Prolog

Prolog is a logic programming language associated with artificial intelligence and computational linguistics.[1][2][3]

Prolog has its roots in first-order logic, a formal logic, and unlike many other programming languages, Prolog is intended primarily as a declarative programming language: the program logic is expressed in terms of relations, represented as facts and rules. A computation is initiated by running a *query* over these relations.[4]

The language was developed and implemented in Marseille, France, in 1972 by Alain Colmerauer with Philippe Roussel, based on Robert Kowalski's procedural interpretation of Horn clauses.[5][6]

Prolog was one of the first logic programming languages[7] and remains the most popular such language today, with several free and commercial implementations available. The language has been used for theorem proving,[8] expert systems,[9] term rewriting,[10] type systems,[11] and automated planning,[12] as well as its original intended field of use, natural language processing.[13][14] Modern Prolog environments support the creation of graphical user interfaces, as well as administrative and networked applications.

Prolog is well-suited for specific tasks that benefit from rule-based logical queries such as searching databases, voice control systems, and filling templates.

Contents

Prolog

Paradigm	Logic
Designed by	Alain Colmerauer, Robert Kowalski
First appeared	1972
Stable release	Part 1: General core-Edition 1 (June 1995) Part 2: Modules-Edition 1 (June 2000)
Typing discipline	Untyped (its single data type is "term")
Filename extensions	`.pl`, `.pro`, `.P`
Website	Part 1: www.iso.org/standard/21413.html (https://www.iso.org/standard/21413.html) Part 2: www.iso.org/standard/20775.html (https://www.iso.org/standard/20775.html)
Major implementations	
B-Prolog, Ciao, ECLiPSe, GNU Prolog, Jekejeke Prolog (http://www.jekejeke.ch/), Poplog Prolog, P#, Quintus Prolog (https://quintus.sics.se/), SICStus, Strawberry, SWI-Prolog, Tau Prolog (http://tau-prolog.org/), tuProlog, WIN-PROLOG, XSB, YAP.	
Dialects	
ISO Prolog, Edinburgh Prolog	
Influenced by	

Planner
Influenced
CHR, Clojure, Datalog, Erlang, KL0, KL1, Mercury, Oz, Strand, Visual Prolog, XSB
Prolog at Wikibooks

Syntax and semantics

In Prolog, program logic is expressed in terms of relations, and a computation is initiated by running a *query* over these relations. Relations and queries are constructed using Prolog's single data type, the *term*.[4] Relations are defined by *clauses*. Given a query, the Prolog engine attempts to find a resolution refutation of the negated query. If the negated query can be refuted, i.e., an instantiation for all free variables is found that makes the union of clauses and the singleton set consisting of the negated query false, it follows that the original query, with the found instantiation applied, is a logical consequence of the program. This makes Prolog (and other logic programming languages) particularly useful for database, symbolic mathematics, and language parsing applications. Because Prolog allows impure predicates, checking the truth value of certain special predicates may have some deliberate side effect, such as printing a value to the screen. Because of this, the programmer is permitted to use some amount of conventional imperative programming when the logical paradigm is inconvenient. It has a purely logical subset, called "pure Prolog", as well as a number of extralogical features.

Data types

Prolog's single data type is the *term*. Terms are either *atoms*, *numbers*, *variables* or *compound terms*.

- An **atom** is a general-purpose name with no inherent meaning. Examples of atoms include `x`, `red`, `'Taco'`, and `'some atom'`.
- **Numbers** can be floats or integers. ISO standard compatible Prolog systems can check the Prolog flag "bounded". Most of the major Prolog systems support arbitrary length integer numbers.
- **Variables** are denoted by a string consisting of letters, numbers and underscore characters, and beginning with an upper-case letter or underscore. Variables closely resemble variables in logic in that they are placeholders for arbitrary terms.
- A **compound term** is composed of an atom called a "functor" and a number of "arguments", which are again terms. Compound terms are ordinarily written as a functor followed by a comma-separated list of argument terms, which is contained in parentheses. The number of arguments is called the term's arity. An atom can be regarded as a compound term with arity zero. An example of a compound term is `person_friends(zelda,[tom,jim])`.

Special cases of compound terms:

- A *List* is an ordered collection of terms. It is denoted by square brackets with the terms separated by commas, or in the case of the empty list, by `[]`. For example, `[1,2,3]` or `[red,green,blue]`.
- *Strings*: A sequence of characters surrounded by quotes is equivalent to either a list of (numeric) character codes, a list of characters (atoms of length 1), or an atom depending on the value of the Prolog flag `double_quotes`. For example, `"to be, or not to be"`.[15]

ISO Prolog provides the `atom/1`, `number/1`, `integer/1`, and `float/1` predicates for type-checking.[16]

Rules and facts

Prolog programs describe relations, defined by means of clauses. Pure Prolog is restricted to Horn clauses. There are two types of clauses: facts and rules. A rule is of the form

```
Head :- Body.
```

and is read as "Head is true if Body is true". A rule's body consists of calls to predicates, which are called the rule's **goals**. The built-in logical operator `,/2` (meaning an arity 2 operator with name `,`) denotes conjunction of goals, and `;/2` denotes disjunction. Conjunctions and disjunctions can only appear in the body, not in the head of a rule.

Clauses with empty bodies are called **facts**. An example of a fact is:

```
cat(crookshanks).
```

which is equivalent to the rule:

```
cat(crookshanks) :- true.
```

The built-in predicate `true/0` is always true.

Given the above fact, one can ask:

is crookshanks a cat?

```
?- cat(crookshanks).
Yes
```

what things are cats?

```
?- cat(X).
X = crookshanks
```

Clauses with bodies are called **rules**. An example of a rule is:

```
animal(X) :- cat(X).
```

If we add that rule and ask *what things are animals?*

```
?- animal(X).
X = crookshanks
```

Due to the relational nature of many built-in predicates, they can typically be used in several directions. For example, `length/2` can be used to determine the length of a list (`length(List, L)`, given a list `List`) as well as to generate a list skeleton of a given length (`length(X, 5)`), and also to generate both list skeletons and their lengths together (`length(X, L)`). Similarly, `append/3` can be used both to append two lists (`append(ListA, ListB, X)` given lists `ListA` and `ListB`) as well as to split a given list into parts (`append(X, Y, List)`, given a list `List`). For this reason, a comparatively small set of library predicates suffices for many Prolog programs.

As a general purpose language, Prolog also provides various built-in predicates to perform routine activities like input/output, using graphics and otherwise communicating with the operating system. These predicates are not given a relational meaning and are only useful for the side-effects they exhibit on the system. For example, the predicate `write/1` displays a term on the screen.

Execution

Execution of a Prolog program is initiated by the user's posting of a single goal, called the query. Logically, the Prolog engine tries to find a resolution refutation of the negated query. The resolution method used by Prolog is called SLD resolution. If the negated query can be refuted, it follows that the query, with the appropriate variable bindings in place, is a logical consequence of the program. In that case, all generated variable bindings are reported to the user, and the query is said to have succeeded. Operationally, Prolog's execution strategy can be thought of as a generalization of function calls in other languages, one difference being that multiple clause heads can match a given call. In that case, the system creates a choice-point, unifies the goal with the clause head of the first alternative, and continues with the goals of that first

alternative. If any goal fails in the course of executing the program, all variable bindings that were made since the most recent choice-point was created are undone, and execution continues with the next alternative of that choice-point. This execution strategy is called chronological backtracking. For example:

```
mother_child(trude, sally).

father_child(tom, sally).
father_child(tom, erica).
father_child(mike, tom).

sibling(X, Y)      :- parent_child(Z, X), parent_child(Z, Y).

parent_child(X, Y) :- father_child(X, Y).
parent_child(X, Y) :- mother_child(X, Y).
```

This results in the following query being evaluated as true:

```
?- sibling(sally, erica).
Yes
```

This is obtained as follows: Initially, the only matching clause-head for the query `sibling(sally, erica)` is the first one, so proving the query is equivalent to proving the body of that clause with the appropriate variable bindings in place, i.e., the conjunction `(parent_child(Z,sally), parent_child(Z,erica))`. The next goal to be proved is the leftmost one of this conjunction, i.e., `parent_child(Z, sally)`. Two clause heads match this goal. The system creates a choice-point and tries the first alternative, whose body is `father_child(Z, sally)`. This goal can be proved using the fact `father_child(tom, sally)`, so the binding `Z = tom` is generated, and the next goal to be proved is the second part of the above conjunction: `parent_child(tom, erica)`. Again, this can be proved by the corresponding fact. Since all goals could be proved, the query succeeds. Since the query contained no variables, no bindings are reported to the user. A query with variables, like:

```
?- father_child(Father, Child).
```

enumerates all valid answers on backtracking.

Notice that with the code as stated above, the query `?- sibling(sally, sally).` also succeeds. One would insert additional goals to describe the relevant restrictions, if desired.

Loops and recursion

Iterative algorithms can be implemented by means of recursive predicates.[17]

Negation

The built-in Prolog predicate `\+/1` provides negation as failure, which allows for non-monotonic reasoning. The goal `\+ illegal(X)` in the rule

```
legal(X) :- \+ illegal(X).
```

is evaluated as follows: Prolog attempts to prove `illegal(X)`. If a proof for that goal can be found, the original goal (i.e., `\+ illegal(X)`) fails. If no proof can be found, the original goal succeeds. Therefore, the `\+/1` prefix operator is called the "not provable" operator, since the query `?- \+ Goal.` succeeds if Goal is not provable. This kind of negation is sound if its argument is "ground" (i.e. contains no variables). Soundness is lost if the argument contains variables and the proof procedure is complete. In particular, the query `?- legal(X).` now cannot be used to enumerate all things that are legal.

Programming in Prolog

In Prolog, loading code is referred to as *consulting*. Prolog can be used interactively by entering queries at the Prolog prompt `?-`. If there is no solution, Prolog writes `no`. If a solution exists then it is printed. If there are multiple solutions to the query, then these can be requested by entering a semi-colon `;`. There are guidelines on good programming practice to improve code efficiency, readability and maintainability.[18]

Here follow some example programs written in Prolog.

Hello World

An example of a query:

```
?- write('Hello World!'), nl.
Hello World!
true.

?-
```

Compiler optimization

Any computation can be expressed declaratively as a sequence of state transitions. As an example, an optimizing compiler with three optimization passes could be implemented as a relation between an initial program and its optimized form:

```
program_optimized(Prog0, Prog) :-
    optimization_pass_1(Prog0, Prog1),
    optimization_pass_2(Prog1, Prog2),
    optimization_pass_3(Prog2, Prog).
```

or equivalently using DCG notation:

```
program_optimized --> optimization_pass_1, optimization_pass_2, optimization_pass_3.
```

Quicksort

The quicksort sorting algorithm, relating a list to its sorted version:

```
partition([], _, [], []).
partition([X|Xs], Pivot, Smalls, Bigs) :-
    (   X @< Pivot ->
        Smalls = [X|Rest],
        partition(Xs, Pivot, Rest, Bigs)
```

```
    ;   Bigs = [X|Rest],
        partition(Xs, Pivot, Smalls, Rest)
    ).

quicksort([])     --> [].
quicksort([X|Xs]) -->
    { partition(Xs, X, Smaller, Bigger) },
    quicksort(Smaller), [X], quicksort(Bigger).
```

Design patterns

A design pattern is a general reusable solution to a commonly occurring problem in software design. Some design patterns in Prolog are skeletons, techniques,[19][20] cliches,[21] program schemata,[22] logic description schemata,[23] and higher order programming.[24]

Higher-order programming

A higher-order predicate is a predicate that takes one or more other predicates as arguments. Although support for higher-order programming takes Prolog outside the domain of first-order logic, which does not allow quantification over predicates,[25] ISO Prolog now has some built-in higher-order predicates such as `call/1`, `call/2`, `call/3`, `findall/3`, `setof/3`, and `bagof/3`.[26] Furthermore, since arbitrary Prolog goals can be constructed and evaluated at run-time, it is easy to write higher-order predicates like `maplist/2`, which applies an arbitrary predicate to each member of a given list, and `sublist/3`, which filters elements that satisfy a given predicate, also allowing for currying.[24]

To convert solutions from temporal representation (answer substitutions on backtracking) to spatial representation (terms), Prolog has various all-solutions predicates that collect all answer substitutions of a given query in a list. This can be used for list comprehension. For example, perfect numbers equal the sum of their proper divisors:

```
perfect(N) :-
    between(1, inf, N), U is N // 2,
    findall(D, (between(1,U,D), N mod D =:= 0), Ds),
    sumlist(Ds, N).
```

This can be used to enumerate perfect numbers, and also to check whether a number is perfect.

As another example, the predicate `maplist` applies a predicate `P` to all corresponding positions in a pair of lists:

```
maplist(_, [], []).
maplist(P, [X|Xs], [Y|Ys]) :-
   call(P, X, Y),
   maplist(P, Xs, Ys).
```

When `P` is a predicate that for all `X`, `P(X,Y)` unifies `Y` with a single unique value, `maplist(P, Xs, Ys)` is equivalent to applying the map function in functional programming as `Ys = map(Function, Xs)`.

Higher-order programming style in Prolog was pioneered in HiLog and λProlog.

Modules

For programming in the large, Prolog provides a module system. The module system is standardised by ISO.[27] However, not all Prolog compilers support modules, and there are compatibility problems between the module systems of the major Prolog compilers.[28] Consequently, modules written on one Prolog compiler will not necessarily work on others.

Parsing

There is a special notation called definite clause grammars (DCGs). A rule defined via `-->/2` instead of `:-/2` is expanded by the preprocessor (`expand_term/2`, a facility analogous to macros in other languages) according to a few straightforward rewriting rules, resulting in ordinary Prolog clauses. Most notably, the rewriting equips the predicate with two additional arguments, which can be used to implicitly thread state around, analogous to monads in other languages. DCGs are often used to write parsers or list generators, as they also provide a convenient interface to difference lists.

Meta-interpreters and reflection

Prolog is a homoiconic language and provides many facilities for reflection. Its implicit execution strategy makes it possible to write a concise meta-circular evaluator (also called *meta-interpreter*) for pure Prolog code:

```
solve(true).
solve((Subgoal1,Subgoal2)) :-
    solve(Subgoal1),
    solve(Subgoal2).
solve(Head) :-
    clause(Head, Body),
    solve(Body).
```

where `true` represents an empty conjunction, and `clause(Head, Body)` unifies with clauses in the database of the form `Head :- Body`.

Since Prolog programs are themselves sequences of Prolog terms (`:-/2` is an infix operator) that are easily read and inspected using built-in mechanisms (like `read/1`), it is possible to write customized interpreters that augment Prolog with domain-specific features. For example, Sterling and Shapiro present a meta-interpreter that performs reasoning with uncertainty, reproduced here with slight modifications:[29]:330

```
solve(true, 1) :- !.
solve((Subgoal1,Subgoal2), Certainty) :-
    !,
    solve(Subgoal1, Certainty1),
    solve(Subgoal2, Certainty2),
    Certainty is min(Certainty1, Certainty2).
solve(Goal, 1) :-
    builtin(Goal), !,
    Goal.
solve(Head, Certainty) :-
    clause_cf(Head, Body, Certainty1),
    solve(Body, Certainty2),
    Certainty is Certainty1 * Certainty2.
```

This interpreter uses a table of built-in Prolog predicates of the form[29]:327

```
builtin(A is B).
builtin(read(X)).
% etc.
```

and clauses represented as `clause_cf(Head, Body, Certainty)`. Given those, it can be called as `solve(Goal, Certainty)` to execute `Goal` and obtain a measure of certainty about the result.

Turing completeness

Pure Prolog is based on a subset of first-order predicate logic, Horn clauses, which is Turing-complete. Turing completeness of Prolog can be shown by using it to simulate a Turing machine:

```
turing(Tape0, Tape) :-
    perform(q0, [], Ls, Tape0, Rs),
    reverse(Ls, Ls1),
    append(Ls1, Rs, Tape).

perform(qf, Ls, Ls, Rs, Rs) :- !.
perform(Q0, Ls0, Ls, Rs0, Rs) :-
    symbol(Rs0, Sym, RsRest),
    once(rule(Q0, Sym, Q1, NewSym, Action)),
    action(Action, Ls0, Ls1, [NewSym|RsRest], Rs1),
    perform(Q1, Ls1, Ls, Rs1, Rs).

symbol([], b, []).
symbol([Sym|Rs], Sym, Rs).

action(left, Ls0, Ls, Rs0, Rs) :- left(Ls0, Ls, Rs0, Rs).
action(stay, Ls, Ls, Rs, Rs).
action(right, Ls0, [Sym|Ls0], [Sym|Rs], Rs).

left([], [], Rs0, [b|Rs0]).
left([L|Ls], Ls, Rs, [L|Rs]).
```

A simple example Turing machine is specified by the facts:

```
rule(q0, 1, q0, 1, right).
rule(q0, b, qf, 1, stay).
```

This machine performs incrementation by one of a number in unary encoding: It loops over any number of "1" cells and appends an additional "1" at the end. Example query and result:

```
?- turing([1,1,1], Ts).
Ts = [1, 1, 1, 1] ;
```

This illustrates how any computation can be expressed declaratively as a sequence of state transitions, implemented in Prolog as a relation between successive states of interest.

Implementation

ISO Prolog

The ISO Prolog standard consists of two parts. ISO/IEC 13211-1,[26][30] published in 1995, aims to standardize the existing practices of the many implementations of the core elements of Prolog. It has clarified aspects of the language that were previously ambiguous and leads to portable programs. There are three corrigenda: Cor.1:2007,[31] Cor.2:2012,[32] and Cor.3:2017.[33] ISO/IEC 13211-2,[26] published in

2000, adds support for modules to the standard. The standard is maintained by the ISO/IEC JTC1/SC22/WG17[34] working group. ANSI X3J17 is the US Technical Advisory Group for the standard.[35]

Compilation

For efficiency, Prolog code is typically compiled to abstract machine code, often influenced by the register-based Warren Abstract Machine (WAM) instruction set.[36] Some implementations employ abstract interpretation to derive type and mode information of predicates at compile time, or compile to real machine code for high performance.[37] Devising efficient implementation methods for Prolog code is a field of active research in the logic programming community, and various other execution methods are employed in some implementations. These include clause binarization and stack-based virtual machines.

Tail recursion

Prolog systems typically implement a well-known optimization method called tail call optimization (TCO) for deterministic predicates exhibiting tail recursion or, more generally, tail calls: A clause's stack frame is discarded before performing a call in a tail position. Therefore, deterministic tail-recursive predicates are executed with constant stack space, like loops in other languages.

Term indexing

Finding clauses that are unifiable with a term in a query is linear in the number of clauses. Term indexing uses a data structure that enables sub-linear-time lookups.[38] Indexing only affects program performance, it does not affect semantics. Most Prologs only use indexing on the first term, as indexing on all terms is expensive, but techniques based on *field-encoded words* or *superimposed codewords* provide fast indexing across the full query and head.[39][40]

Hashing

Some Prolog systems, such as WIN-PROLOG and SWI-Prolog, now implement hashing to help handle large datasets more efficiently. This tends to yield very large performance gains when working with large corpora such as WordNet.

Tabling

Some Prolog systems, (B-Prolog, XSB, SWI-Prolog, YAP, and Ciao), implement a memoization method called *tabling*, which frees the user from manually storing intermediate results. Tabling is a space–time tradeoff; execution time can be reduced by using more memory to store intermediate results:[41][42]

> Subgoals encountered in a query evaluation are maintained in a table, along with answers to these subgoals. If a subgoal is re-encountered, the evaluation reuses information from the table rather than re-performing resolution against program clauses.[43]

Tabling can be extended in various directions. It can support recursive predicates through SLG-resolution or linear tabling. In a multi-threaded Prolog system tabling results could be kept private to a thread or shared among all threads. And in incremental tabling, tabling might react to changes.

Implementation in hardware

During the Fifth Generation Computer Systems project, there were attempts to implement Prolog in hardware with the aim of achieving faster execution with dedicated architectures.[44][45][46] Furthermore, Prolog has a number of properties that may allow speed-up through parallel execution.[47] A more recent approach has been to compile restricted Prolog programs to a field programmable gate array.[48] However, rapid progress in general-purpose hardware has consistently overtaken more specialised architectures.

Limitations

Although Prolog is widely used in research and education, Prolog and other logic programming languages have not had a significant impact on the computer industry in general.[49] Most applications are small by industrial standards, with few exceeding 100,000 lines of code.[49][50] Programming in the large is considered to be complicated because not all Prolog compilers support modules, and there are compatibility problems between the module systems of the major Prolog compilers.[28] Portability of Prolog code across implementations has also been a problem, but developments since 2007 have meant: "the portability within the family of Edinburgh/Quintus derived Prolog implementations is good enough to allow for maintaining portable real-world applications."[51]

Software developed in Prolog has been criticised for having a high performance penalty compared to conventional programming languages. In particular, Prolog's non-deterministic evaluation strategy can be problematic when programming deterministic computations, or when even using "don't care non-determinism" (where a single choice is made instead of backtracking over all possibilities). Cuts and other language constructs may have to be used to achieve desirable performance, destroying one of Prolog's main attractions, the ability to run programs "backwards and forwards".[52]

Prolog is not purely declarative: because of constructs like the cut operator, a procedural reading of a Prolog program is needed to understand it.[53] The order of clauses in a Prolog program is significant, as the execution strategy of the language depends on it.[54] Other logic programming languages, such as Datalog, are truly declarative but restrict the language. As a result, many practical Prolog programs are written to conform to Prolog's depth-first search order, rather than as purely declarative logic programs.[52]

Extensions

Various implementations have been developed from Prolog to extend logic programming capabilities in numerous directions. These include types, modes, constraint logic programming (CLP), object-oriented logic programming (OOLP), concurrency, linear logic (LLP), functional and higher-order logic programming capabilities, plus interoperability with knowledge bases:

Types

Prolog is an untyped language. Attempts to introduce types date back to the 1980s,[55][56] and as of 2008 there are still attempts to extend Prolog with types.[57] Type information is useful not only for type safety but also for reasoning about Prolog programs.[58]

Modes

The syntax of Prolog does not specify which arguments of a predicate are inputs and which are outputs.[59] However, this information is significant and it is recommended that it be included in the comments.[60] Modes provide valuable information when reasoning about Prolog programs[58] and can also be used to accelerate execution.[61]

Mode specifier	Interpretation
`+`	`nonvar` on entry
`-`	`var` on entry
`?`	Not specified

Constraints

Constraint logic programming extends Prolog to include concepts from constraint satisfaction.[62][63] A constraint logic program allows constraints in the body of clauses, such as: `A(X,Y) :- X+Y>0.` It is suited to large-scale combinatorial optimisation problems[64] and is thus useful for applications in industrial settings, such as automated time-tabling and production scheduling. Most Prolog systems ship with at least one constraint solver for finite domains, and often also with solvers for other domains like rational numbers.

Object-orientation

Flora-2 is an object-oriented knowledge representation and reasoning system based on F-logic and incorporates HiLog, Transaction logic, and defeasible reasoning.

Logtalk is an object-oriented logic programming language that can use most Prolog implementations as a back-end compiler. As a multi-paradigm language, it includes support for both prototypes and classes.

Oblog is a small, portable, object-oriented extension to Prolog by Margaret McDougall of EdCAAD, University of Edinburgh.

Objlog was a frame-based language combining objects and Prolog II from CNRS, Marseille, France.

Prolog++ was developed by Logic Programming Associates and first released in 1989 for MS-DOS PCs. Support for other platforms was added, and a second version was released in 1995. A book about Prolog++ by Chris Moss was published by Addison-Wesley in 1994.

Visual Prolog is a multi-paradigm language with interfaces, classes, implementations and object expressions.

Graphics

Prolog systems that provide a graphics library are SWI-Prolog,[65] Visual Prolog, WIN-PROLOG, and B-Prolog.

Concurrency

Prolog-MPI is an open-source SWI-Prolog extension for distributed computing over the Message Passing Interface.[66] Also there are various concurrent Prolog programming languages.[67]

Web programming

Some Prolog implementations, notably Visual Prolog, SWI-Prolog and Ciao, support server-side web programming with support for web protocols, HTML and XML.[68] There are also extensions to support semantic web formats such as RDF and OWL.[69][70] Prolog has also been suggested as a client-side language.[71] In addition Visual Prolog supports JSON-RPC and Websockets.

Adobe Flash

Cedar (https://sites.google.com/site/cedarprolog/) is a free and basic Prolog interpreter. From version 4 and above Cedar has a FCA (Flash Cedar App) support. This provides a new platform to programming in Prolog through ActionScript.

Other

- F-logic extends Prolog with frames/objects for knowledge representation.
- Transaction logic extends Prolog with a logical theory of state-changing update operators. It has both a model-theoretic and procedural semantics.
- OW Prolog has been created in order to answer Prolog's lack of graphics and interface.

Interfaces to other languages

Frameworks exist which can bridge between Prolog and other languages:

- The LPA Intelligence Server (https://www.lpa.co.uk/int.htm) allows the embedding of LPA Prolog for Windows (https://www.lpa.co.uk/win.htm) within C, C#, C++, Java, VB, Delphi, .Net, Lua, Python and other languages. It exploits the dedicated string data-type which LPA Prolog provides
- The Logic Server API allows both the extension and embedding of Prolog in C, C++, Java, VB, Delphi, .NET and any language/environment which can call a .dll or .so. It is implemented for Amzi! Prolog Amzi! Prolog + Logic Server (http://www.amzi.com/) but the API specification can be made available for any implementation.
- JPL (https://jpl7.org/) is a bi-directional Java Prolog bridge which ships with SWI-Prolog by default, allowing Java and Prolog to call each other (recursively). It is known to have good concurrency support and is under active development.
- InterProlog (https://web.archive.org/web/20050406192103/http://www.declarativa.com/InterProlog/), a programming library bridge between Java and Prolog, implementing bi-directional predicate/method calling between both languages. Java objects can be mapped into Prolog terms and vice versa. Allows the development of GUIs and other functionality in Java while leaving logic processing in the Prolog layer. Supports XSB, with support for SWI-Prolog and YAP planned for 2013.
- Prova provides native syntax integration with Java, agent messaging and reaction rules. Prova positions itself as a rule-based scripting (RBS) system for middleware. The language breaks new ground in combining imperative and declarative programming.
- PROL (https://web.archive.org/web/20110221120826/http://www.igormaznitsa.com/projects/prol/index.html) An embeddable Prolog engine for Java. It includes a small IDE and a few libraries.
- GNU Prolog for Java (https://www.gnu.org/software/gnuprologjava/) is an implementation of ISO Prolog as a Java library (gnu.prolog)
- Ciao provides interfaces to C, C++, Java, and relational databases.
- C#-Prolog (http://sourceforge.net/projects/cs-prolog/) is a Prolog interpreter written in (managed) C#. Can easily be integrated in C# programs. Characteristics: reliable and fairly

fast interpreter, command line interface, Windows-interface, builtin DCG, XML-predicates, SQL-predicates, extendible. The complete source code is available, including a parser generator that can be used for adding special purpose extensions.

- Jekejeke Prolog API (http://www.jekejeke.ch/idatab/doclet/prod/en/docs/05_run/10_docu/03_interface/package.html) Archived (https://web.archive.org/web/20191215210030/http://www.jekejeke.ch/idatab/doclet/prod/en/docs/05_run/10_docu/03_interface/package.html) 2019-12-15 at the Wayback Machine provides tightly coupled concurrent call-in and call-out facilities between Prolog and Java or Android, with the marked possibility to create individual knowledge base objects. It can be used to embed the ISO Prolog interpreter in standalones, applets, servlets, APKs, etc..
- A Warren Abstract Machine for PHP (https://github.com/Trismegiste/WamBundle) A Prolog compiler and interpreter in PHP 5.3. A library that can be used standalone or within Symfony2.1 framework which was translated from Stephan Buettcher's (http://stefan.buettcher.org/) work in Java which can be found [here stefan.buettcher.org/cs/wam/index.html (http://stefan.buettcher.org/cs/wam/index.html)]
- tuProlog (https://web.archive.org/web/20190317003033/http://apice.unibo.it/xwiki/bin/view/Tuprolog/WebHome) is a light-weight Prolog system for distributed applications and infrastructures, intentionally designed around a minimal core, to be either statically or dynamically configured by loading/unloading libraries of predicates. tuProlog natively supports multi-paradigm programming, providing a clean, seamless integration model between Prolog and mainstream object-oriented languages—namely Java, for tuProlog Java version, and any .NET-based language (C#, F#..), for tuProlog .NET version.[72]

History

The name *Prolog* was chosen by Philippe Roussel as an abbreviation for *programmation en logique* (French for *programming in logic*). It was created around 1972 by Alain Colmerauer with Philippe Roussel, based on Robert Kowalski's procedural interpretation of Horn clauses. It was motivated in part by the desire to reconcile the use of logic as a declarative knowledge representation language with the procedural representation of knowledge that was popular in North America in the late 1960s and early 1970s. According to Robert Kowalski, the first Prolog system was developed in 1972 by Colmerauer and Phillipe Roussel.[5] The first implementation of Prolog was an interpreter written in Fortran by Gerard Battani and Henri Meloni. David H. D. Warren took this interpreter to Edinburgh, and there implemented an alternative front-end, which came to define the “Edinburgh Prolog” syntax used by most modern implementations. Warren also implemented the first compiler for Prolog, creating the influential DEC-10 Prolog in collaboration with Fernando Pereira. Warren later generalised the ideas behind DEC-10 Prolog, to create the Warren Abstract Machine.

European AI researchers favored Prolog while Americans favored Lisp, reportedly causing many nationalistic debates on the merits of the languages.[73] Much of the modern development of Prolog came from the impetus of the Fifth Generation Computer Systems project (FGCS), which developed a variant of Prolog named *Kernel Language* for its first operating system.

Pure Prolog was originally restricted to the use of a resolution theorem prover with Horn clauses of the form:

```
H :- B1, ..., Bn.
```

The application of the theorem-prover treats such clauses as procedures:

```
to show/solve H, show/solve B1 and ... and Bn.
```

Pure Prolog was soon extended, however, to include negation as failure, in which negative conditions of the form not(B_i) are shown by trying and failing to solve the corresponding positive conditions B_i.

Subsequent extensions of Prolog by the original team introduced constraint logic programming abilities into the implementations.

Use in industry

Prolog has been used in Watson. Watson uses IBM's DeepQA software and the Apache UIMA (Unstructured Information Management Architecture) framework. The system was written in various languages, including Java, C++, and Prolog, and runs on the SUSE Linux Enterprise Server 11 operating system using Apache Hadoop framework to provide distributed computing. Prolog is used for pattern matching over natural language parse trees. The developers have stated: "We required a language in which we could conveniently express pattern matching rules over the parse trees and other annotations (such as named entity recognition results), and a technology that could execute these rules very efficiently. We found that Prolog was the ideal choice for the language due to its simplicity and expressiveness."[14] Prolog is being used in the Low-Code Development Platform GeneXus, which is focused around AI.[74] Open source graph database TerminusDB is implemented in prolog.[75] TerminusDB is designed for collaboratively building and curating knowledge graphs.

See also

- Comparison of Prolog implementations
- Logico-linguistic modeling. A method for building knowledge-based system that uses Prolog.
- Answer set programming. A fully declarative approach to logic programming.
- Association for Logic Programming

Related languages

- The Gödel language is a strongly typed implementation of concurrent constraint logic programming. It is built on SICStus Prolog.
- Visual Prolog, formerly known as PDC Prolog and Turbo Prolog, is a strongly typed object-oriented dialect of Prolog, which is very different from standard Prolog. As Turbo Prolog, it was marketed by Borland, but it is now developed and marketed by the Danish firm PDC (Prolog Development Center) that originally produced it.
- Datalog is a subset of Prolog. It is limited to relationships that may be stratified and does not allow compound terms. In contrast to Prolog, Datalog is not Turing-complete.
- Mercury is an offshoot of Prolog geared toward software engineering in the large with a static, polymorphic type system, as well as a mode and determinism system.
- GraphTalk is a proprietary implementation of Warren's Abstract Machine, with additional object-oriented properties.
- In some ways Prolog is a subset of Planner. The ideas in Planner were later further developed in the Scientific Community Metaphor.
- AgentSpeak is a variant of Prolog for programming agent behavior in multi-agent systems.
- Erlang began life with a Prolog-based implementation and maintains much of Prolog's unification-based syntax.
- Pilog (https://picolisp.com/wiki/?accessToLispFunctionFromPilog) is a declarative language built on top of PicoLisp, that has the semantics of Prolog, but uses the syntax of Lisp.

References

1. Clocksin, William F.; Mellish, Christopher S. (2003). *Programming in Prolog*. Berlin; New York: Springer-Verlag. ISBN 978-3-540-00678-7.
2. Bratko, Ivan (2012). *Prolog programming for artificial intelligence* (4th ed.). Harlow, England; New York: Addison Wesley. ISBN 978-0-321-41746-6.
3. Covington, Michael A. (1994). *Natural language processing for Prolog programmers*. Englewood Cliffs, N.J.: Prentice Hall. ISBN 978-0-13-629213-5.
4. Lloyd, J. W. (1984). *Foundations of logic programming*. Berlin: Springer-Verlag. ISBN 978-3-540-13299-8.
5. Kowalski, R. A. (1988). "The early years of logic programming" (http://www.doc.ic.ac.uk/~rak/papers/the%20early%20years.pdf) (PDF). *Communications of the ACM*. **31**: 38. doi:10.1145/35043.35046 (https://doi.org/10.1145%2F35043.35046). S2CID 12259230 (https://api.semanticscholar.org/CorpusID:12259230).
6. Colmerauer, A.; Roussel, P. (1993). "The birth of Prolog" (http://alain.colmerauer.free.fr/alcol/ArchivesPublications/PrologHistory/19november92.pdf) (PDF). *ACM SIGPLAN Notices*. **28** (3): 37. doi:10.1145/155360.155362 (https://doi.org/10.1145%2F155360.155362).
7. See Logic programming § History.
8. Stickel, M. E. (1988). "A prolog technology theorem prover: Implementation by an extended prolog compiler". *Journal of Automated Reasoning*. **4** (4): 353–380. CiteSeerX 10.1.1.47.3057 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.47.3057). doi:10.1007/BF00297245 (https://doi.org/10.1007%2FBF00297245). S2CID 14621218 (https://api.semanticscholar.org/CorpusID:14621218).
9. Merritt, Dennis (1989). *Building expert systems in Prolog* (https://archive.org/details/buildingexpertsy0000merr). Berlin: Springer-Verlag. ISBN 978-0-387-97016-5.
10. Felty, Amy. "A logic programming approach to implementing higher-order term rewriting." Extensions of Logic Programming (1992): 135-161.
11. Kent D. Lee (19 January 2015). *Foundations of Programming Languages* (https://books.google.com/books?id=dERFBgAAQBAJ&q=prolog+type+inference&pg=PA298). Springer. pp. 298–. ISBN 978-3-319-13314-0.
12. Ute Schmid (21 August 2003). *Inductive Synthesis of Functional Programs: Universal Planning, Folding of Finite Programs, and Schema Abstraction by Analogical Reasoning* (https://books.google.com/books?id=p-Fy25LE4lMC). Springer Science & Business Media. ISBN 978-3-540-40174-2.
13. Fernando C. N. Pereira; Stuart M. Shieber (2005). *Prolog and Natural Language Analysis* (http://mtome.com/Publications/PNLA/pnla.html). Microtome.
14. Adam Lally; Paul Fodor (31 March 2011). "Natural Language Processing With Prolog in the IBM Watson System" (http://www.cs.nmsu.edu/ALP/2011/03/natural-language-processing-with-prolog-in-the-ibm-watson-system/). Association for Logic Programming. See also Watson (computer).
15. ISO/IEC 13211-1:1995 Prolog, 6.3.7 Terms - double quoted list notation. International Organization for Standardization, Geneva.
16. Verify Type of a Term - SWI-Prolog (http://www.swi-prolog.org/pldoc/man?section=typetest)
17. Carlsson, Mats (27 May 2014). *SICStus Prolog User's Manual 4.3: Core reference documentation* (https://books.google.com/books?id=dZimAwAAQBAJ&q=prolog%20failure%20driven%20loop%20%22iteration%22&pg=PA148). BoD – Books on Demand. ISBN 9783735737441 – via Google Books.

18. Covington, Michael A.; Bagnara, Roberto; O'Keefe, Richard A.; Wielemaker, Jan; Price, Simon (2011). "Coding guidelines for Prolog". *Theory and Practice of Logic Programming*. **12** (6): 889–927. arXiv:0911.2899 (https://arxiv.org/abs/0911.2899). doi:10.1017/S1471068411000391 (https://doi.org/10.1017%2FS1471068411000391). S2CID 438363 (https://api.semanticscholar.org/CorpusID:438363).
19. Kirschenbaum, M.; Sterling, L.S. (1993). "Applying Techniques to Skeletons". *Constructing Logic Programs, (Ed. J.M.J. Jacquet)*: 27–140. CiteSeerX 10.1.1.56.7278 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.56.7278).
20. Sterling, Leon (2002). "Patterns for Prolog Programming". *Computational Logic: Logic Programming and Beyond*. Lecture Notes in Computer Science / Lecture Notes in Artificial Intelligence. **2407**. pp. 17–26. doi:10.1007/3-540-45628-7_15 (https://doi.org/10.1007%2F3-540-45628-7_15). ISBN 978-3-540-43959-2.
21. D. Barker-Plummer. Cliche programming in Prolog. In M. Bruynooghe, editor, Proc. Second Workshop on Meta-Programming in Logic, pages 247--256. Dept. of Comp. Sci., Katholieke Univ. Leuven, 1990.
22. Gegg-harrison, T. S. (1995). *Representing Logic Program Schemata in Prolog*. Procs Twelfth International Conference on Logic Programming. pp. 467–481.
23. Deville, Yves (1990). *Logic programming: systematic program development*. Wokingham, England: Addison-Wesley. ISBN 978-0-201-17576-9.
24. Naish, Lee (1996). Higher-order logic programming in Prolog (Report). Department of Computer Science, University of Melbourne. CiteSeerX 10.1.1.35.4505 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.35.4505).
25. "With regard to Prolog variables, variables only in the head are implicitly universally quantified, and those only in the body are implicitly existentially quantified" (http://okmij.org/ftp/Prolog/quantification.txt). Retrieved 2013-05-04.
26. ISO/IEC 13211: Information technology — Programming languages — Prolog. International Organization for Standardization, Geneva.
27. ISO/IEC 13211-2: Modules.
28. Moura, Paulo (August 2004), "Logtalk" (http://www.cs.kuleuven.ac.be/~dtai/projects/ALP/newsletter/aug04/nav/print/all.html#logtalk), *Association of Logic Programming*, **17** (3)
29. Shapiro, Ehud Y.; Sterling, Leon (1994). *The Art of Prolog: Advanced Programming Techniques*. Cambridge, Massachusetts: MIT Press. ISBN 978-0-262-19338-2.
30. A. Ed-Dbali; Deransart, Pierre; L. Cervoni (1996). *Prolog: the standard: reference manual*. Berlin: Springer. ISBN 978-3-540-59304-1.
31. "ISO/IEC 13211-1:1995/Cor 1:2007 -" (http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50405).
32. "ISO/IEC 13211-1:1995/Cor 2:2012 -" (http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=58033).
33. "ISO/IEC 13211-1:1995/Cor 3:2017 -" (https://www.iso.org/standard/73194.html).
34. "ISO/IEC JTC1 SC22 WG17" (http://iso.org/jtc1sc22wg17).
35. "X3J17 and the Prolog Standard" (https://web.archive.org/web/20090823160951/http://www.sju.edu/~jhodgson/x3j17.html). Archived from the original (http://www.sju.edu/~jhodgson/x3j17.html) on 2009-08-23. Retrieved 2009-10-02.
36. David H. D. Warren. "An abstract Prolog instruction set" (http://www.ai.sri.com/pubs/files/641.pdf). Technical Note 309, SRI International, Menlo Park, CA, October 1983.
37. Van Roy, P.; Despain, A. M. (1992). "High-performance logic programming with the Aquarius Prolog compiler". *Computer*. **25**: 54–68. doi:10.1109/2.108055 (https://doi.org/10.1109%2F2.108055). S2CID 16447071 (https://api.semanticscholar.org/CorpusID:16447071).
38. Graf, Peter (1995). *Term indexing*. Springer. ISBN 978-3-540-61040-3.

39. Wise, Michael J.; Powers, David M. W. (1986). *Indexing Prolog Clauses via Superimposed Code Words and Field Encoded Words*. *International Symposium on Logic Programming*. pp. 203–210.
40. Colomb, Robert M. (1991). "Enhancing unification in PROLOG through clause indexing" (https://doi.org/10.1016%2F0743-1066%2891%2990004-9). *The Journal of Logic Programming*. **10**: 23–44. doi:10.1016/0743-1066(91)90004-9 (https://doi.org/10.1016%2F0743-1066%2891%2990004-9).
41. Swift, T. (1999). "Tabling for non-monotonic programming". *Annals of Mathematics and Artificial Intelligence*. **25** (3/4): 201–240. doi:10.1023/A:1018990308362 (https://doi.org/10.1023%2FA%3A1018990308362). S2CID 16695800 (https://api.semanticscholar.org/CorpusID:16695800).
42. Zhou, Neng-Fa; Sato, Taisuke (2003). "Efficient Fixpoint Computation in Linear Tabling" (http://www.sci.brooklyn.cuny.edu/~zhou/papers/ppdp03.pdf) (PDF). *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*: 275–283.
43. Swift, T.; Warren, D. S. (2011). "XSB: Extending Prolog with Tabled Logic Programming". *Theory and Practice of Logic Programming*. **12** (1–2): 157–187. arXiv:1012.5123 (https://arxiv.org/abs/1012.5123). doi:10.1017/S1471068411000500 (https://doi.org/10.1017%2FS1471068411000500). S2CID 6153112 (https://api.semanticscholar.org/CorpusID:6153112).
44. Abe, S.; Bandoh, T.; Yamaguchi, S.; Kurosawa, K.; Kiriyama, K. (1987). "High performance integrated Prolog processor IPP". *Proceedings of the 14th annual international symposium on Computer architecture - ISCA '87*. p. 100. doi:10.1145/30350.30362 (https://doi.org/10.1145%2F30350.30362). ISBN 978-0818607769. S2CID 10283148 (https://api.semanticscholar.org/CorpusID:10283148).
45. Robinson, Ian (1986). *A Prolog processor based on a pattern matching memory device*. Third International Conference on Logic Programming. Lecture Notes in Computer Science. **225**. Springer. pp. 172–179. doi:10.1007/3-540-16492-8_73 (https://doi.org/10.1007%2F3-540-16492-8_73). ISBN 978-3-540-16492-0.
46. Taki, K.; Nakajima, K.; Nakashima, H.; Ikeda, M. (1987). "Performance and architectural evaluation of the PSI machine". *ACM SIGPLAN Notices*. **22** (10): 128. doi:10.1145/36205.36195 (https://doi.org/10.1145%2F36205.36195).
47. Gupta, G.; Pontelli, E.; Ali, K. A. M.; Carlsson, M.; Hermenegildo, M. V. (2001). "Parallel execution of prolog programs: a survey" (http://oa.upm.es/11160/). *ACM Transactions on Programming Languages and Systems*. **23** (4): 472. doi:10.1145/504083.504085 (https://doi.org/10.1145%2F504083.504085). S2CID 2978041 (https://api.semanticscholar.org/CorpusID:2978041).
48. "Statically Allocated Systems" (http://www.cl.cam.ac.uk/~am21/research/sa/byrdbox.ps.gz).
49. Logic programming for the real world. Zoltan Somogyi, Fergus Henderson, Thomas Conway, Richard O'Keefe. Proceedings of the ILPS'95 Postconference Workshop on Visions for the Future of Logic Programming.
50. "FAQ: Prolog Resource Guide 1/2 [Monthly posting] Section - [1-8] The Prolog 1000 Database" (http://www.faqs.org/faqs/prolog/resource-guide/part1/section-9.html).
51. Jan Wielemaker and Vıtor Santos Costa: Portability of Prolog programs: theory and case-studies (http://www.swi-prolog.org/download/publications/porting.pdf). CICLOPS-WLPE Workshop 2010 (http://www.floc-conference.org/CICLOPS-WLPE-accepted.html).
52. Kiselyov, Oleg; Kameyama, Yukiyoshi (2014). *Re-thinking Prolog* (http://okmij.org/ftp/kakuritu/logic-programming.html#vs-prolog). Proc. 31st meeting of the Japan Society for Software Science and Technology.
53. Franzen, Torkel (1994), "Declarative vs procedural" (http://dtai.cs.kuleuven.be/projects/ALP/newsletter/archive_93_96/comment/decl.html), *Association of Logic Programming*, **7** (3)

54. Dantsin, Evgeny; Eiter, Thomas; Gottlob, Georg; Voronkov, Andrei (2001). "Complexity and Expressive Power of Logic Programming". *ACM Computing Surveys*. **33** (3): 374–425. CiteSeerX 10.1.1.616.6372 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.616.6372). doi:10.1145/502807.502810 (https://doi.org/10.1145%2F502807.502810). S2CID 518049 (https://api.semanticscholar.org/CorpusID:518049).
55. Mycroft, A.; O'Keefe, R. A. (1984). "A polymorphic type system for prolog". *Artificial Intelligence*. **23** (3): 295. doi:10.1016/0004-3702(84)90017-1 (https://doi.org/10.1016%2F0004-3702%2884%2990017-1).
56. Pfenning, Frank (1992). *Types in logic programming*. Cambridge, Massachusetts: MIT Press. ISBN 978-0-262-16131-2.
57. Schrijvers, Tom; Santos Costa, Vitor; Wielemaker, Jan; Demoen, Bart (2008). "Towards Typed Prolog". In Maria Garcia de la Banda; Enrico Pontelli (eds.). *Logic programming : 24th international conference, ICLP 2008, Udine, Italy, December 9-13, 2008 : proceedings*. Lecture Notes in Computer Science. **5366**. pp. 693–697. doi:10.1007/978-3-540-89982-2_59 (https://doi.org/10.1007%2F978-3-540-89982-2_59). ISBN 9783540899822.
58. Apt, K. R.; Marchiori, E. (1994). "Reasoning about Prolog programs: From modes through types to assertions" (http://www.cwi.nl/~apt/am.ps). *Formal Aspects of Computing*. **6** (S1): 743. CiteSeerX 10.1.1.57.395 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.57.395). doi:10.1007/BF01213601 (https://doi.org/10.1007%2FBF01213601). S2CID 12235465 (https://api.semanticscholar.org/CorpusID:12235465).
59. O'Keefe, Richard A. (1990). *The craft of Prolog*. Cambridge, Massachusetts: MIT Press. ISBN 978-0-262-15039-2.
60. Michael Covington; Roberto Bagnara; et al. (2010). "Coding guidelines for Prolog". arXiv:0911.2899 (https://arxiv.org/abs/0911.2899) [cs.PL (https://arxiv.org/archive/cs.PL)].
61. Roy, P.; Demoen, B.; Willems, Y. D. (1987). "Improving the execution speed of compiled Prolog with modes, clause selection, and determinism" (https://archive.org/details/tapsoft87proceed0000inte/page/111). *Tapsoft '87*. Lecture Notes in Computer Science. **250**. pp. 111 (https://archive.org/details/tapsoft87proceed0000inte/page/111). doi:10.1007/BFb0014976 (https://doi.org/10.1007%2FBFb0014976). ISBN 978-3-540-17611-4.
62. Jaffar, J. (1994). "Constraint logic programming: a survey" (https://doi.org/10.1016%2F0743-1066%2894%2990033-7). *The Journal of Logic Programming*. 19–20: 503–581. doi:10.1016/0743-1066(94)90033-7 (https://doi.org/10.1016%2F0743-1066%2894%2990033-7).
63. Colmerauer, Alain (1987). "Opening the Prolog III Universe". *Byte*. August.
64. Wallace, M. (2002). "Constraint Logic Programming". *Computational Logic: Logic Programming and Beyond*. Lecture Notes in Computer Science. **2407**. pp. 512–556. doi:10.1007/3-540-45628-7_19 (https://doi.org/10.1007%2F3-540-45628-7_19). ISBN 978-3540456285.
65. "XPCE graphics library" (http://www.swi-prolog.org/packages/xpce/).
66. "prolog-mpi" (http://apps.lumii.lv/prolog-mpi/). Apps.lumii.lv. Retrieved 2010-09-16.
67. Ehud Shapiro. *The family of concurrent logic programming languages* ACM Computing Surveys. September 1989.
68. Wielemaker, J.; Huang, Z.; Van Der Meij, L. (2008). "SWI-Prolog and the web" (https://pure.uva.nl/ws/files/4221287/58604_285112.pdf) (PDF). *Theory and Practice of Logic Programming*. **8** (3): 363. doi:10.1017/S1471068407003237 (https://doi.org/10.1017%2FS1471068407003237). S2CID 5404048 (https://api.semanticscholar.org/CorpusID:5404048).
69. Jan Wielemaker and Michiel Hildebrand and Jacco van Ossenbruggen (2007), S. Heymans; A. Polleres; E. Ruckhaus; D. Pearse; G. Gupta (eds.), "Using {Prolog} as the fundament for applications on the semantic web" (http://ftp.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-287/paper_1.pdf) (PDF), *Proceedings of the 2nd Workshop on Applications of Logic Programming and to the Web, Semantic Web and Semantic Web Services*, CEUR Workshop Proceedings, Porto, Portugal: CEUR-WS.org, **287**, pp. 84–98

70. Processing OWL2 Ontologies using Thea: An Application of Logic Programming (http://ceur-ws.org/Vol-529/owled2009_submission_43.pdf). Vangelis Vassiliadis, Jan Wielemaker and Chris Mungall. Proceedings of the 5th International Workshop on OWL: Experiences and Directions (OWLED 2009), Chantilly, VA, United States, October 23–24, 2009
71. Loke, S. W.; Davison, A. (2001). "Secure Prolog-based mobile code". *Theory and Practice of Logic Programming*. **1** (3): 321. arXiv:cs/0406012 (https://arxiv.org/abs/cs/0406012). CiteSeerX 10.1.1.58.6610 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.58.6610). doi:10.1017/S1471068401001211 (https://doi.org/10.1017%2FS1471068401001211). S2CID 11754347 (https://api.semanticscholar.org/CorpusID:11754347).
72. "Archived copy" (https://web.archive.org/web/20190317003033/http://apice.unibo.it/xwiki/bin/view/Tuprolog/WebHome). Archived from the original (http://apice.unibo.it/xwiki/bin/view/Tuprolog/WebHome) on 2019-03-17. Retrieved 2019-06-08.
73. Pountain, Dick (October 1984). "POP and SNAP" (https://archive.org/stream/byte-magazine-1984-10/1984_10_BYTE_09-11_Databases#page/n377/mode/2up). *BYTE*. p. 381. Retrieved 23 October 2013.
74. "Wikipedia GeneXus Page" (https://en.wikipedia.org/wiki/GeneXus).
75. *terminusdb/terminusdb* (https://github.com/terminusdb/terminusdb), TerminusDB, 2020-12-13, retrieved 2020-12-15

Further reading

- Blackburn, Patrick; Bos, Johan; Striegnitz, Kristina (2006). *Learn Prolog Now!* (http://www.learnprolognow.org/). ISBN 978-1-904987-17-8.
- Ivan Bratko, *Prolog Programming for Artificial Intelligence*, 4th ed., 2012, ISBN 978-0-321-41746-6. Book supplements and source code (https://ailab.si/ivan/datoteke/dokumenti/32/107_19_Bratko_Prolog_book_web_resources.htm)
- William F. Clocksin, Christopher S. Mellish: *Programming in Prolog: Using the ISO Standard*. Springer, 5th ed., 2003, ISBN 978-3-540-00678-7. *(This edition is updated for ISO Prolog. Previous editions described Edinburgh Prolog.)*
- William F. Clocksin: *Clause and Effect. Prolog Programming for the Working Programmer*. Springer, 2003, ISBN 978-3-540-62971-9.
- Michael A. Covington, Donald Nute, Andre Vellino, *Prolog Programming in Depth*, 1996, ISBN 0-13-138645-X.
- Michael A. Covington, *Natural Language Processing for Prolog Programmers*, 1994, ISBN 978-0-13-629213-5
- M. S. Dawe and C.M.Dawe, *Prolog for Computer Sciences*, Springer Verlag 1992.
- *ISO/IEC 13211: Information technology — Programming languages — Prolog*. International Organization for Standardization, Geneva.
- Feliks Kluźniak and Stanisław Szpakowicz (with a contribution by Janusz S. Bień). *Prolog for Programmers*. Academic Press Inc. (London), 1985, 1987 (available under a Creative Commons license at sites.google.com/site/prologforprogrammers/ (https://sites.google.com/site/prologforprogrammers/)). ISBN 0-12-416521-4.
- Richard O'Keefe, *The Craft of Prolog*, ISBN 0-262-15039-5.
- Robert Smith, John Gibson, Aaron Sloman: 'POPLOG's two-level virtual machine support for interactive languages', in *Research Directions in Cognitive Science Volume 5: Artificial Intelligence*, Eds D. Sleeman and N. Bernsen, Lawrence Erlbaum Associates, pp 203–231, 1992.
- Leon Sterling and Ehud Shapiro, *The Art of Prolog: Advanced Programming Techniques*, 1994, ISBN 0-262-19338-8.

- David H D Warren, Luis M. Pereira and Fernando Pereira, Prolog - the language and its implementation compared with Lisp. ACM SIGART Bulletin archive, Issue 64. Proceedings of the 1977 symposium on Artificial intelligence and programming languages, pp 109–115.
