \wedge C \wedge A_1 \wedge \dots \wedge A_n; (A = H) \wedge D \rangle.$$

**Failure** If there is no (fresh variant of) clause  $H :- C, A_1, \dots, A_n$  in  $P$  such that

$$CT \models \exists ((A = H) \wedge C)$$

then

$$\langle A \wedge G; D \rangle \mapsto \langle A \wedge G; false \rangle.$$

**Solve** If  $CT \models \forall ((C \wedge D_1) \leftrightarrow D_2)$  then

$$\langle C \wedge G; D_1 \rangle \mapsto \langle G; D_2 \rangle.$$

A few explanations are in order, here.

In this operational semantics a state consist of the goal which has still to be evaluated ( $G$ ), together with the (conjunction of) all the constraints which have been computed so far ( $D$ ). Hence in the rule “unfold”, analogously to the logic programming case, once we have selected an atom  $A$  in the current goal  $A \wedge G$ , a computation (or derivation, or reduction, or unfolding) step is possible if we find a suitable clause in the program whose head unifies with  $A$ , under the value for the variables defined by the constraint  $C$  in the clause. This is exactly what the formula  $CT \models \exists ((A = H) \wedge C)$  says, where the theory  $CT$  has the function of specifying the meaning of the constraints.<sup>7</sup> As in the case of logic programming, the atom  $A$  to be rewritten can be any atom in the current goal (we have used here a simplified notation, which avoids mentioning the rules obtained by considering the commutative nature of  $\wedge$ ).

Rule “failure” describes what happens when we cannot find in the program such a clause: the computation fails, as it is for logic programs.

Rule “solve” instead introduces the major innovative feature of CLP, namely the constraint solving operation. As previously mentioned, different constraint domains, and therefore different theories  $CT$ , use different solvers with different internal

<sup>6</sup> For better clarity, in states we use the symbol  $\wedge$  to denote conjunction rather than the comma, which appear in the syntax of programs.

<sup>7</sup> Note that the case of logic programming could be obtained from this general case by simply considering the only theory CET rather than a generic CT.

computations and a different syntactic manipulation of the constraints. However, all the different constraint system and solvers share the purpose of simplifying the constraints in order to reduce the search space and to obtain a solution. Certainly all these simplifications must be logically sound, that is, must reduce a constraint to another one, possibly simpler and closer to the solution, while preserving the logical meaning of the constraint itself. Hence, rule “solve” which states that, whenever we select in the current goal  $C \wedge G$  a constraint  $C$  for evaluation, we can perform a derivation step which modifies (and simplifies) the constraint  $C$  and the constraint computed so far  $D_1$  thus producing a new constraint  $D_2$ , provided that this new constraint is still logically equivalent to  $C \wedge D_1$ , as expressed by the condition  $CT \models \forall ((C \wedge D_1) \leftrightarrow D_2)$ .

Once we have introduced the transition systems, the definition of computation (or derivation) is pretty standard<sup>8</sup>:

**Definition 13.6** (*CLP Computations*) Assume given a transition system  $(S; \mapsto)_P$  for the CLP program  $P$ . An *initial state* has the form  $\langle G; true \rangle$ , a *successful final state* has the form  $\langle true; C \rangle$  with  $C$  different from *false*, while a *failed final state* has the form  $\langle G; false \rangle$ .

A *finite computation* (or derivation) is a sequence of states  $S_1, S_2, \dots, S_n$  such that  $S_i \mapsto S_{i+1}$  for  $i \in [1, n - 1]$ .

A *successful computation* has the form  $S_1, S_2, \dots, S_n$  where  $S_1$  is an initial state and  $S_n$  is a successful final state. In this case, if  $S_1$  is the state  $\langle G; true \rangle$  and  $S_n$  is the state  $\langle true; C \rangle$  we say that  $C$  is the *computed constraint* for the goal  $G$  in the program  $P$ .

A *finitely failed computation* has the form  $S_1, S_2, \dots, S_n$  where  $S_1$  is an initial state while  $S_n$  is a failed final state.

Note that, as already mentioned, in CLP an answer is a constraint and, as a such, it can provide a set of possible values rather than a single value. Next subsection will provide a significative example. Moreover, also in the case of CLP we have two forms of non-determinism. These are completely analogous to those of logic programming and therefore their discussion is omitted.

---

## 13.3 Generate and Test Versus Constraint and Generate

So far we have seen CLP as a smooth extension of logic programming, indeed many implemented Prolog system today includes libraries and support for constraint programming. There is however a major difference between CLP and (traditional) Prolog,

---

<sup>8</sup> Here, we consider only finite computations. Of course, infinite computations can be defined as well.

which affects the style of programming and, most important, the efficiency of the programs.

As we have seen in the previous chapter, arithmetics is rather clumsy in Prolog, in particular many arithmetic built-ins require that their arguments are ground in order to be evaluated. For example, while the evaluation of goal  $p(Z)$  with the program

```
p(X):- Y =1, X is Y + 1.
```

produces that answer  $Z=2$  as expected, the same goal in the program

```
p(X):- X is Y + 1, Y =1.
```

produces the error

```
Arguments are not sufficiently instantiated
```

because when the predicate `is` is evaluated the variable  $Y$  is not yet bound to 1.

This feature of arithmetic built-ins is the main reason for the “generate and test” style of programming that is common with standard Prolog: when we have to solve a problem where among the many different values for some variables we must select those which satisfy a given set of constraints, being the constraints expressed in terms of built-ins the only possible approach is first *generating* a selection of values for the variables and then *testing* whether those values satisfy the constraints. If not, new values are generated and tested and so on, until either we find a solution or we terminate the available values. This methodology uses somehow a “brute force” approach which does not exploit the power of constraint programming and, in particular, the constraint propagation techniques which are behind several solvers (especially on finite domains).

Indeed, as mentioned before, in many cases the constraint themselves can be used to guide the selection of the right values, thus pruning in a considerable way the solution search space. The general idea of the “constraint and generate” style is then to *constraint* first the admissible values by using the problem constraints, and then *generate* the values chosen from the set of values which has been already reduced by the constraints. This approach is possible because, differently from Prolog, in CLP the computation proceed by accumulating constraints which may include non instantiated variables. The pruning of the search space obtained with the constraint and generate approach makes in general the computation much more efficient than with the generate and test approach and, in many cases, transforms a computationally unsolvable problem into a feasible one.

As a simple example illustrating these two different programming methodologies, consider the crypto-arithmetic problem consisting in solving the equation

```

  S E N D
+  M O R E
-----
= M O N E Y

```

where each letter represents a different digit.

This problem can be solved by a simple CLP program which exploits the constraint and generate methodology. The program, as well the other ones in the following, is derived from [2] and can be executed on SWI Prolog.

```
:- use_module(library(clpfd)).

send([S,E,N,D,M,O,R,Y]) :-
    gen_domains([S,E,N,D,M,O,R,Y],0..9),
    S #\= 0, M #\= 0,
    all_distinct([S,E,N,D,M,O,R,Y]),
    1000* S + 100* E + 10*N + D
    + 1000* M + 100* O + 10*R + E
    #= 10000* M + 1000* O + 100* N + 10*E + Y,
    labeling([], [S,E,N,D,M,O,R,Y]).

gen_domains([],_).
gen_domains([H|T],D) :- H in D, gen_domains(T,D).
```

Previous program uses the library `clpfd` which allows us to use a finite domain solver and several related functionalities. In particular, the predicate `in\2` checks that the first argument is an element of the domain appearing as second argument, where the range domain `0..9` denotes all the integer numbers greater or equal to 0 and smaller or equal to 9.

The program solves the problem by using the usual declarative approach. After having introduced the domain `0..9` for all the variables in the list `[S,E,N,D,M,O,R,Y]` (by using the predicate `gen_domains`), the constraints which model the problem at hand in term of equations and inequations are introduced: The constraints

$$S \neq 0, M \neq 0$$

require that the most significative digit cannot be zero, while the equation

$$\begin{aligned} &1000 * S + 100 * E + 10 * N + D \\ &+ 1000 * M + 100 * O + 10 * R + E \\ &\#= 10000 * M + 1000 * O + 100 * N + 10 * E + Y, \end{aligned}$$

provides the arithmetic formulation of the problem itself. The fact that different letters correspond to different digits is guaranteed by the global constraint

```
all_distinct([S,E,N,D,M,O,R,Y]).
```

However, using only these constraints the constraint solver cannot enforce a single solution since some variables have a range of several admissible values. In order to obtain a specific value for the variables then, we need to use the `labeling\2` built-in which essentially set the variables appearing in its second argument to all possible values compatible with the domain, backtracking in case the set values do not satisfy the constraints.<sup>9</sup>

The constraint and generate approach here emerges from the fact that the setting of values, done by the

```
labeling([], [S,E,N,D,M,O,R,Y])
```

<sup>9</sup> The first argument of `labeling\2` specifies some options on the generation of the values.

call, is done *after* the introduction of the constraints in the computation.

It is useful to compare the execution time of previous program with that one of the following programs, which implements a generate and test approach, since the values for the variables (through the call to the `labeling` built-in) are now set *before* the introduction of the constraints.

```
:- use_module(library(clpfd)).

send([S,E,N,D,M,O,R,Y]) :-
    gen_domains([S,E,N,D,M,O,R,Y],0..9),
    labeling([], [S,E,N,D,M,O,R,Y]),
    S #\= 0, M #\= 0,
    all_distinct([S,E,N,D,M,O,R,Y]),
    1000*S + 100*E + 10*N + D
    + 1000*M + 100*O + 10*R + E
    #= 10000*M + 1000*O + 100*N + 10*E + Y.

gen_domains([],_).
gen_domains([H|T],D) :- H in D, gen_domains(T,D).
```

By using the `time` functionality of SWI Prolog one can check that while the execution of the goal

```
?- send([S,E,N,D,M,O,R,Y]).
```

in the first program produces the solution

```
S= 9, E=5, N=6, D=7, M=1, O=0, R=8, Y=2
```

almost instantaneously (on the online SWI Prolog system), the same goal in the second program produces the answer

```
Time limit exceeded
```

and fails to find the solution. This provide a clear evidence of the computational advantage of the constraint and generate approach.

### 13.3.1 A Further Example

To get an idea of the possibilities of constraint logic programming, consider the problem of defining the amount of the instalments of a loan. The variables involved in this problem are the following: *F* is the requested financing, or the initial sum loaned, *NumR* is the number of the instalment, *Int* is the amount of interest, *Rate* is the amount of a single instalment and, finally, *Deb* is the remaining debt. The relation between these variables is intuitively the following. In the case in which there is no amount paid back (or that  $\text{NumR} = 0$ ), clearly the debt is equal to the initial financing:

```
Deb = Fin.
```

In the case of the payment of a single instalment, we have the following relation:

$$\text{Deb} = \text{Fin} + \text{Fin} * \text{Int} - \text{Rate}$$

Here, the remaining debt is the initial sum to which we have added the interest accrued and have subtracted the amount repaid in an instalment. In the case of two instalments, matters are more complicated because interest is calculated on the remaining debt and not on the initial sum. The remaining debt, then, becomes the new financing. Using the `NuFin1` and `NuFin2` variables to indicate this new value of financing, we then have therefore the relation:

$$\begin{aligned}\text{NuFin1} &= \text{Fin} + \text{Fin} * \text{Int} - \text{Rate} \\ \text{NuFin2} &= \text{NuFin1} + \text{Fin} * \text{Int} - \text{Rate} \\ \text{Deb} &= \text{NuFin2}\end{aligned}$$

In the general case, we can reason recursively. We have therefore the following constraint program, which we will call `loan`:

```
:- use_module(library(clpr)).

loan(Fin, NumR, Int, Rate, Deb) :-
    {NumR = 0,
     Deb = Fin}.

loan(Fin, NumR, Int, Rate, Deb) :-
    {NumR >= 1,
     NuFin = Fin+Fin*Int-Rate,
     NuNumR = NumR-1},
    loan(NuFin, NuNumR, Int, Rate, Deb).
```

Understanding this program, given what we have already said, should not be too difficult. The first line tell us that here we are using a CLP system defined over the domain of real numbers, with the associated solver and functionalities.<sup>10</sup>

Previous program can be used, just like logic programs, in many ways. For example, if we take out a loan of 1000 Euros and having paid 10 instalments of 150 Euros each at 10% interest, we can use it to find out what the remaining debt is. The goal to solve this problem is:

```
?- loan(1000, 10, 10/100, 150, Deb).
```

When evaluated (with the previous program) this goal produces the result `Deb = 203.128`<sup>11</sup>

We can also use the same program in an inverse fashion. That is, rather finding out what the requested funding was, knowing that we have paid the same 10 instalments at 150 Euros each, again at 10% interest, but with residual debt of? 0. The goal in this case is:

<sup>10</sup> The `loan` program is written using the syntax required by the `clpr` library as used in SWI Prolog. The curly brackets in the rules are required by the syntax of `clpr` on SWI Prolog to separate constraints.

<sup>11</sup> Here and in the following the results are obtained on SWI Prolog online.



```
?- loan(Fin, 10, 10/100, 150, 0).
```

This yields the result  $\text{Fin} = 921.685$ .

Finally, we can also leave more than one variable in the goal, for example to determine the relation that holds between financing, instalment and debt, knowing that we pay 10 instalments at 10% interest. We can therefore formulate the query:

```
?- loan(Fin, 10, 10/100, Rate, Deb)
```

With appropriate assumptions on the constraint solver<sup>12</sup> we obtain the result:

```
Deb = 2.593 * Fin - 15.937 * Rate.
```

That is—and this is unique to the various paradigms that we have seen so far—it allows us to obtain a relation on the numerical domains as a result.

## 13.4 MiniZinc

In this section we briefly outline the main features of MiniZinc [3], a free and open-source constraint modeling language which can be used to model constraint satisfaction and optimization problems in a high-level and solver-independent way. Here we aim only at giving a taste of how constraint programming can be embedded in the context of an imperative setting, without any claim of completeness.

MiniZinc models<sup>13</sup> are contained in files having the extension `.mzn` and are compiled to FlatZinc, a solver input language that is understood by many different solvers. Data can be contained in the MiniZinc models or in separate files having the extension `.dzn`. Some of the examples of MiniZinc models that we will use in the following are from the Handbook of MiniZinc [4].

### 13.4.1 A MiniZinc CSP Model

A MiniZinc model is composed of four main parts that we will now illustrate by using the same crypto-arithmetic example seen in Sect. 13.3.

First, the libraries that will be used in the model are introduced by using the `include` command. For example, the line

```
include "globals.mzn";
```

<sup>12</sup> Systems defined over real numbers, can require fairly substantial modifications even if the basic idea is always the same. In particular, the possibility of obtaining the results just discussed depends a great deal on the power of the constraint solver adopted by the CLP language.

<sup>13</sup> Sometime the terminology “program” is also used instead of “model”. The latter however is more correct, since MiniZinc models lack many traditional parts of standard programs.

is used to include all the global constraints defined in the file `globals.mzn`. One of such constraints is `alldifferent`, which has exactly the same meaning seen before in Sect. 13.3.

Second, variables and parameters are specified with their types, as in

```
var 1..9: S;
var 0..9: E;
var 0..9: N;
var 0..9: D;
var 1..9: M;
var 0..9: O;
var 0..9: R;
var 0..9: Y;
```

where the integer range types `1..9` and `0..9` (with the obvious meaning) are used for the (global) variables `S`, `E`, ... This is similar to the declarations of variables that one find in the imperative languages, however there is an important difference: in MiniZinc we have *decision variables* (also called variables tout court) and *parameters*. Decision variables, which are declared by using the keyword `var` (as before), are normally assigned by the solver when searching for a solution or by the user with an explicit constraint of the form

```
constraint <variable> = <expression>;
```

Parameters are declared by either using the keyword `par` or omitting any specific keyword. A parameter can be assigned explicitly a fixed value in the model or in the data file, very much as for a variable in imperative programs with the notable difference that a parameter cannot be assigned twice.

Third we introduce in the model the constraints, which have the form

```
constraint <Boolean expression>;
```

where in Boolean expression we can use the usual arithmetic operators (`+`, `-`, `*`, ...), comparison predicates (`>`, `<`, ...), logical connectives (`¬`, `∧`, `∨`, `→`, ...) global constraints and other constraints. In our example we can use the constraints

```
constraint 1000 * S + 100 * E + 10 * N + D
          + 1000 * M + 100 * O + 10 * R + E
          = 10000 * M + 1000 * O + 100 * N + 10 * E + Y;
```

```
constraint alldifferent([S,E,N,D,M,O,R,Y]);
```

with the obvious meaning.

Finally we must include in our model one (and only one) “solve” item which allow us to specify whether we are solving a satisfiability problem, and therefore we are just looking for one solution, or we are solving and optimization problem, and therefore we are looking for the optimal solution. Solve item has then one of the following three forms:

```
solve satisfy;
solve maximize <arithmetic expression>;
solve minimize <arithmetic expression>;
```

In our example we simply need one solution, so we write

```
solve satisfy;
```

To present the results of the model execution in a more nice way we can use output items of the form:

```
output [<string expression>, ... , <string expression>];
```

where the string expressions provide some specific instructions for presenting the results.

Putting together all the components of our example, and adding the output part, we obtain the model

```
include "globals.mzn";

var 1..9: S;
var 0..9: E;
var 0..9: N;
var 0..9: D;
var 1..9: M;
var 0..9: O;
var 0..9: R;
var 0..9: Y;

constraint 1000 * S + 100 * E + 10 * N + D
           + 1000 * M + 100 * O + 10 * R + E
           = 10000 * M + 1000 * O + 100 * N + 10 * E + Y;

constraint alldifferent([S,E,N,D,M,O,R,Y]);

solve satisfy;

output ["␣(S)␣(E)␣(N)␣(D)␣n",
        "␣␣(M)␣(O)␣(R)␣(E)␣n",
        "␣␣␣(M)␣(O)␣(N)␣(E)␣(Y)␣n"];
```

which, once have been saved on a file, call it send.mzn, can be solved on any system supporting MiniZinc to obtain the following result:

```
Compiling send.mzn
Running send.mzn
  9567
+ 1085
= 10652
-----
Finished in 174msec
```

It is worth noticing that, differently from traditional imperative languages, the ordering of the various parts of the model is not relevant: for example, constraints could be defined before parameters and variables.

### 13.4.2 A MiniZinc COP Model

We now show an example of an optimization model by using a simplified version of the Traveling Salesperson Problem (TSP). In doing so, we will introduce some more features of MiniZinc (these features, of course, can be used also in satisfaction problems).

Concerning data, we can use *arrays*, both of variables and of parameters, in a fashion very much similar to other imperative languages (perhaps with the exception that here array indexes start from 1).

```
int: n;                % number of cities
int: start_city;       % the city where we start the tour
int: end_city;         % the city where we end the tour

array[1..n] of string: city_name; % names corresponding to
                                % integers denoting cities
array[1..n,1..n] of int: distance; % city distances
array[1..n] of var 1..n: solution; % order of cities in solution
```

Lines starting with % in the code above are comments which should help understanding the model. Here we introduce three parameters `n`, `start_city` and `end_city`, two arrays of parameters `city_name` and `distance` and one array of variables `solution`. The problem consists in finding a path for visiting all the `n` given cities once, from `start_city` to `end_city`, while minimizing the total distance travelled, where the distance between two next cities in the path is provided in the matrix `distance`.<sup>14</sup> The (array of) parameters will be instantiated by the data provide as input to the problem, while the array of variables will be instantiated by the solver and will provide the solution in the form of a sequences of cities which will form our travel.

Next we introduce the constraints, which simply state which are the start and end city of the travel in the solution, and that no city can be visited more than once. Notice that this last constraint is enforced by simply using as argument to `all_different` the name of the array whose elements must be different (of course, we must include the appropriate library to be able to use this global constraint).

```
constraint solution[1] = start_city;
constraint solution[n] = end_city;
constraint all_different(solution); % no city visited more than once
```

---

<sup>14</sup> Here we assume that every city is connected to all other cities by a direct path (an arc in the graph). This assumption can be relaxed easily, by introducing a max value which is greater than the sum of all other distances to represent the absence of a direct path between two cities, and then checking accordingly the found solution. Moreover, normally the TSP problem requires that the end city is the same as the start city while here we relax this constraint.

Finally we must define the objective function which need to be minimized and then call the solver:

```
var int: total_distance = sum(i in 2..n)
    (distance[solution[i-1],solution[i]]);
solve minimize total_distance;
```

Here we use the MiniZinc aggregation function `sum` which allows to sum all the elements of an array for a given range of the index. In our case, we are summing the values of the matrix `distance` in position `[solution[i-1],solution[i]]`, for all the values of the index `i` contained in the range `2..n`. This means we are summing the distances from the first city of the tour to the second, from the second to the third and so on until the end city. It is also worth noticing that the indices `solution[i-1]` and `solution[i]` of the distance matrix are variables which will be instantiated by the solver when this has found a solution.

MiniZinc also supports the following aggregation functions for arithmetic arrays; `product`, which multiplies the elements of the array, `min` and `max` which respectively return the least and greatest element in the array. For arrays containing Boolean expressions MiniZinc offers four other aggregation functions: `forall` returns the logical conjunction of all the constraints in the array, thus guaranteeing that all the constraints hold, while `exists` returns the logical disjunction of all the constraints in the array, thus guaranteeing that at least one constraint hold. Moreover we have also `xorall` and `iffall` which ensure that an odd or even number of constraints hold, respectively.

We can now put together all the components of our TSP model, and by adding some cosmetic aspects for the output we obtain:

```
include "all_different.mzn";

int: n;                % number of cities
int: start_city;       % the city where we start the tour
int: end_city;         % the city where we end the tour

array[1..n] of string: city_name;% names corresponding to
                                % integers denoting cities
array[1..n,1..n] of int: distance; % city distances
array[1..n] of var 1..n: solution; % order of cities in solution

constraint solution[1] = start_city;
constraint solution[n] = end_city;
constraint all_different(solution);% no city visited more than once

var int: total_distance = sum(i in 2..n)
    (distance[solution[i-1],solution[i]]);
solve minimize total_distance;

output[show(solution)];
output [city_name[fix(solution[i])] ++ "_->_" | i in 1..n ] ++
    ["\nTotal hours travelled: ", show(total_distance) ];
```

If we save this model in a file `tsp.mzn`, add to it the following data (either in the same `tsp.mzn` file or in a separate `.dzn` file)

```
n=5;
distance=[|0,2,3,6,8|2,0,1,4,6|3,1,0,3,5|6,4,3,0,2|8,6,5,2,0|];
start_city=2;
end_city=1;
city_name=["milano","bologna","firenze","roma","napoli"];
```

and we then we run `tsp.mzn` we obtain the result:

```
Compiling tsp.mzn
Running tsp.mzn
[2, 5, 4, 3, 1]bologna -> napoli -> roma -> firenze -> milano ->
Total hours travelled: 14
-----
=====
Finished in 73msec
```

---

## 13.5 Summary

In this chapter we have introduced the notion of constraint and we have then discussed the main programming paradigms which allow us to use constraints in a native way. In particular we have focused on Constraint logic programs (CLP), a smooth integration of Constraint programming and logic programming which retains the advantages of both the approaches. We have also discussed the use of constraint programming in the context of an imperative language by considering MiniZinc, a free and open-source constraint modeling language which can be used to model both constraint satisfaction and optimization problems in a high-level and solver-independent way.

The use of sophisticated constraint solvers and also optimisers (for example, based on the Simplex algorithm) makes constraint programming competitive with respect to other formalisms in many commercial applications where solutions to combinatorial problems, solutions to optimisation problems and more generally solutions to problems expressible as relations over appropriate domains are required. Examples in this sense include a variety of problems typical of artificial intelligence, from traditional scheduling problems and bioinformatics, to modern decision support systems and fintech applications. The bibliographical notes in the next section provide references to many more examples.

---

## 13.6 Bibliographical Notes

Solving an equation over the integers is a form of constraint solving and is one of the oldest problems in mathematics—already the Babylonians were able to solve several equations with two variables. So it is not surprising that there exists a huge literature on constraint solving and programming. Here we mention only a few publications

which are directly relevant to the content of this chapter and which can also be consulted for finding references to more specialised material.

Constraint satisfaction problems arose in Artificial Intelligence already in the 70s of previous century, in particular for applications to picture processing, see for example the article [5] by Montanari which introduced also the notions of node and arc consistency.

The first true constraint logic programming language was PROLOG II, a language designed by Colmerauer in 1982 [6] which allowed equations and disequations over rational trees. Later on in 1986 Colmerauer extended PROLOG II to PROLOG III [7], a language which provided also constraints on booleans, strings and arithmetics. Jaffar, Lassez and Maher showed in [8] how the usual equation over terms of logic programming could be extended to constraints over generic domains while maintaining the traditional semantics of logic programming, a result later obtained also for other more operational semantics [9].

The paper [10] in 1987 clarified that these and other extensions of logic programming were instances of the constraint logic programming scheme, thus introducing the terminology “Constraint Logic Programming”. Jaffar and his colleagues at Monash university in these years developed also CLP( $\mathcal{R}$ ) [11, 12], a CLP language over real arithmetic, while Dincabas and others at ECRC in the same period developed CHIP [13], an extension of Prolog which allowed constraints over finite domains and integer ranges. The survey [14] provides an excellent overview of theory and applications of CLP, including an historical perspective and a large number of references.

The books [2] by Marriott and Stuckey, [15] by Frühwirth and Abdennadher, and [16] by Apt all provide very good introductions to languages with constraints, in particular [2] focus on CLP.

The documentation available at [3] can be consulted for a more in depth introduction to MiniZinc while the recent book [17] by Wallace introduces the principles of Decision Support Systems and illustrate how to build them by using MiniZinc.

---

## 13.7 Exercises

1. Consider the following program

```
:- use_module(library(clpr)).
p(X,Y):- {X=0, Y=0}.
p(X,Y):- {Y = X*X+Z}, p(X-1, Z).
```

Show a derivation for the goal

```
?- p(2, W).
```

and provide the answer computed by such a derivation.

2. Consider the following program

```
:- use_module(library(clpr)).
```

```
p(X,Y):- {X>=1, Z>=0, Y = X+Z}, p(X-1, Z).
p(X,Y):- {X=0, Y=0}.
```

What is the answer for the following goal?

```
?- p(W, 3).
```

Consider now the program

```
:- use_module(library(clpr)).
p(X,Y):- {X>=1, Y = X+Z}, p(X-1, Z).
p(X,Y):- {X=0, Y=0}.
```

And the same goal as before

```
?- p(W, 3).
```

What happens now in the derivation for this goal?

3. Antonio, Mary, Carlos, and David are eating a cake and must divide the 9 slices between each other. Antonio cooked the cake, so he wants more slices than anyone else. Mary has done her workout in the morning, so she deserves a treat and wants at least 3 slices. Carlos is on a diet, so he will eat less than 3 slices. David wants to feel unique, so he will eat a number of slices that is different from anyone else, and at least 1. They want to save the remaining slices in the fridge, but the fridge is almost full, so only 1 slice can remain.

Write a CLP or a MiniZinc program to compute how they can divide the slices.

4. Morgan is a traveling salesperson that must visit three clients: Luca, Beth, and Mario. The house of Morgan is distant 3 h from each client. The house of Luca is distant 2 h from Beth, and 3 from Mario. The houses of Beth and Mario are only 1 h distant from each other. Beth must be visited before 14, because she is very busy in the afternoon.

Knowing that Morgan won't leave the house before 9 and want to be back at home at most at 18, write a CLP or a MiniZinc program to compute the possible timetables for Morgan, including leaving and returning home.

5. A traveller is planning her next hike. Starting from her current base, she plans to reach the top of a mountain and then descend to a village. There are two possible path to reach the mountain, the path A requires 2 h, the path B requires 3 h. Then it is possible to take the path C to directly reach the village in 4 h. Alternatively, it is possible to visit another mountain, following the path D, in 1 h. From the second mountain, it is possible to reach the village in 3 h following the path E, or 4 h following the path F. The traveller wants to complete the hike in strictly less than 8 h. Write a CLP or a MiniZinc program to compute which choices traveller has and, for each possible path, what is the required time.
6. A scheduling problem consists in a defining an appropriate scheduling for executing a number of tasks, while observing the given durations of each task and the precedence constraints among them (e.g. painting must be performed after the wall has been constructed).



Write a CLP or a MiniZinc program to solve a scheduling problem, where we have the following tasks and durations:

Task	Duration
t1	12
t2	6
t3	11
t4	7
t5	5
t6	14

and we have the following precedence relation

t1 before t4
t2 before t5
t3 before t4
t4 before t5

Hint: represent each task by a different variable which specifies the starting time unit of the task. Precedence constraints among the tasks can be encoded as inequalities among the corresponding task variables.

7. (Zebra puzzle). A small street has five differently colored houses on it. Five persons of different nationalities live in these five houses. Each person has a different profession, each person likes a different drink, and each has a different pet animal. Moreover we know the following:

- The English lives in the red house.
- The Spaniard has a dog.
- The Japanese is a painter.
- The Italian drinks tea.
- The Norwegian lives in the first house on the left.
- The owner of the green house drinks coffee.
- The green house is on the right of the white house.
- The sculptor breeds snails.
- The diplomat lives in the yellow house.
- They drink milk in the middle house.
- The Norwegian lives next door to the blue house.
- The violinist drinks fruit juice.
- The fox is in the house next to the house of the doctor.
- The horse is in the house next to the house of the diplomat.

We want to know who has the zebra and who drinks water.

Write a CLP or a MiniZinc program to solve the Zebra puzzle by using finite domains.

## References

1. E.C. Freuder, A constraints journal. *Constraints* **1**(1), 5 (1996)
2. K. Marriott, P.J. Stuckey, *Programming with Constraints* (MIT Press, 1998)
3. Monash University and Data61 CSIRO. MiniZinc home page (2014). <https://www.minizinc.org/>. Accessed 05-January-2023
4. P.J. Stuckey, K. Marriott, G. Tack, *The MiniZinc Handbook* (2015). [www.minizinc.org/doc-2.6.4/en/index.html](http://www.minizinc.org/doc-2.6.4/en/index.html). Accessed 05-January-2023
5. U. Montanari, Networks of constraints: fundamental properties and applications to picture processing. *Inf. Sci.* **7**, 95–132 (1974)
6. A. Colmerauer, Prolog II reference manual and theoretical model. Technical report (Université Aix-Marseille II, Groupe Intelligence Artificielle, 1982)
7. A. Colmerauer, Note sur Prolog III, in *SPLT'86, Séminaire Programmation en Logique, 21-23 mai 1986, Trégastel, France* (1986), pp. 159–174
8. J. Jaffar, J. Lassez, M.J. Maher, Logic programming language scheme, in *Logic Programming: Functions, Relations, and Equations*, ed. by D. DeGroot, G. Lindstrom (Prentice-Hall, 1986), pp. 441–467
9. M. Gabbrielli, G. Levi, Modeling answer constraints in constraint logic programs, in *Logic Programming, Proceedings of the Eight International Conference, Paris, France, June 24-28, 1991*, ed. by K. Furukawa (MIT Press, 1991), pp. 238–252
10. J. Jaffar, J. Lassez, Constraint logic programming, in *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21–23, 1987* (ACM Press, 1987), pp. 111–119
11. J. Jaffar, S. Michaylov, Methodology and implementation of a CLP system, in *Logic Programming, Proceedings of the Fourth International Conference, Melbourne, Victoria, Australia, May 25–29, 1987 (2 Volumes)*, ed. by J.-L. Lassez (MIT Press, 1987), pp. 196–218
12. J. Jaffar, S. Michaylov, P.J. Stuckey, R.H.C. Yap, The CLP(R) language and system. *ACM Trans. Program. Lang. Syst.* **14**(3), 339–395 (1992)
13. A. Aggoun, M. Dincbas, A. Herold, H. Simonis, P. Van Hentenryck, The CHIP system. Technical Report TR-LP-24 (European Computer Industry Research Centre (ECRC), Munich, Germany, 1987)
14. J. Jaffar, M.J. Maher, Constraint logic programming: a survey. *J. Log. Program.* **19**(20), 503–581 (1994)
15. T.W. Frühwirth, S. Abdennadher, *Essentials of Constraint Programming, Cognitive Technologies* (Springer, 2003)
16. K.R. Apt, *Principles of Constraint Programming* (Cambridge University Press, 2003)
17. M. Wallace, *Building Decision Support Systems - Using MiniZinc* (Springer, 2020)

---

# Index

## Symbols

$\beta$ -rule, 340  
 $\lambda$ -calculus, 359  
 $\pi$ -calculus, 542

## A

A-list, 110  
abstract machine, 1  
abstraction  
  on control, 163  
  on data, 199, 268  
Ackermann  
  function, 145  
ACP, 535  
activation record, 91  
  dynamic chain pointer, 102  
  for in-line blocks, 92  
  for function, 94  
  for procedure, 94  
  pointer, 96  
  static chain pointer, 94  
Ada, 534  
Aiken, H., 520  
ALGOL, 214, 524  
  scope, 74  
alias, 67  
aliasing, 64, 67, 130  
analysis  
  lexical, 39  
  semantic, 41  
  syntactic, 40

APL, 121  
applet, 537  
application, 337, 340  
arithmetic  
  pointer, 224  
array, 216  
  address calculation, 219  
  allocation, 218, 219  
    dope vector, 220, 221  
  bounds checking, 218  
  in C, 227  
  Java, 221  
  multidimensional, 217  
  shape, 219  
  slice, 217  
  stride, 219  
ASCC/MARK I, 520  
assembly language, 4, 521  
Assignment, 129, *see also* Command  
  augmented, 132  
  chained, 131  
  compound, 132  
  multiple, 132  
association  
  creation, 71  
  deactivation, 71  
  destruction, 71  
  environment, 65  
  lifetime, 72  
  reactivation, 72

**B**

Böhm, C., 136  
 Babbage, C., 533  
 backtracking, 393  
 Backus, J., 524  
 barrier, 444  
 base class  
   fragile, 308  
 Bernstein, A., 527  
 Beta rule, 340  
 binding  
   deep, 178  
   environment, 65  
   shallow, 178  
 blackboard, 439  
 block, 66, 136  
   activation record, 92  
   and procedure, 68  
   in-line, 68  
   protected, 188  
 BNF, 525  
 Boolean, 206  
 Brinch Hansen, P., 449, 532  
 busy waiting, 442  
 Byron Lovelace, A., 533  
 bytecode, 19  
   Java, 537

**C**

C, 528  
   array, 216  
   augmented assignment, 132  
   call by reference, 169  
   contextual constraints, 37  
   enumeration, 209  
   environment, 182  
   function declaration, 79  
   parameter passing, 168  
   union, 213  
 C++, 534  
   call by reference, 243  
   class, 288  
   dynamic method dispatch, 303  
   inheritance, 295  
   multiple inheritance, 297  
   subtype, 290  
   template, 241  
   virtual member function, 292  
 C#, 542  
 call  
   of procedure, 96

Case, *see* Command  
 cast, 236, 237  
   non-converting, 237  
 CCS, 470  
 Central Reference Table, 109  
 channel, 452  
 character, 207  
 CHIP, 536  
 Chomsky, N., 27  
 Church's Thesis, 60  
 Church, A., 60  
 class, 286  
   abstract, 292  
   derived, 290  
   fragile base, 308  
   implementation, 305  
 classless, 288  
 clause, 376, 377  
   as procedure, 387  
   body, 376  
   definite, 376  
   head, 376  
   in Clp, 415  
   unit, 376, 415  
 closure, 175, 179, 181, 183, 186  
 CLP, *see* Constraint logic programs  
 CLP(R), 536  
 COBOL, 526  
 Coercion, 237  
 Colmerauer, A., 531, 536  
 command, 127  
   ;, 135  
   case, 138  
   do, 142  
   for, 143  
   goto, 136  
   if, 137  
   repeat, 142  
   while, 142  
   assignment, 129  
   assignment in C, 131  
   block, 136  
   composite, 136  
   conditional, 137  
     syntactic ambiguity, 138  
   guarded, 460  
   iteration  
     bounded, 142  
     unbounded, 142  
   iterative, 141  
   sequence, 134

- structured, 149
  - communication
    - asynchronous, 440
    - in Java threads, 466
    - mechanisms of, 438
    - message exchange, 439
    - shared memory, 439
    - synchronous, 441
  - compatibility of types, 234
    - structural, 235
  - compile time, 65
  - compiler, 1, 39
  - composition
    - parallel, 462
  - computability, 57
  - computation, 43
  - computed answer, 391–393
  - communication
    - shared memory, 439
    - synchronous, 456
  - concurrency, 433, 436
    - logical, 437
  - constraint, 410, 536
    - and generate, 418
    - definition, 414
    - optimization problem, 413
    - propagation, 411
    - satisfaction problem, 412
  - Constraint logic programming, *see* Constraint logic programs
  - Constraint logic programs, 413, 536
    - computation, 417
    - semantics, 416
    - solve rule, 416
    - syntax, 415
    - transition relation, 416
    - transition system, 416
    - unfold rule, 416
  - constructor, 211, 229, 293
    - chaining, 294
    - type, 230
  - continuation, 157
  - contravariant, 327
  - COP, *see* Constraint optimization problem
  - copy rule, 173, 339
  - covariant, 326
  - CRT, *see* Central Reference Table
  - crypto-arithmetic puzzle, 418
  - CSP, *see* Constraint satisfaction problem
  - CSP language, 451, 470
    - naming, 451
    - Synchronous communication, 456
    - Curry, H., 351
- D**
- Dahl, O.J., 526
  - dangling pointer, 248
  - Datalog, 402
  - deadlock, 447
  - deallocation, 224, 247
  - declaration, 66
    - scope, 80
  - deep binding, 178
  - delegation, 288
  - denotable, 204
  - dereference, 223, 248
    - l-value, 222
  - derivation, 30
  - Dijkstra, E.W., 159, 445, 470, 527, 532
  - dispatch, 300
    - multiple, 304
  - Display, 107
  - dope vector, 220
  - dotted pair, 119
  - downcast, 307
  - duck typing, 235
  - dynamic
    - chain, 93
    - selection, 285
- E**
- Eckert, J.P., 520
  - EDSAC, 520
  - EDVAC, 520
  - Eich, B., 540
  - emulation, 9, 10
  - engine
    - analytic, 533
    - difference, 533
  - ENIAC, 520
  - enumeration, 210
  - environment, 64
    - and block, 66
    - global, 70
    - local, 69
    - non-local, 70
    - referencing, 66
  - Epilogue, 96
  - equality
    - of terms, 381
  - equation
    - between terms, 377

- equivalence
  - by name, 232
  - of types, 203, 231
  - structural, 232
- Erlang, 336, 350, 356
- evaluation
  - applicative-order, 343
  - by name, 344
  - by need, 345
  - by value, 343
  - eager, 343
  - innermost, 343
  - lazy, 345
  - leftmost, 343
  - normal order, 344
  - outermost, 344
  - strategy, 342
- exception, 187
  - implementation, 192
  - resumption after, 189
  - rethrowing, 192
- expressible, 204
- expression, 117
  - associativity, 120
  - evaluation, 120, 123
    - APL, 121
    - eager, 125
    - lazy, 126
    - short circuit, 126
    - tree, 123
  - infix, 124
  - lambda, 185
  - notation
    - Cambridge Polish, 120
    - infix, 118
    - Polish, 119
    - postfix, 120
    - prefix, 119
  - precedence, 120
  - prefix, 122
  - representation as tree, 120
  - semantics, 47
  - syntax, 118
- expressiveness
  - of bounded iteration, 145
- F**
- F#, 542
- fairness, 444
- Fibonacci
  - function, 151
  - sequence, 88
- Fibonacci, L., 88
- field
  - of record, 211
  - shadowing, 291
- filter, 349
- firmware, 10
- First-order Logic, 373
- fixed point
  - operator, 363, 364
  - representation, 208
- float, 207
- floating point, 207
- fold, 349
- For, *see* Command
- FORTRAN, 522, 524
- fragile
  - base class, 308
- fragmentation, 100
  - external, 100
  - internal, 100
- Free list, 100
- funarg problem, 195
- function, 164
  - activation record, 94
  - as parameter, 177
  - as result, 182
  - call, 96
  - computable, 58
  - covariant, 327
  - higher-order, 176
  - mutually recursive, 79
  - partial, 12
  - virtual in C++, 292
- Futumara, projection, 24
- G**
- Gödel, K., 58
- garbage collection, 251
  - copying, 257
  - mark and compact, 256
  - mark and sweep, 254
  - pointer reversal, 255
  - reference counting, 252
  - stop and copy, 257
- Gauss-Jordan, 411
- generate and test, 418
- generator, 185
- generic, 318, 320
  - implementation, 325
- GHC, 535

Go, 544  
goal, 376, 415  
    evaluation, 389  
Gosling, J., 536  
Goto, *see* Command  
grammar  
    ambiguous, 35  
    context-free, 28  
    contextual, 37  
    regular, 40  
Griesemer, R., 544  
guard, 460  
guarded command, 460

**H**  
Halting Problem, 53  
handler, 188  
Hanoi  
    towers of, 396  
hardware  
    implementation, 8  
Haskell, 204, 336, 540  
heap, 97, 98  
    block  
        fixed length, 98  
        variable length, 99  
    buddy system, 102  
    compaction, 101  
    Fibonacci, 102  
    fragmentation, 100  
        external, 100  
        internal, 100  
    management, 98  
    multiple free list, 101  
    single free list, 100  
Herbrand Universe, 381  
Herbrand, J., 369  
hiding  
    of information, 267, 270  
hierarchy of abstract machines, 20  
higher order, 176, 366  
Hoare, C.A.R., 449, 470, 525, 532  
Hoare, G., 544  
Hopper, G., 526  
HTML, 536

**I**

Ichbiah, J., 534  
If, *see* Command  
implementation, 8, 26, 267  
    compiled, 12

    of Pascal, 19  
    purely compiled, 14  
    purely interpreted, 12, 13  
    real case, 16  
index  
    type of, 216  
Inductive definitions, 151  
inference  
    type, 203  
information hiding, 267, 270  
inheritance, 295, 297  
    implementation, 295  
    interface, 295  
    multiple, 312  
    with replication, 315  
    with sharing, 316  
integer, 207  
interface, 269  
interleaving, 462  
interpreter, 1, 2, 7, 13  
interrupt, 434  
interval, 205, 210  
iterable, 147  
iteration  
    bounded, 142  
    unbounded, 142

**J**

Jacopini, G., 136  
Jaffar, J., 536  
Java, 201  
    Runnable, 464  
    Socket, 468  
    notifyAll, 468  
    notify, 468  
    run, 463  
    start, 463  
    synchronized, 466  
    wait, 468  
    array, 325  
    augmented assignment, 132  
    bytecode, 19, 537  
    contextual constraints, 37  
    evaluation order, 125, 130  
    exception, 187  
    for-each, 147  
    generic, 320  
    inheritance, 296  
    interface, 292  
    JVM, 19  
    parameter passing, 171

- reference model for variables, 129
- RMI, 469
- subtype, 289
- thread, 463
- Virtual Machine, 19, 309, 310, 537
- wildcard, 324, 326

JavaScript, 288, 538

Jensen

- device, 177

Jolie, 486

JVM, 19, 537

## K

Kay, A., 527, 530

Kleene, S., 58

Kowalski, R., 531

## L

l-value, 130, 211

lambda, 185

lambda calculus, 359

Landin, P., 159, 356

Language

- assembly, 4, 521

- declarative, 134

- domain-specific, 543

- formal, 28

- functional, 347

- high-level, 4

- higher order, 176

- imperative, 134

- logic, 375

- low-level, 4

- scripting, 538

Lerdorf, R., 539

lifetime, 72, 166, 221

Linda, 439, 535

Liskov substitution principle, 236

Liskov, B., 236

LISP, 109, 119, 525

- garbage collection, 251

- side effects, 353

list

- Prolog, 370

- Python, 226

loan

- CLP example, 420

lock, 442

locks and keys, 250

logic program, 375

- computational model, 386

- declarative interpretation, 387

- failure, 390

- procedural interpretation, 387

- success, 390

Lukasiewicz, L., 119

## M

machine

- abstract, 1, 20

- hardware, 3

- implementation, 9, 17

- intermediate, 16

- Java, 19

- Pascal, 19

- interpreter, 1, 2

- memory, 5, 7

- SECD, 356

- von Neumann, 520

map, 349

MARK I, 520

Matsumoto, Y., 539

Mauchly, J., 520

McCarthy, J., 525

memory, 87

- heap, 97

- semantic domain, 133

- stack, 90

- static, 89

Menabrea, L., 533

Messaging Patterns, 487

- Enterprise Service Bus, 489

- request-response, 488

method

- abstract, 292

- binary, 329

- dynamic lookup, 300

- overriding, 291

- redefinition, 290

- resolution order, 299, 300

- static, 288

Milner, R., 470, 532, 541, 542

MiniZinc, 422

- constraints, 423

- COP example, 425

- CSP example, 422

- parameters, 423

- variables, 423

mix-in, 296

ML, 336, 337, 351–353, 530

- recursive type, 228

- reference cell, 353



Modula, 532  
module, 274, 458  
monitor, 448, 449  
    signal, 450  
    wait, 450  
    condition variable, 449, 450  
monomorphic, 238  
MRO, 299, 300  
multimethod, 304  
mutual exclusion, 441  
    lock, 442

## N

name, 63  
    clash, 297  
    conflict, 297  
    mangling, 289  
naming mechanisms, 451  
Naur, P., 524  
negation, 400  
    as failure, 401  
non-determinism, 392  
Nygaard, K., 526

## O

object, 284  
    allocation, 288  
    creation, 72  
    deallocation, 288  
    denotable, 63, 65  
        access, 72  
        creation, 72  
        destruction, 72  
        lifetime, 72  
        modification, 72  
    implementation, 306  
OCaml, 541  
Occam, 452, 532, 535  
Odersky, M., 543  
operator  
    associativity, 121  
    precedence, 118  
option type, 215  
order  
    column-major, 218  
    row-major, 218  
overloading, 239  
overriding, 291  
    contravariant, 327

## P

P-code, 529  
package, 274  
paradigm  
    concurrent, 433  
    constraint, 409  
    functional, 335  
    logic, 369  
    object-oriented, 279  
    service-oriented, 473  
parallel composition, 462  
parallelism  
    maximal, 462  
parameter, 165, 167  
    actual, 165  
    call by assignment, 171  
    call by constant, 170  
    call by name, 173  
    call by need, 174  
    call by object reference, 171  
    call by reference, 168  
        in C, 169  
    call by result, 171  
    call by value, 168  
    call by value-result, 172  
    formal, 165  
    passing, 167  
PARLOG, 535  
parser, 40  
Pascal, 529  
    contextual constraints, 37  
    equivalence by name, 232  
    variant record, 213  
pattern matching, 348  
Peano, G., 151  
PHP, 538  
pi-calculus, 542  
Pike, R., 544  
pointer, 222  
    arithmetic, 224  
    dangling, 248  
    dereference, 248  
    dynamic chain, 93, 102  
    reversal, 255  
    static chain, 94  
polymorphism, 318  
    explicit, 241  
    implicit, 241  
    inclusion, 239  
    parametric, 239  
    subtype, 239

- universal, 239
- universal parametric, 240
- universal subtype, 242
- predicate, 371, 374
  - calculus, 373
- procedure call
  - calling sequence, 96, 97
  - epilogue, 96
- process, 433
  - heavyweight, 433
  - lightweight, 433
- Production, 29
- program transformation, 22
- Programming
  - concurrent, 433
  - constraint, 409
  - distributed, 437
  - functional, 347
  - logic, 369
    - constraint, 413
    - declarative interpretation, 387
    - procedural interpretation, 387
  - multithreaded, 437
  - sequential, 433
  - service-oriented, 473
- Prolog, 397
  - arithmetic, 398
  - backtracking, 393
  - clause selection, 391
  - computational model, 386
  - control, 392
  - cut, 399
  - disjunction, 400
  - failure, 394
  - history, 531
  - negation, 400
  - non-determinism, 393
  - occurs check, 384
  - selection rule, 389
- Prolog II, 536
- Prolog III, 536
- Prologue, 96
- prototype-based, 288
- Python, 538
  - int type, 207
  - assignment
    - augmented, 132
    - chained, 131
    - expression, 131
    - multiple, 132
  - Boolean operator evaluation, 127

- bytecode, 19
- class constructor, 294
- class hierarchy representation, 305
- contextual constraints, 37
- duck typing, 235
- dynamic access attributes, 285
- EAFP, 191
- evaluation order, 125, 130
- exception, 187
- for, 148
- garbage collection, 254
- generator, 185
- indentation, 39, 68, 136
- inheritance, 297
- lambda expression, 186
- list, 226
- MRO, 300
- multiple inheritance, 297
- name
  - definition, 79
  - mangling, 289
- parameter passing, 171
- private attribute, 289
- reference model for variables, 129
- scope, 74
- suite, 68
- tuple, 226
- visibility, 69
- visibility rule, 193

## Q

- query, 376, 415

## R

- r-value, 130
- Rails, 539
- range, 210
- real, 207
- receive, 439
- record, 211
  - variant, 213
- recursion, 88, 89, 151
  - mutual, 79
  - tail, 88, 153
- redex, 340
- reduce, 349
- reduction, 338
- reference counter, 252
- refutation
  - SLD, 391
- rendez-vous, 457

- representation independence, 273
- resolution, 369, 376
  - SLD, 376, 391
- rewriting, 336, 338
- Ritchie, D., 528
- Robinson, A., 369, 531
- root set, 254
- Roussel, A, 531
- RPC, 457
- Ruby, 538
  - array, 226
  - on Rails, 539
- rule
  - beta, 340
  - copy, 173, 339
  - visibility, 63, 193
- runtime, 65
- Rust, 544
- S**
- S-expressions, 119
- Scala, 173, 350, 543
  - lambda expression, 186
  - reference model for variables, 129
  - structural compatibility, 235
- Scheme, 204, 248, 540
- scope, 73
  - A-list, 110
  - CRT, 109
  - declaration, 74
  - display, 107
  - dynamic, 76
  - dynamic chain, 93
  - implementation, 102
  - Java, 74
  - problems, 78
  - rules, 73
  - static, 74
  - Static Chain, 102
  - static chain, 94
- SECD, 356
- selection rule, 389
- Self, 288
- semantics, 44, 46
  - denotational, 44
  - operational, 44
  - static, 38
- semaphore, 445
  - P, 446
  - V, 446
- send, 439
- sequence, 226
- serialisation, 481
- service, 476
  - choreography, 479
  - orchestration, 479
- Service-Oriented Architectures, 510
- Shadowing, 291
- shallow binding, 178
- side effect, 125
- Simula, 526
  - class, 286
- slice, 217
- Smalltalk, 529
  - class hierarchy representation, 305
- SNOBOL, 124
- solution
  - of constraints, 413
- spaghetti code, 149
- specification
  - of ADT, 272
- speed-up, 436
- SR, 535
- stack, 91
- standard model, 56
- static
  - in C, 166
- Static Chain, *see* Scope
- strategy
  - evaluation, 342
- stream, 352
- strongly typed, 203
- Stroustrup, B., 534
- stub, 458
- subclass, 290
- substitution, 378
  - application, 379
  - composition, 379
  - ground, 379
  - most general, 381
  - renaming, 380
  - restriction, 380
- subtype, 236, 289
  - behavioural, 236
- Swift, 545
- synchronisation
  - busy waiting, 442
  - condition, 441
  - mechanisms, 440
  - mutual exclusion, 441, 442
  - scheduler-based, 445
- Synchronising Resources, 535

syntactic sugar, 119, 159  
syntax, 25, 27  
    abstract, 40  
system stack, 91

## T

tail recursion, 88, 153  
Template, 243  
term, 374  
    ground, 375  
theorem  
    of Böhm and Jacopini, 136  
Thompson, K., 528, 544  
thread, 433  
thunk, 181, 186  
tombstone, 248  
towers of Hanoi, 396  
trait, 543  
transition, 44, 46  
transition system  
    for constraint logic programs, 416  
Traveling Salesperson Problem, 425  
tree, 31  
    derivation, 31  
    syntax, 123  
tuple space, 439  
Turing  
    completeness, 58  
    equivalent, 58  
    machine, 17, 58  
Turing, A.M., 57  
type, 199  
    base of interval, 210  
    checker, 201, 233  
    checking, 204, 245  
    compatibility, 234  
    composite, 211  
    conversion, 234  
    discrete, 211  
    equivalence, 231  
    explicit conversion, 238  
    index, 216  
    inference, 245  
    monomorphic, 238  
    of functions, 230  
    option, 215  
    recursive, 228  
    recursive in ML, 230  
    safety, 199, 203, 205, 218, 226  
    scalar, 205  
    simple, 205

    system, 199  
    void, 208  
typing  
    duck, 235  
    dynamic, 204  
    static, 204

## U

undecidability, 55  
    of halting problem, 55  
    of static type checking, 205  
unification, 377  
    algorithm, 383  
    Martelli and Montanari algorithm, 383  
theory, 377  
unifier  
    most general, 381  
union, 213  
    tagged, 215  
unit, 209

## V

value  
    denotable, 133, 134, 204  
    expressible, 133, 134, 204  
    functional, 342  
    storable, 133, 134, 204  
van Rossum, G., 539  
variable, 128, 335, 377  
    bound, 165  
    capture, 174  
    class, 288  
    external, 182  
    global  
        static allocation, 89  
    in activation record, 91  
    in functional languages, 335  
    in imperative languages, 128  
    in logic programming, 377  
    instance, 286  
    logic, 377  
    modifiable, 128  
    reference model, 129  
    shadowing, 291  
    static, 166  
variance, 380  
vector  
    dope, 220  
visibility, 69

rule, [63](#), [193](#)  
void, [209](#)  
von Neumann, J., [336](#), [520](#)  
vtable, [307](#)

**W**  
Warren Abstract Machine, [404](#)  
Wilkes, M., [520](#)  
Wirth, N., [525](#), [529](#), [532](#)