PROLOG FOR PROGRAMMERS

Feliks Kluźniak

Stanisław Szpakowicz

PROLOG FOR PROGRAMMERS

Feliks Kluźniak

Stanisław Szpakowicz

Institute of Informatics
Warsaw University
Warsaw, Poland

With a contribution by Janusz S. Bień

1985

ACADEMIC PRESS, INC.
Harcourt Brace Jovanovich, Publishers
London Orlando San Diego
New York Austin Montreal Sydney
Tokyo Toronto

ACADEMIC PRESS INC. (LONDON) LTD.
24–28 Oval Road
LONDON NW1 7DX

United States Edition published by
ACADEMIC PRESS, INC.
Orlando, Florida 32887

British Library Cataloguing in Publication Data

Kluźniak, Feliks

Prolog for programmers.
1. Programming languages (Electronic computers)
2. Electronic digital computers--Programming
I. Title II. Szpakowicz, Stanisław
000.64'24 QA76.7

Library of Congress Cataloging in Publication Data

Kluźniak, Feliks.
Prolog for programmers.

Includes index.
1. Prolog (Computer program language) I. Szpakowicz,
Stanisław. II. Bień, Janusz St. III. Title.
QA76.73.P76K58 1985 001.64'24 84-14520
ISBN 0-12-416520-6 (alk. paper)

PRINTED IN THE UNITED STATES OF AMERICA

86 87 88 89 9 8 7 6 5 4 3 2

PREFACE

Prolog is a non-conventional programming language for a wide spectrum of applications, including language processing, data base modelling and implementation, symbolic computing, expert systems, computer-aided design, simulation, software prototyping and planning. A version of Prolog has been chosen as a systems programming language for so-called fifth-generation computers; experiments with systems programming and concurrent programming are in progress.

Prolog, devised by Alain Colmerauer, is a logic programming language. Logic programming is a new discipline which lends a unifying view to many domains of computer science. Prolog can be classified as a descriptive programming language, as opposed to prescriptive (or imperative) languages such as Pascal, C and Ada. In principle, the programmer is only supposed to specify *what* is to be done by his or her program, without bothering with *how* this should be achieved. Robert A. Kowalski has coined the "equation"

Algorithm = Logic + Control,

which emphasizes the distinction between the *what* (logic) and the *how* (control). The programmer need not always specify the control component. In practice however, Prolog can be treated as a procedural language.

Prolog is not standardized, and it comes in many different flavours. The most widespread dialect of Prolog is Prolog-10, originally implemented by David H. D. Warren for DEC-10 computers. We describe a variant of this dialect, based on an interpreter written in Pascal especially for this book.

The main part of the book is Chapters 1–5. Chapters 1 and 3 are an introduction to Prolog, intended for those who use prescriptive languages in their everyday practice. Both intuitions and the presentation are "practically" biased, but we assume the reader has a certain amount of programming experience and sophistication. Chapter 2 explains Prolog in terms of logic. It requires no deep knowledge of mathematics and is intended as a counterpoint to Chapter 1, but can be skipped on a first reading. Chapter 4 contains some useful programming techniques and hints. Chapter 5 is a reference manual for the version of Prolog described in this book. In addition, Chapter 8 is a discussion of two rather illuminating applications.

For those who wish to gain more insight into the language and its inner workings, Chapter 6 introduces basic principles of Prolog implementations. An implementation of the dialect described in this book is presented in Chapter 7 and in the appendices (which contain complete listings). We used this implementation to test our examples, including the case studies of Chapter 8.

Chapter 9, written by Janusz S. Bień (who also did most of the bibliography), briefly outlines the most characteristic features of several other Prolog dialects.

The material in this book, supplemented by some additional reading and a programming assignment, can be used for a two-semester course at the level of third-year computer science majors. Re-implementation of or extensions to the interpreter of Chapter 7 might make interesting assignments for a translator-writing course.

While working on this book, we used the computing facilities of the Institute of Informatics, Warsaw University. We would like to thank Pawel Gburzynski and Krzysztof Kimbler, who helped us switch almost painlessly to a different machine when the one we originally used broke down for a protracted period of time. We thank David H. D. Warren for permitting us to include the listings of WARPLAN. We are also grateful to all those who have provided us with logic programming literature for the past 10 years.

CONTENTS

1 AN INTRODUCTION TO PROLOG

Prolog is an unconventional language. In particular, its data structures are quite different from those found in other programming languages. As it is difficult to talk about a computation without understanding the sort of data that can be processed, we shall discuss data structures at some length before coming to the question of how to do anything with them. Have patience.

1.1. DATA STRUCTURES

1.1.1. Constants

Constants are the primitive building blocks of data structures. Constants have no structure, so they are often called "atoms." They represent only themselves—they can be thought of as identical with their names.

In Basic or Fortran, 1951 is a constant. The integer variable **J** is not, because it represents both a memory cell and—in certain contexts—a value. The value is something quite different from the variable itself.

One is accustomed to treating 1951 as a number greater than 1948, but this is because in programming languages constants usually belong to certain types. The usual properties of integer constants (their ordering, ability to be used in arithmetic operations, etc.) are taken for granted by virtue of their belonging to the type **integer**, just as in Pascal blue is a successor of red when one writes

```
colour = ( red, blue, green )
```

Such type definitions impose a certain structure on the otherwise undifferentiated universe of individual symbolic constants, each of which has only one attribute: its name.

The interpretation of a constant rests solely with the programmer. 1951 can be the price of a computer, the weight of a truck, the time of day or a year of birth. One can always multiply it by 4, but this seldom makes sense when it represents a car's registration number. The constant blue is less burdened with inadequate interpretations, but one might wish not to have the colours ordered. Constants are the primitives, and collecting them into types should only be done when necessary.

In Prolog, as in other symbolic languages (such as Lisp) there is no need to declare constants or group them into types. One can use them freely, simply by writing down their names.

A legal constant name is one of the following:

—A sequence of digits, possibly prefixed by a minus sign; by convention, such constants are called **integers** (e.g. 0, −7, 1951);
—An **identifier**, which may contain letters, digits and underscores but must begin with a lower case letter (e.g. q, aName, number_9);
—A symbol which is a nonempty sequence of any of the following characters:

+ − * / < = > . : ? $ & @ # \ ¬

—Any one of the characters

, or ; or !

—The symbol [] (pronounced "nil");
—A **quoted name**, written according to the Pascal convention for strings: an arbitrary sequence of characters enclosed in apostrophes, an apostrophe being represented by two consecutive apostrophes (e.g. 'Can''t do this.' consists of 14 characters).

All of these constants are purely symbolic and have no inherent interpretation. However, some primitive operations in Prolog do treat them in a special way:

—Arithmetic operations interpret integers as representations of integer values (they can also create new integers);
—Comparison operations interpret integers as integer values, and all other constants as representations of the sequences of characters forming their names (these are lexicographically ordered by the underlying collating sequence);
—Input/output operations interpret all symbols as sequences of characters forming their names.

Each occurrence of a constant's description (name) is treated as referring to *the same* constant, but of course we are free to interpret each separately.

1.1.2. Compound Objects

An important aspect of the expressive power of a programming language is its ability to *directly* describe various data structures. Of the popular and widely used languages, Pascal is the most powerful in this respect, but it has several shortcomings. (This is *not* a criticism of Pascal: our point of view does not take into account important design objectives such as a safe type mechanism.)

Firstly, type definitions in Pascal are overspecified. It is impossible to program general algorithms which process stacks or trees regardless of the type of their elements. (Records with variants are only a rough approximation to generic data types found in some more recent programming languages.)

Secondly, those data structures which change their form dynamically can only be built with pointers. One must therefore deal with the structures at a very low level: the level of representation rather than the conceptual level at which many other things are done in Pascal. Programs using pointers are error-prone and hard to understand, because operations on such data structures are *encoded* rather than directly *expressed*.

The third shortcoming has a similar effect. Ironically, Pascal types are also *under*specified, in that there is no way to directly express certain quite natural constraints on the arguments of operations. One cannot say that the function POP can only be applied to a non-empty stack; one can only write a piece of code (hopefully correct) which checks the argument.

It is interesting that these shortcomings are not shared by Prolog data types (or rather by their counterparts, since "type" is not really a Prolog concept). And yet Prolog data structures are very simple. Let us look at the details.

Functors

To describe a compound object, it is not enough to list its components. The ordered pair (19, 24) can be an object of the type

```
rectangle = record
                height, width : integer
            end
```

as well as an object of the type

```
timeofday = record
                hour, minute : integer
            end
```

The complete description of a compound object must include a definition of its structure. Structure is defined principally by describing the way in which the object and its components are interrelated. Describing these interrelationships often consists in simply giving a **type name** to an aggregate of components (as in the example above). It is the programmer's responsibility to interpret this name in terms of real-world relations between entities being modelled by the program.

In conventional programming languages, the structure of a compound object is usually described in a declaration associating the object with a type definition. The type definition lists the name of the object–component relationship (type name) and, possibly, additional information about the structure (types) of the components. The object is described by its name (or the name of a pointer). The name's definition is textually remote from its occurrences.

A different approach is taken in Prolog. Here, the type name is an integral part of all the occurrences of the object's description. The notation is very simple: a description of a compound object is the type name followed by a parenthesized sequence of descriptions of its components, separated by commas. We write either

```
rectangle( 19, 24 )
```

or

```
timeofday( 19, 24 ).
```

The notation is similar to that used for writing functions in mathematics. Terminology reflects this similarity. The type's name is called a **functor**, and the components are called **arguments**. There is more to it than superficial similarity of two simple syntactic conventions. One can certainly regard a type such as *rectangle* as a function mapping components into compound objects. From this point of view it is not surprising that we can have functors with no arguments: these are simply constants. Sometimes it is also useful to have one-argument functors. For example, the integer 2 can be represented by the object successor(successor(zero)) (the fact that $2 > 1 > 0$ is evident from its structure).

From the discussion above, it should be obvious that the important attributes of a functor are both its name and its **arity** (i.e. the number of arguments it takes). In Prolog, we can use both

```
timeofday( 17, 13 )
```

and

```
    timeofday( 1713 )
```

in the same program. Even if the intended interpretation is the same, these are two different objects: one has two components, and the other has one. There are also two different functors, both named timeofday. Whenever we speak of a functor in a context which gives no indication about its arity, the arity must be given explicitly. The conventional notation is to write it after a slash: timeofday/2 or timeofday/1.

The lexical rules for forming functor names are the same for all arities, but integers can only be constants. Thus

```
    123( a, b )
```

is incorrect, but

```
    '123'( a, b )
```

is perfectly all right. Also, [] is only a constant.

Object Descriptions

Descriptions of constants and compound objects are referred to as **terms**. Usually, the objects themselves are also called terms: this causes no confusion in practice, but in this chapter we shall try to distinguish between the two meanings.

The arguments of a term are arbitrary terms. For example, one can write a term describing a "record":

```
    customer( name( john, smith ),
              address( street( north_ave ), number ( 173 ) ) ).
```

Of the various functors in this example, the outermost, customer/2, can be said to define the general structure of the term. It is called the **main functor**, or **principal functor**. Similarly, name/2 is the main functor of the first argument.

Here is another example of a common data structure. A list can be defined as either the empty list, or a list constructed of any object (a head) and a list (a tail). A list of the first three letters in the alphabet could then be described by the term

```
    cons( a, cons( b, cons( c, emptylist ) ) ).
```

The following term would be a description of the two-element list constructed of the above list and a list containing the integer zero:

```
    cons( cons( a,cons( b,cons( c,emptylist ) ) ),cons( 0,emptylist ) )
```

Even this small example demonstrates that nested parentheses can be difficult to read. Prolog therefore provides syntactic sugar to hide this **standard** or **canonic form** of terms. Instead of writing

successor(successor(zero))

one can choose to use *successor* as a **prefix functor** and write

successor successor zero.

Alternatively, *successor* can be made a **postfix functor**:

zero successor successor.

Functors with two arguments can be declared as **infix functors**, e.g.

&(a, b)

can be written as

a & b.

The term

a & b & c

would be ambiguous, so an infix functor is either **left-associative** or **right-associative** (or non-associative, in which case the term is incorrect). If & is right associative, the term's standard form is

&(a, &(b, c)) ;

if & is left-associative, then the term stands for

&(&(a, b), c).

We can use parentheses to stress or override associativity. If & is right-associative, then

a & b & c

is equivalent to

a & (b & c)

but not to

(a & b) & c,

which stands for

&(&(a, b), c)

regardless of associativity.

To make parentheses even less frequent, prefix, postfix and infix functors are given **priorities**. Functors with lower priority take precedence over those with a higher priority (a Prolog-10 convention, different from that used in other programming languages and mathematics). For example, if the priority of * is lower than that of +, then

```
3 * 4 + 5      and      5 + 3 * 4
```

denote

```
+( *( 3, 4 ), 5 )      and      +( 5, *( 3, 4 ) )
```

We can use parentheses to stress or override priorities, by writing

```
( 3 * 4 ) + 5      or      3 * ( 4 + 5 ).
```

Prefix, postfix and infix functors are usually referred to by the generic name **operators**. Remember that these are not operators in any conventional sense: they are only a syntactic convenience.

Operator names may not be quoted. If an operator is to be written in standard form or with a different number of arguments, it must be quoted. If + is an infix functor,

```
a + b,      '+'( a, b )      and      '+'( a, b, c )
```

are correct terms, but

```
+      and      +( a, b )
```

are not.

It is also possible to declare **mixed operators**, i.e. functors such as the minus sign, which is both prefix and infix in ordinary arithmetic. Details about declaring prefix, postfix and infix functors can be found in Sections 5.1 and 5.7.3.

For the time being, we shall only use infix functors to write terms representing lists. However, instead of

```
a cons b cons c cons emptylist
```

we shall use a more concise notation, modelled after Lisp. The empty list will be denoted by the constant [] (pronounced "nil"), and the constructing functor – by the right-associative infix functor ./2. Our two lists are then written as

```
a.b.c.[]
```

and

```
( a.b.c.[] ).0.[]
```

The convention is arbitrary, in that any constant and two-argument functor would do in place of [ ] and the dot. It is more convenient than others, because these are the symbols expected by several built-in procedures.

(You can write such terms after feeding Prolog with

```
:- op( 800, xfy, '.' ).
```

However, a minor technical difficulty makes it impossible to use the period as a functor when it is immediately followed by a white space character, such as blank, tab or new line. This is a nuisance, and Prolog provides special syntactic sugar for **lists**: it is somewhat confusing, so we will put it off until Chapter 4.)

STRINGS

Characters are constants whose names consist of single characters. One can use quoted names for characters which are not correct identifiers (e.g. ' ', '(', '3'; and 'x' is equivalent to x).

Strings are lists of characters. One can also write them in double quotes. For example

```
"string"     and     " '''"
```

stand for

```
s.t.r.i.n.g.[]     and     ' '.''''.[]
```

(Actually, the convention adopted in this book is different from that of Prolog-10. There, a string denotes a list of ASCII codes and not a list of characters, so "string" stands for

```
115.116.114.105.110.103.[ ].
```

Similarly, in Prolog-10 operations for reading and writing characters deal directly with ASCII codes. We refuse to accept these conventions.)

1.1.3. Variables

Objects discussed so far are all, in a sense, constant. Their structure is fixed, we know everything about them and cannot learn anything new. A programming language in which one could specify only such fully defined objects would hardly be interesting. One must be able to use objects whose complete form is defined dynamically during a computation.

In Prolog, the simplest such as-yet-unknown objects are called **variables** (do not confuse them even for a moment with the variables of conventional programming languages!). The term denoting a variable is called a **variable name** (this is also usually called a variable: as with terms and objects, we shall try to maintain the distinction throughout chapter 1). A variable name is written as an identifier starting with an upper case letter or an underscore (e.g. Q, Number_9, _nnn).

A variable is an object whose structure is totally unknown. As a computation progresses, the variable may become **instantiated**, i.e. a more precise description of the object may be determined. The term embodying this description is called the variable's **instantiation**. An instantiated variable is identical with the object described by its instantiation, so it ceases to be a variable, although the object can still be referred to through the variable's name. (In general, a variable may be instantiated also to another variable—we shall soon see the meaning of this.)

There is also an alternative terminology. One says that a **free** (or **unbound**) variable becomes **bound** to another term and is henceforth indistinguishable from that term (which is called its **binding**). The variable becomes **ground** if its binding contains no variables. This terminology brings to mind the process of binding formal parameters to actual parameters. If the formal parameters were not allowed to change their value (as in pure Lisp, say), the similarity would be very close indeed, except that a binding need not be ground.

Intuitively, Prolog variables are somewhat like the variables used in mathematics. When we say that

$$f(x) = e^x + 3x$$

is a function of one variable, we mean that the equation allows us to determine the function's value for any (one) given argument. The variable denotes a single (albeit arbitrary) substitution and is not in itself an object to which values can be assigned.

You can also regard a Prolog variable as an "invisible" pointer. When not free, the pointer is automatically dereferenced in all contexts, so it is impossible to distinguish it from the referenced object: in particular, it is impossible to exchange the object for something else.

1.1.4. Terms

If one thinks of a type as a set of objects, then a term is also a definition of a type. The term Variable3 describes the set of all objects, because a variable can be instantiated to anything. On the other hand, one can have a very precise type specification. For example, the term a.b.c.[] describes a set containing only one object: the list of length 3, whose first element is a, whose second element is b and whose third element is c. There is a wide range of choices between these extremes.