William F. Clocksin

Clause and Effect

Prolog Programming for the Working Programmer

William F. Clocksin

Clause and Effect

Prolog Programming for the Working Programmer

Springer

Dr. William F. Clocksin
Computer Laboratory
University of Cambridge
Pembroke Street
Cambridge CB2 3QG, UK

Computing Reviews Classification (1991): D.1.6

Library of Congress Cataloging-in-Publication Data

Clocksin, W. F. (William F.), 1955–
Clause and effect: Prolog programming for the working programmer / William F. Clocksin.
p. cm.
Includes bibliographical references and index.
ISBN 978-3-540-62971-9 ISBN 978-3-642-58274-5 (eBook)
DOI 10.1007/978-3-642-58274-5
1. Prolog (Computer program language)
QA76.73.P76C565 1997
005.13'--dc21 97-35795
CIP

ISBN 978-3-540-62971-9

Originally published by Springer-Verlag Berlin Heidelberg in 1997

Cover Design: Künkel + Lopka Werbeagentur, Heidelberg
Typesetting: Camera ready by the author
SPIN 11540823 45/3111 – 5 4 3 2 1 – Printed on acid-free paper

Table of Contents

CHAPTER ONE
GETTING STARTED

Prolog is the most widely used programming language to have been inspired by logic programming research. There are a number of reasons for the popularity of Prolog as a programming language:

- Powerful symbol manipulation facilities, including unification with logical variables. Programmers can consider logical variables as named 'holes' in data structures. Unification also serves as the parameter passing mechanism, and provides a constructor and selector of data structures. When combined with recursive procedures and a surface syntax for data structures, the symbol manipulation possibilities of Prolog surpass those of other languages.
- Automatic backtracking provides generate-and-test as the basic control flow model. This is more general than the strict unidirectional sequential flow of control in conventional languages. Although generate-and-test is not appropriate for some applications, other control flow models can be programmed to correspond to the demands of a particular application.
- Program clauses and data structures have the same form. This gives a unified model for representing data as programs and programs as data. Other languages such as Lisp also have this feature.
- The procedural interpretation of clauses, together with a backtracking control structure, provides a convenient way to express and to use nondeterministic procedures. However, the price to pay is the occasional necessity to employ extralogical control features such as fail and cut.
- The relational form of clauses lends the possibility to define 'procedures' that can be used for more than one purpose. It is the

responsibility of the programmer to ensure whether a particular procedure completely implements a given relation.

- A Prolog program can be regarded as a relational database that contains rules as well as facts. It is easy to add and remove information from the database, and to pose sophisticated queries.

1.1 Syntax

Everything (programs and data structures) in Prolog is constructed from terms. There are three kinds of terms: constants, variables and compound terms:

A *constant* names an individual. Constants are further divided into *numbers* and *atoms*. Numbers are the usual signed integer and floating-point numbers. Examples of numbers are 17, 17.2, -65, -0.22E+07. There are several ways to write atoms:

(a) An atom may begin with a lower-case letter which may be followed by digits and letters and may include the underscore character. For example, alpha, gross_pay, john_smith.

(b) An atom may also consist of a sequence of *sign characters*, for example, +, **, /\, =/=.

(c) An atom may be any sequence of characters enclosed in single-quotes, for example, '12Q&A'. Quotes may or may not be necessary, depending on the sequence of characters making up the name. For example, this and 'this' denote the same atom.

A *variable* stands for a term. A variable begins with an upper-case letter or underscore character which may be followed by digits and letters and may include the underscore character. For example, X, Gross_pay, _257. A single underscore character names the *anonymous variable*. An anonymous variable is distinct from any other variable. Its uses will be described later.

A *compound term* names an individual by its parts. A compound term consists of a *functor* and one or more *arguments*. The arguments may be any terms. The arguments are written separated by commas and enclosed in a pair of round brackets. The number of arguments of a compound term is called its *arity*.

For example,

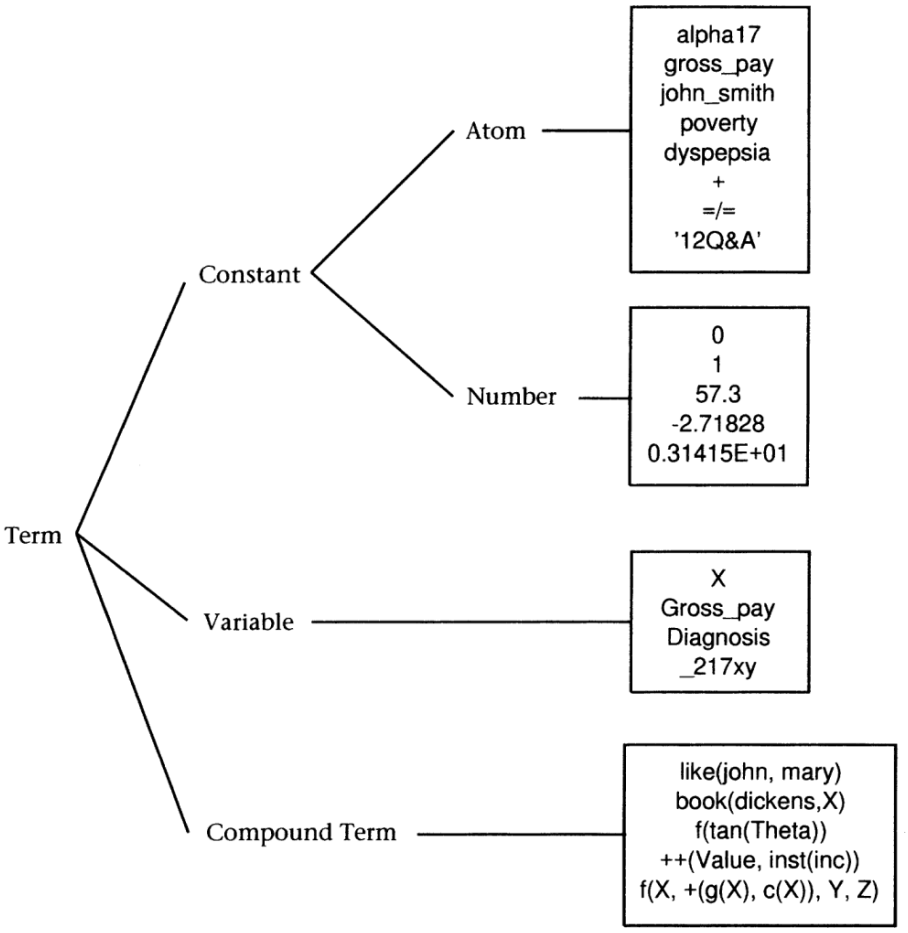
```
likes(john,mary)
```

is a compound term with functor likes of arity 2, and arguments john and mary. The term

```
++(V, inc(a), 128)
```

has functor ++ of arity 3, with arguments V (which is a variable), inc(a), and 128. The argument inc(a) is itself a compound term with functor inc of arity 1 and argument a.

The taxonomy of terms is illustrated as follows, showing examples in the boxes:



1.1.1 Operator Notation

Any atom may be designated as an operator. This does not change the meaning of the atom. The only purpose of operators is for convenience. Whether an atom is designated as an operator only affects how the term containing the atom is parsed. So, if '+' is declared as an infix operator, the term 3+17 is not the same thing as the integer 20. The plus sign

does not automatically mean 'add'. It is simply the functor of a term which could just as well be written +(3,17).

Operators have three properties: position, priority, and associativity. The position of an operator may be prefix, infix, or postfix. Here are some examples:

	Operator Syntax	*Equivalent to*
Prefix:	-a	-(a)
Infix:	5+17	+(5,17)
Postfix:	N!	!(N)

Once an atom is designated as an operator, the operator syntax shown for the above examples may be used. This is equivalent to a Prolog term which also can be written in the usual way.

Associativity and precedence determine how operators bind to arguments relative to other operators in the term. In Prolog, operators may associate on the right, on the left, or prohibit association. The priority is an integer from 1 to 1200, with lower numbers binding more tightly.

Although we won't be declaring any operators for the moment, it is useful to know the built-in declarations of commonly used operators:

Operator	*Class*	*Priority*	*Used for*
:-	xfx	1200	Separating head and body of a clause
,	xfy	1000	Separating goals in a clause
is	xfx	700	Arithmetic evaluation
+ –	yfx	500	
* /	yfx	400	
–	fy	200	

The class is used to encode position and associativity. The 'f' represents an operator in a term in which 'x' and possibly 'y' represent subterms. The yfx declaration for '+' above means that '+' is a binary (arity 2) operator that associates on the left, so for example the term a+b+c is parsed as (a+b)+c.

Notice that the same atom may have more than one operator declaration, so the hyphen '-' may be used as a binary (arity 2) operator and a unary (arity 1) operator, depending on the context in which it appears in the expression. The ':-' and 'is' operators prohibit association

to reduce the risk of syntax errors. In any expression, round brackets may be used to enforce the association of subexpressions in a way that overrides the operator declaration.

1.1.2 Trees

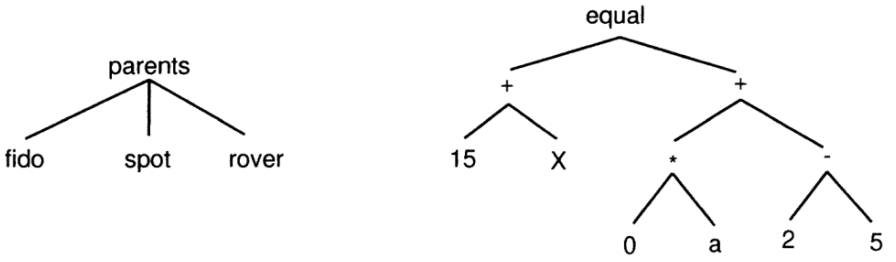
In this book, terms will often be drawn in tree form. Drawing terms this way graphically depicts the syntactic structure of programs and data structures. An *n*-ary compound term is drawn as a node (its functor) having *n* branches (its arguments). Constants and variables appear as the leaves of the tree. For example, the terms

```
parents(fido, spot, rover)
```

and

```
equal(15+X, (0*a)+(2-5))
```

are depicted as:

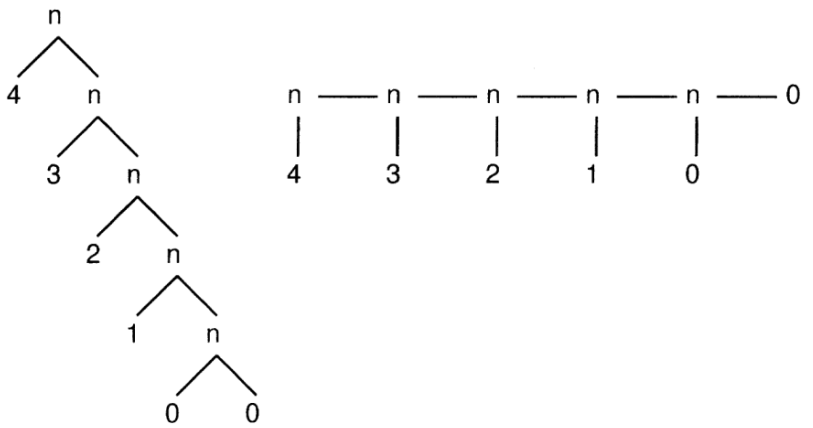


Although the equal term looks like an arithmetic expression, it is important to remember that no arithmetic interpretation is necessarily made. This term is just like any other. Later we shall see how terms that look like arithmetic expressions may be given a special interpretation as arithmetic expressions and be evaluated as such.

Sometimes it is useful to draw trees in a slightly different way. For example, suppose we have a binary (that is, having arity 2) term which is deeply nested on one of the arguments, such as

```
n(4, n(3, n(2, n(1, n(0, 0)))))
```

Although the tree shown on the left looks like a tree, the drawing on the right is usually more convenient, because it shows the linear structure of the term and takes up less vertical space on the page:



You should be able to see that the two drawings depict equivalent data structures.

1.2 Programs

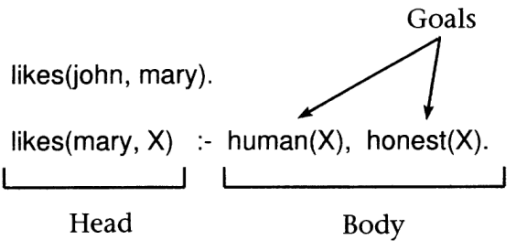
A Prolog *program* consists of a collection of *procedures*. Each procedure defines a particular *predicate*, being a certain relationship between its arguments. A procedure consists of one or more assertions, or *clauses*. It is convenient to think of two kinds of clauses: *facts* and *rules*.

If T is a term of the form H :- B (where H and B are terms and ':-' is an infix functor), then T is called a rule. The term H is called the *head*, and B is called the *body* of the clause. If the :- sign and the body are missing, then T is called a *fact*. When facts and rules are written down to make a program, each one is terminated by a dot (that is, the 'full stop' or 'period' character).

Here is an example of a procedure drink consisting of three clauses, all facts:

```
drink(beer).
drink(milk).
drink(water).
```

If the body of a rule consists of n terms $G_1, G_2, \ldots, G_n$ separated by commas, then all the G are called *goals*. In the next example, procedure likes consists of two clauses: one fact and one rule. The rule is defined in terms of goals human and honest. Procedures defining these would need to appear elsewhere in the program:



Clauses can be given a declarative reading or a procedural reading. For example, the clause

$$H :- G_1, G_2, \ldots, G_n.$$

can be read declaratively as

"That H is provable follows from goals $G_1, G_2, \ldots, G_n$ being provable"

or procedurally as

"In order to execute procedure H, the procedures called by goals $G_1, G_2, \ldots, G_n$ should be executed."

Before turning to the mechanics of program execution, we need to introduce unification.

1.3 Unification

Unification is a basic operation on terms. Two terms *unify* if substitutions can be made for any variables in the terms so that the terms are made identical. If no such substitution exists, then the terms do not unify. Unification is a very powerful technique, and in Prolog, unification is used for passing actual parameters, 'pattern matching', and database access. In the *Programming in Prolog* book, unification is called 'matching', because it is a more descriptive word. This was probably misjudged, because in computer science the word 'matching' is more often used in another sense to mean the less powerful one-way pattern matching such as what the language ML does.

An algorithm for unification proceeds by recursive descent of the two input terms: when attempting to unify two terms, determine whether their corresponding components unify. Ultimately, constants, variables and compund terms will be compared. The rules are as follows:

- *Constants* unify if they are identical. For example, john will unify with john, but john and mary will not unify.
- *Variables* unify with any term, including other variables. When as a result of unification a term has been substituted for a variable, we

(In this book, answers from the Prolog system will appear in italics.)
If the query had been

```
?- drink(2+6).
```

Prolog would answer

no

because there is no way that the term drink(2+6) can be derived from the program. More usefully, if the query is

```
?- drink(X).
```

we are asking the Prolog system to find a value for X that logically follows from the clauses in the program. The first solution according to the above program is

X = beer

There are more solutions, because there are three possible choices from the program that unify with drinks(X). Depending on which Prolog system you are using, there are various ways to ask for the next solution, and we shall see how to do this later.
What about rules? Suppose our program is about who drinks what:

```
drinks(john, water).
drinks(jeremy, milk).
drinks(camilla, beer).
drinks(jeremy, X) :- drinks(john, X).
```

The first three facts are straightforward: the first argument of each drinks clause is a person's name, and the second argument is the drink which is drunk by that person. The last clause, a rule, says that jeremy also drinks anything that john drinks.
By posing various queries, we can find out who drinks what:

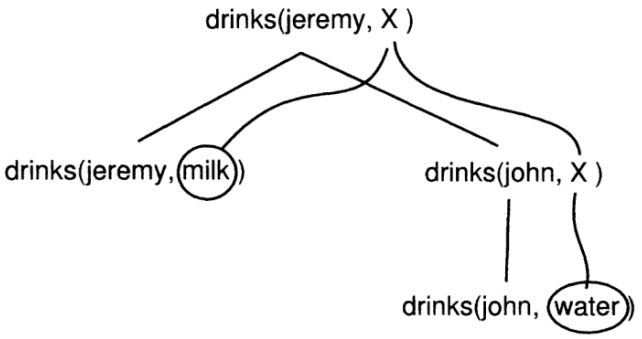
```
?- drinks(camilla, X)
```

X = beer

```
?- drinks(X, gin).
```

no.

The more interesting query is to find out what jeremy drinks. From the above program, you should be able to tell that jeremy drinks milk (by virtue of clause 2), and that he also drinks water (because according to clause 4, he drinks whatever john drinks, and according to clause 1, john drinks water). All the possibilities for what jeremy drinks can be depicted in this 'proof tree':

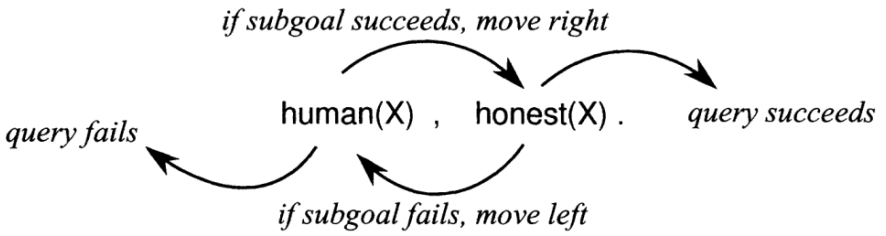


The straight lines show how one goal sets up another goal. The curved lines show the alternative values for X for different solutions.

The general case is of a sequence of queries that must be satisfied. The subgoals in a query are separated by commas. Prolog begins from left to right attempting to satisfy each query. If a subgoal succeeds, Prolog tries the next one on the right. If a subgoal fails, Prolog goes back to the goal on the left to see if there are any more solutions. So, if we wish to test whether some X is human and honest, the query

```
?- human(X), honest(X).
```

is executed. This picture might help:



This way of showing how success and failure propagate though a sequence of subgoals works for any number of subgoals in the body of a clause.