Logic programming is, in its broadest sense, the use of mathematical logic for computer programming. In this view of logic programming, which can be traced at least as far back as John McCarthy's [1958] advice-taker proposal, logic is used as a purely declarative representation language, and a theorem-prover or model-generator is used as the problem-solver. The problem-solving task is split between the programmer, who is responsible only for ensuring the truth of programs expressed in logical form, and the theorem-prover or model-generator, which is responsible for solving problems efficiently.

However, logic programming, in the narrower sense in which it is more commonly understood, is the use of logic as both a declarative and procedural representation language. It is based upon the fact that a backwards reasoning theorem-prover applied to declarative sentences in the form of implications:

\[ \text{If } B_1 \text{ and } \ldots \text{ and } B_n \text{ then } H \]

treats the implications as goal-reduction procedures:

\[ \text{to show/solve } H, \text{ show/solve } B_1 \text{ and } \ldots \text{ and } B_n. \]

For example, it treats the implication:

If you press the alarm signal button,

then you alert the driver of the train of a possible emergency

as the procedure:

To alert the driver of the train of a possible emergency,

press the alarm signal button.

Note that this is consistent with the BHK interpretation of constructive logic, where implication would be interpreted as a solution of problem \( H \) given solutions of \( B_1 \ldots B_n \). However, the defining feature of logic programming is that sets of formulas can be regarded as programs and proof search can be given a computational meaning. This is achieved by restricting the underlying logic to a "well-behaved" fragment such as Horn clauses or Hereditary Harrop formulas. See D. Miller et al., 1991.

As in the purely declarative case, the programmer is responsible for ensuring the truth of programs. But since automated proof search is generally infeasible, logic programming as commonly understood also relies on the programmer to ensure that inferences are generated efficiently (see #Problem solving). In many cases, to achieve efficiency, one needs to be aware of and to exploit the problem-solving behavior of the theorem-prover. In this respect, logic programming is comparable to conventional imperative programming; using programs to control the behaviour of a program executor. However, unlike conventional imperative programs, which have only a procedural interpretation, logic programs also have a declarative, logical interpretation, which helps to ensure their correctness. Moreover, such programs, being declarative, are at a higher conceptual level than purely imperative programs; and their program executors, being theorem-provers, operate at a higher conceptual level than conventional compilers and interpreters.
**History**

Logic programming in the first and wider sense gave rise to a number of implementations, such as those by Fischer Black (1964), James Slagle (1965) and Cordell Green (1969), which were question-answering systems in the spirit of McCarthy's advice-taker. Foster and Elcock's Absys (1969), on the other hand, was probably the first language to be explicitly developed as an assertional programming language.

Logic programming in the narrower sense can be traced back to debates in the late 1960s and early 1970s about declarative versus procedural representations of knowledge in Artificial Intelligence. Advocates of declarative representations were notably working at Stanford, associated with John McCarthy, Bertram Raphael and Cordell Green, and in Edinburgh, with J. Alan Robinson (an academic visitor from Syracuse University), Pat Hayes, and Bob Kowalski. Advocates of procedural representations were mainly centered at MIT, under the leadership of Marvin Minsky and Seymour Papert.

Although it was based on logic, Planner, developed at MIT, was the first language to emerge within this proceduralist paradigm [Hewitt, 1969]. Planner featured pattern directed invocation of procedural plans from goals (i.e. forward chaining) and from assertions (i.e. backward chaining). The most influential implementation of Planner was the subset of Planner, called Micro-Planner, implemented by Gerry Sussman, Eugene Charniak and Terry Winograd. It was used to implement Winograd's natural-language understanding program SHRDLU, which was a landmark at that time. In order to cope with the very limited memory systems that were available when it was developed, Planner used backtracking control structure so that only one possible computation path had to be stored at a time. From Planner there developed the programming languages QA-4, Popler, Conniver, QLISP, and the concurrent language Ether.

Hayes and Kowalski in Edinburgh tried to reconcile the logic-based declarative approach to knowledge representation with Planner's procedural approach. Hayes (1973) developed an equational language, Golux, in which different procedures could be obtained by altering the behavior of the theorem prover. Kowalski, on the other hand, showed how SL-resolution treats implications as goal-reduction procedures. Kowalski collaborated with Colmerauer in Marseille, who developed these ideas in the design and implementation of the programming language Prolog. From Prolog there developed, among others, the programming languages ALF, Fril, Gödel, Mercury, Oz, Ciao, Visual Prolog, XSB, and λProlog, as well as a variety of concurrent logic programming languages, (see Shapiro (1989) for a survey), constraint logic programming languages and datalog.

In 1997, the Association of Logic Programming bestowed to fifteen recognized researchers in logic programming the title *Founders of Logic Programming* to recognize them as pioneers in the field. The individuals receiving this honor were: Maurice Bruynooghe (Belgium), Jacques Cohen (USA), Alain Colmerauer (France), Keith Clark (UK), Veronica Dahl (Canada/Argentina), Maarten van Emden (Canada), Herve Gallaire (France), Robert Kowalski (UK), Jack Minker (USA), Fernando Pereira (USA), Luis Moniz Pereira (Portugal), Ray Reiter (Canada), Alan Robinson (USA), Peter Szeredi (Hungary), and David H.D. Warren (UK).

**Prolog**

The programming language Prolog was developed in 1972 by Alain Colmerauer. It emerged from a collaboration between Colmerauer in Marseille and Robert Kowalski in Edinburgh. Colmerauer was working on natural language understanding, using logic to represent semantics and using resolution for question-answering. During the summer of 1971, Colmerauer and Kowalski discovered that the clausal form of logic could be used to represent formal grammars and that resolution theorem provers could be used for parsing. They observed that some theorem provers, like hyper-resolution, behave as bottom-up parsers and others, like SL-resolution (1971), behave as top-down parsers.

It was in the following summer of 1972, that Kowalski, again working with Colmerauer, developed the procedural interpretation of implications. This dual declarative/procedural interpretation later became formalised in the Prolog
which can be read (and used) both declaratively and procedurally. It also became clear that such clauses could be restricted to definite clauses or Horn clauses, where \( H, B_1, \ldots, B_n \) are all atomic predicate logic formulae, and that SL-resolution could be restricted (and generalised) to LUSH or SLD-resolution. Kowalski's procedural interpretation and LUSH were described in a 1973 memo, published in 1974.

Colmerauer, with Philippe Roussel, used this dual interpretation of clauses as the basis of Prolog, which was implemented in the summer and autumn of 1972. The first Prolog program, also written in 1972 and implemented in Marseille, was a French question-answering system. The use of Prolog as a practical programming language was given great momentum by the development of a compiler by David Warren in Edinburgh in 1977. Experiments demonstrated that Edinburgh Prolog could compete with the processing speed of other symbolic programming languages such as Lisp. Edinburgh Prolog became the de facto standard and strongly influenced the definition of ISO standard Prolog.

**Negation as failure**

Micro-Planner had a construct, called "thnot", which when applied to an expression returns the value true if (and only if) the evaluation of the expression fails. An equivalent operator is normally built-in in modern Prolog's implementations and has been called "negation as failure". It is normally written as not(p), where p is an atom whose variables normally have been instantiated by the time not(p) is invoked. A more cryptic (but standard) syntax is \( \lnot p \). Negation as failure literals can occur as conditions not\( (B_i) \) in the body of program clauses.

The logical status of negation as failure was unresolved until Keith Clark [1978] showed that, under certain natural conditions, it is a correct (and sometimes complete) implementation of classical negation with respect to the completion of the program. Completion amounts roughly to regarding the set of all the program clauses with the same predicate on the left hand side, say

\[
H ::= \text{Body}_1, \ldots, \text{Body}_k.
\]

as a definition of the predicate

\[
H \text{ iff } (\text{Body}_1 \text{ or } \ldots \text{ or } \text{Body}_k)
\]

where "iff" means "if and only if". Writing the completion also requires explicit use of the equality predicate and the inclusion of a set of appropriate axioms for equality. However, the implementation of negation by failure needs only the if-halves of the definitions without the axioms of equality.

The notion of completion is closely related to McCarthy's circumscription semantics for default reasoning, and to the closed world assumption.

As an alternative to the completion semantics, negation as failure can also be interpreted epistemically, as in the stable model semantics of answer set programming. In this interpretation not\( (B_i) \) means literally that \( B_i \) is not known or not believed. The epistemic interpretation has the advantage that it can be combined very simply with classical negation, as in "extended logic programming", to formalise such phrases as "the contrary can not be shown", where "contrary" is classical negation and "can not be shown" is the epistemic interpretation of negation as failure.
Problem solving

In the simplified, propositional case in which a logic program and a top-level atomic goal contain no variables, backward reasoning determines an and-or tree, which constitutes the search space for solving the goal. The top-level goal is the root of the tree. Given any node in the tree and any clause whose head matches the node, there exists a set of child nodes corresponding to the sub-goals in the body of the clause. These child nodes are grouped together by an "and". The alternative sets of children corresponding to alternative ways of solving the node are grouped together by an "or".

Any search strategy can be used to search this space. Prolog uses a sequential, last-in-first-out, backtracking strategy, in which only one alternative and one sub-goal is considered at a time. Other search strategies, such as parallel search, intelligent backtracking, or best-first search to find an optimal solution, are also possible.

In the more general case, where sub-goals share variables, other strategies can be used, such as choosing the subgoal that is most highly instantiated or that is sufficiently instantiated so that only one procedure applies. Such strategies are used, for example, in concurrent logic programming.

The fact that there are alternative ways of executing a logic program has been characterised by the equation:

Algorithm = Logic + Control

where "Logic" represents a logic program and "Control" represents different theorem-proving strategies.¹

Knowledge representation

The fact that Horn clauses can be given a procedural interpretation and, vice versa, that goal-reduction procedures can be understood as Horn clauses + backward reasoning means that logic programs combine declarative and procedural representations of knowledge. The inclusion of negation as failure means that logic programming is a kind of non-monotonic logic.

Despite its simplicity compared with classical logic, this combination of Horn clauses and negation as failure has proved to be surprisingly expressive. For example, it has been shown to correspond, with some further extensions, quite naturally to the semi-formal language of legislation. It is also a natural language for expressing common-sense laws of cause and effect, as in the situation calculus and event calculus.

Abductive logic programming

Abductive Logic Programming is an extension of normal Logic Programming that allows some predicates, declared as abducible predicates, to be incompletely defined. Problem solving is achieved by deriving hypotheses expressed in terms of the abducible predicates as solutions of problems to be solved. These problems can be either observations that need to be explained (as in classical abductive reasoning) or goals to be achieved (as in normal logic programming). It has been used to solve problems in Diagnosis, Planning, Natural Language and Machine Learning. It has also been used to interpret Negation as Failure as a form of abductive reasoning.

Metalogic programming

Because mathematical logic has a long tradition of distinguishing between object language and metalanguage, logic programming also allows metalevel programming. The simplest metalogic program is the so-called "vanilla" meta-interpreter:

```
solve(true).
solve((A,B)):- solve(A),solve(B).
solve(A):- clause(A,B),solve(B).
```

Source URL: http://en.wikipedia.org/wiki/Logic_programming
Saylor URL: http://www.saylor.org/courses/cs408

Attributed to [Wikipedia]
where true represents an empty conjunction, and clause(A,B) means there is an object-level clause of the form A :- B.

Metalogic programming allows object-level and metalevel representations to be combined, as in natural language. It can also be used to implement any logic that is specified by means of inference rules.

**Constraint logic programming**

Constraint logic programming is an extension of normal Logic Programming that allows some predicates, declared as constraint predicates, to occur as literals in the body of clauses. These literals are not solved by goal-reduction using program clauses, but are added to a store of constraints, which is required to be consistent with some built-in semantics of the constraint predicates.

Problem solving is achieved by reducing the initial problem to a satisfiable set of constraints. Constraint logic programming has been used to solve problems in such fields as civil engineering, mechanical engineering, digital circuit verification, automated timetabling, air traffic control, and finance. It is closely related to abductive logic programming.

**Concurrent logic programming**

Keith Clark, Steve Gregory, Vijay Saraswat, Udi Shapiro, Kazunori Ueda, etc. developed a family of Prolog-like concurrent message passing systems using unification of shared variables and data structure streams for messages. Efforts were made to base these systems on mathematical logic, and they were used as the basis of the Japanese Fifth Generation Project (ICOT). However, the Prolog-like concurrent systems were based on message passing and consequently were subject to the same indeterminacy as other concurrent message-passing systems, such as Actors (see Indeterminancy in concurrent computation). Consequently, the ICOT languages were not based on logic in the sense that computational steps could not be logically deduced [Hewitt and Agha, 1988].

Concurrent constraint logic programming combines concurrent logic programming and constraint logic programming, using constraints to control concurrency. A clause can contain a guard, which is a set of constraints that may block the applicability of the clause. When the guards of several clauses are satisfied, concurrent constraint logic programming makes a committed choice to the use of only one.

**Inductive logic programming**

Inductive logic programming is concerned with generalizing positive and negative examples in the context of background knowledge. Generalizations, as well as the examples and background knowledge, are expressed in logic programming syntax. Recent work in this area, combining logic programming, learning and probability, has given rise to the new field of statistical relational learning and probabilistic inductive logic programming.

**Higher-order logic programming**

Several researchers have extended logic programming with higher-order programming features derived from higher-order logic, such as predicate variables. Such languages include the Prolog extensions HiLog and λProlog.

**Linear logic programming**

Basing logic programming within linear logic has resulted in the design of logic programming languages that are considerably more expressive than those based on classical logic. Horn clause programs can only represent state change by the change in arguments to predicates. In linear logic programming, one can use the ambient linear logic to support state change. Some early designs of logic programming languages based on linear logic include LO [Andreoli & Pareschi, 1991], Lolli [Hodas & Miller, 1994], ACL [Kobayashi & Yonezawa, 1994], and Forum.
[Miller, 1996]. Forum provides a goal-direct interpretation of all of linear logic.

References


General introductions


Other sources

Further reading

- Ulf Nilsson and Jan Maluszynski, Logic, Programming and Prolog (http://www.ida.liu.se/~ulfni/lpp/)

External links

- *Logic Programming* Virtual Library entry (http://vl.fennet.info/logic-prog/)
- Bibliographies on Logic Programming (http://linwww.ira.uka.de/bibliography/LogicProgramming/)
- Association for Logic Programming (ALP) (http://www.cs.kuleuven.be/~dtai/projects/ALP/)
- Logic programming in C++ with Castor (http://www.mpprogramming.com/Cpp/)
- Logic programming in (http://www.mozart-oz.org/documentation/tutorial/node12.html) Oz
- Prolog Development Center (http://www.pdc.dk/)
Article Sources and Contributors


License

Creative Commons Attribution-Share Alike 3.0 Unported
http://creativecommons.org/licenses/by-sa/3.0/

Attributed to [Wikipedia]  

Source URL: http://en.wikipedia.org/wiki/Logic_programming  
Saylor URL: http://www.saylor.org/courses/cs408