
INTRODUCTION TO LOGIC PROGRAMMING

- SRIVIDYA KONA

Imperative Programming

- How to Solve, rather than what to solve
- Requires the programmer to specify an algorithm to be run
- Sequence of statements
- Makes the algorithm explicit and leaves the goal implicit
- Example: factorial function

Why did the Computer Programmer die in the shower ?

He was just following the instructions on the shampoo bottle:

“ Lather, Rinse, Repeat ”

Following a sequence of instructions to the letter is the essence of imperative programming.

Declarative Programming

- What to Solve, rather than how to solve
- Requires the programmer to specify just the problem, the language compiler figures the algorithm
- Collection of definitions
 - functions or predicates
- Makes the goal explicit and leaves the algorithm implicit
- Example: factorial function

Imperative Vs Declarative Programming

Algorithm = Logic + Control

- Imperative programming needs both logic & control whereas Declarative programming needs just logic, it figures control on its own.
- Declarative programming is a higher level programming paradigm than Imperative.
- Imperative programming is more closer to the popular (Von Neumann) architecture of a computer, whereas Declarative Programming is independent of the architecture.

Logic

- Classical Logic - A means of clarifying / formalizing the human thought process

Example:

Aristotle likes cookies, and

Plato is a friend of anyone who likes cookies

imply that *Plato is a friend of Aristotle*

- Symbolic Logic - A shorthand for classical logic, plus many useful results

Example:

a1 : likes(aristotle, cookies)

a2 : for all X, likes(X, cookies) \rightarrow friend(plato,X)

t1 : friend(plato, aristotle)

$T[a1, a2] \vdash t1$

- Can logic be used in computation and AI?
 - To represent the problem?
 - To solve the problem?

Logic

- Convenient to represent the problem in logic, but which Logic?
 - propositional
 - predicate calculus (first order)
 - higher-order logics
 - modal logics
 - λ calculus ...
- Which reasoning procedure?
 - natural deduction, classical methods
 - resolution
 - Prawitz/Bibel
 - bottom-up fixpoint
 - rewriting
 - narrowing ...

History of Logic Programming

■ 60's:

- Greene: problem solving using Logic
- Robinson: linear resolution technique for inferencing

■ 70's:

- **(early)** Kowalski: procedural interpretation of Horn clause logic.
A if B_1 and B_2 and ... and B_n can be interpreted as
to solve (execute) A, solve (execute) B_1 and B_2 and, ..., B_n
- **(early)** Colmerauer: specialized theorem prover (Fortran)
embedding the procedural interpretation: Prolog (Programmation
et Logique).
- In the U.S.: “next-generation AI languages” of the time (i.e.
planner) seen as inefficient and difficult to control.
- **(late)** D.H.D. Warren develops DEC-10 Prolog compiler, almost
completely written in Prolog. Very efficient (same as LISP), very
useful control built-ins.

History of Logic Programming

■ Late 80's, 90's

- ❑ Major research in the basic paradigms and advanced implementation techniques: Japan (Fifth Generation Project), US (MCC), Europe (ECRC, ESPRIT projects).
- ❑ Numerous commercial Prolog implementations, programming books, and a *de facto* standard, the Edinburgh Prolog family.
- ❑ First parallel and concurrent logic programming systems.
- ❑ CLP – Constraint Logic Programming: Major extension and had many new applications areas.
- ❑ 1995: ISO Prolog standard.

Logic Programming

- Many commercial CLP systems with fielded applications.
- Extensions to full higher order logic, inclusion of functional programming, etc.
- Highly optimizing compilers, automatic parallelism, automatic debugging.
- Concurrent constraint programming systems.
- Distributed systems.
- Object oriented dialects.
- Applications
 - Natural language processing
 - Scheduling/Optimization problems
 - AI related problems
 - (Multi) agent systems programming.
 - Program analyzers

Prolog

PROgramming in LOGic

- It is the most widely used logic programming language
- Its development started in 1970 and it was result of a collaboration between researchers from Marseille, France, and Edinburgh, Scotland

What is Prolog good for ?

- Knowledge representation
- Natural language processing
- State-space searching (Rubik's cube)
- Logic problems
- Theorem provers
- Expert systems, deductive databases
- Agents

Programming in Prolog

Used to solve problems involving:

- Objects, and
- Objects and their relationships

Writing Prolog programs involves:

- Declaring Facts (unconditional truths)
- Defining Rules (conditional truths)
- Asking Questions (goal/theorem)

Programming in Prolog

- A logic program comprises
 - collection of axioms (facts and inference rules)
 - one or more goal statements
- Axioms are a *theory*
- Goal statement is a *theorem*
- Computation is *deduction* to prove the theorem within the theory (using the axioms)
- Interpreter tries to find a collection of axioms and inference steps that imply the goal

Programming in Prolog

- A predicate is a tuple:
 - `pred(a,b,c)`
- Tuple is an element in a relation
- Prolog program is a specification of a relation (in contrast to functional programming)
 - `brother (sam, bill)`
 - `brother (sam, bob)`
 - brother is not a function, since it maps “sam” to two different range elements. brother is a relation*
- Relations are n-ary, not just binary
 - `family(jane,sam,[ann,tim,sean])`

Relations - Examples

(2,4),(3,9),(4,16),(5,25),(6,36),(7,49), ... “square”

(t,t,f), (t,f,t), (f,t,t), (f,f,f) ... “xor” boolean algebra

(smith, bob, 43, male, richmond, plumber),

(smith, bob, 27, male, richmond, lawyer),

(jones, alice, 31, female, durham, doctor),

(jones, lisa, 12, female, raleigh, student),

(smith, chris, 53, female, durham, teacher)

Prolog

- Prolog programs define relations and allow you to query them to extract various tuples from the relations
- Relations are defined using predicates. Example:
 - *square(A,B) is true if B is A*A*
 - *pred(B,H,A) is true if A is $\frac{1}{2}$ B*H*
- Prolog uses Horn clauses for explicit definition (facts) and for rules

Directionality

- Parameters are not directional (in, out)
 - Prolog programs can be run “in reverse”
- (2,4), (3,9),(4,16), (5,25),(6,36),(7,49), ... “square”
 - can ask `square(X,9)`
“what number, when squared, gives 9”
 - can ask `square(4,X)`
“what number is the square of 4”
 - can ask `square(4,16)`
“is 16 the square of 4”

Prolog Syntax

- **Variables:** start with uppercase character (or “_”), may include “_” and digits:
Examples: X, Ys, A_num, _, _x, _22
- **Constants:** lowercase first character, may include “_” and digits. Also, numbers and some special characters. Any quoted string.
Examples: a, dog, a_big_cat, 23, 'Hungry man', []
- **Structures:** a functor (like a constant name) followed by a fixed number of arguments between parentheses:
Example: date(monday, Month, 1994)
- Arguments can in turn be variables, constants and structures.
- **Arity:** is the number of arguments of a structure. Functors are represented as *name/arity*. A constant can be seen as a structure with arity zero.
- Variables, constants, and structures as a whole are called *terms* (they are the terms of a “first-order language”): the *data structures* of a logic program.

Prolog Syntax

- **Atoms:** an expression of the form $p(t_1, t_2, \dots, t_n)$
 - p is the atom's predicate symbol (same convention as with functors), n is its arity, and t_1, t_2, \dots, t_n are *terms*.
 - The predicate symbol of an atom is also represented as p/n .
- Atoms and terms are syntactically identical. They are distinguished by context:
 - if $dog(name(barry), color(black))$ is an atom
then $name(barry)$ and $color(black)$ are terms
 - if $color(dog(barry,black))$ is an atom
then $dog(barry,black)$ is a term
- Atoms cannot appear inside terms; terms are the arguments of atoms.
- **Literals:** A literal is a positive or negative (negated) atom.

Prolog Syntax

- *Examples of terms:*

Term	Type	Main functor
dad	constant	dad/0
time(min,sec)	structure	time/2
pair(calvin,tiger(Hobbes))	structure	pair/2
A_num	variable	--
Tee(Alf,rob)	illegal	--

- *Functors can be defined as prefix, postfix, or infix operators:*
 - $a + b$ is the term '+'(a,b) if +/2 declared infix
 - $-b$ is the term '-'(b) if -/1 declared prefix
 - $a < b$ is the term '<'(a,b) if </2 declared infix
 - john father mary is the term father(john,mary) if father/2 declared infix
- We assume that some such operator definitions are always preloaded.

Prolog Syntax

- Rules: A rule is an expression of the form:

$$\begin{aligned} p_0(t_1, t_2, \dots, t_{n_0}) \leftarrow \\ p_1(t^1_1, t^1_2, \dots, t^1_{n_1}), \\ \dots \\ p_m(t^m_1, t^m_2, \dots, t^m_{n_m}). \end{aligned}$$

- The expression to the left of the arrow has to be an *atom* (no negation) and is called the *head* of the rule.
- To the right of the arrow are *literals* and form the *body* of the rule.
- Literals in the body of a rule are also called procedure calls.
- *Example:*

```
meal(First, Second, Third) ←  
    appetizer(First),  
    main_dish(Second),  
    dessert(Third).
```

Prolog Syntax (facts, clauses, predicates)

- **Facts:** A fact is an expression of the form

$p(t_1, t_2, \dots, t_n) \leftarrow .$

(i.e., a fact is a rule with an empty body).

Examples:

`dog(name(barry), color(black)).`

`friends('Ann', 'John').`

- Rules and facts are both called clauses.

- **Predicates:** all clauses whose heads have the same name and arity form a predicate (or *procedure*) definition.

Example:

`pet(spot).`

`pet(X) ← animal(X), barks(X).`

`pet(X) ← animal(X), meows(X).`

Predicate `pet/1` has three clauses.

Of those, one is a fact and two are rules.

Declarative Meaning of Facts & Rules

The declarative meaning is the corresponding one in first order logic

- **Rules:**
 - Commas in rule bodies represent conjunction, i.e.,
 $p \leftarrow p_1, \dots, p_m$ represents $p \leftarrow p_1 \wedge, \dots, \wedge p_m$
 - “ \leftarrow ” represents logical implication.

(i.e), a rule $p \leftarrow p_1, \dots, p_m$ means
“if p_1 and ... and p_m are true, then p is true”

Example: the rule $pet(X) \leftarrow animal(X), barks(X)$.
can be read as “ X is a pet if it is an animal and it barks”.

- **Facts:** state things that are true.
(Note that a fact p can be seen as the rule “ $p \leftarrow true$.”)

Example: the fact $animal(spot) \leftarrow$.
can be read as “spot is an animal”.

Declarative Meaning of Predicates

- **Predicates:** clauses in the same predicate

$$p \leftarrow p_1, \dots, p_n.$$

$$p \leftarrow q_1, \dots, q_m.$$

provide different *alternatives* (for p).

Example: the rules

$$pet(X) \leftarrow animal(X), barks(X).$$

$$pet(X) \leftarrow animal(X), meows(X).$$

express two ways for X to be a pet.

- **Variable Scope:** the X variables in the two clauses above are different, even if they have the same name. Variables are *local to clauses* (and are *renamed* any time a clause is used).

Prolog Programs

- **Logic Program:** a set of predicates.

- *Example:*

```
animal(spot).           barks(spot).
animal(barry).         meows(barry).
animal(hobbes).       roars(hobbes).
pet(X) ← animal(X), barks(X).
pet(X) ← animal(X), meows(X).
```

- **Query:** an expression of the form: $\leftarrow p_1(t^1_1, \dots, t^1_{n_1}), \dots, p_n(t^n_1, \dots, t^n_{n_m})$.
(i.e., a clause without a head). A query represents a *question to the program*.
Example: $\leftarrow \text{pet}(X)$.
- **Execution:** given a program and a query, *executing* the logic program is *attempting to find an answer to the query*. *Example:* above, the system will try to find a “substitution” for X which makes pet(X) true. Intuitively, we have two possible answers: spot and barry.
- The *declarative semantics* does not specify *how* this is done – this is the role of the operational semantics.

Prolog Syntax

- Left arrow “←” replaced by :- in rules, .in facts.

Example:

$a \leftarrow b, c.$ becomes $a :- b, c.$

$a \leftarrow .$ becomes simply $a.$

- Comments:
 - Using “%”: rest of line is a comment.
 - Using “/* ... */”: everything in between is a comment.
- In Prolog terminology, “atom” refers to the *constants* in the program.

Unification (Parameter-passing mechanism)

- Prolog associates (binds) variables and values using a process known as *unification*.
 - Variable that receive a value are said to be *instantiated*
- Unification rules
 - A constant unifies only with itself
 - Two structures unify if and only if they have the same functor symbol and the same number of arguments, and the corresponding arguments unify recursively
 - An unbound variable unifies with anything

Unification

- Unifying two terms is finding (the minimal) values for the variables in those terms which make them syntactically equal.
- Only variables can be given values!
- Two terms can be made identical only by making their arguments identical.
- *Example:*

Unify A	With B	Using θ
dog	dog	Φ
X	Y	$\{X = Y\}$
X	a	$\{X = a\}$
$f(X, g(t))$	$f(m(h), g(M))$	$\{X=m(h), M=t\}$
$f(X, g(t))$	$f(m(h), t(M))$	Impossible (1)
$f(X, X)$	$f(Y, b(Y))$	Impossible (2)

- (1) Structures with different name and/or arity cannot be unified.
- (2) A variable cannot be given as value a term which contains that variable, because it would create an infinite term. Checking for this is known as the occurs check.

Database Programming

- Database: A collection of Prolog facts
- Prolog draws knowledge from these facts
- Programmer is responsible for the accuracy of the facts
- Questions are answered based on these facts

Database Programming - Example

Facts in the database:

likes(john,apple).

likes(mary,apple).

likes(john,fish).

likes(joe,mary).

Questions:

?-likes(joe, money).

no

?-likes(joe,mary).

yes

?-likes(john,X).

X = apple ;

X = fish ;

no.

Deductive Databases

- “Deductive databases” uses these ideas to develop *logic-based databases*.
- They have syntactic restrictions (i.e., a subset of definite programs) are used (e.g. “Datalog” – no functors, no existential variables).