Prolog

Seminar on the History of Programming Languages

Ahmad Taherkhani
December 8, 2009
Presentation Outline

- Prolog
  - Fundamentals
  - Features
  - Syntax and semantics, examples

- History
  - The very earlier history
  - Design, implementation and evolution
Prolog: **Programming in Logic**

A declarative/descriptive language

Based on first-order logic

Comprised of two fundamental elements: the database (i.e., knowledge-base) and the interpreter

- The database contains the **facts** and **rules** used to describe the problem
- The interpreter provides the control in the form of a deduction method
The execution of Prolog is a natural generalization of the execution of imperative languages

It can be summarized as:
Prolog = imperative language + unification + backtracking

As in imperative languages, control flow is left to right within a **clause**. When a goal is called, the clauses in the **predicate’s** definition are chosen in textual order from top to bottom
Clause

- A clause represents a statement about objects and relations between objects. Clauses represent the knowledge which the Prolog system can draw upon when processing a query.

- A unit clause (a clause without body) such as $e$ means $e = \text{true}$. For example, male(john) means john is a male person. This is called a fact.
Predicate

- Clauses with the same clause name, the same number of arguments and defined in the same module form the definition of a predicate. The common clause name is called the predicate name or functor of the predicate. The number of arguments is called the arity.

- For example, the predicate `fac/2` is defined by the collection of all clauses with the clause head `fac(Arg1,Arg2)`, where `Arg1` and `Arg2` may be any terms.

- The two clauses below define the predicate `fac/2` for calculating the factorial of a natural number:
  - `fac(0, 1) :- !. % 0! is 1`
  - `fac(N, Nfac) :-
      N > 0, % for integers N > 0
      M is N - 1, % defines NFac as N * (N-1)!
      fac(M, Mfac), % thus MFac is calculated
      Nfac is N * Mfac. % recursively as (N-1)!

- `fac/2` can be called by entering, for example
  `- fac(5,X).`

- The Prolog system searches the clauses belonging to the predicate `fac/2` and displays the answer: `X = 120`
Term

- All Prolog data structures are called terms. A term can be:
  - A **constant**, which can be either an **atom** or a **number**
  - A **variable**
  - A **structure** (i.e., compound term)

- An atom means a single data item. It may be of one of three types:
  - A string atom, like 'This is a string' or
  - A symbol, like *likes*, *john*, and *mary*, in `likes(john, mary)`. Atoms of this type must start with a lower case letter. They can include digits (after the initial lower-case letter) and the underscore character (\_).
  - Strings of special characters, like `<---`, `...`, `===>`
Unification

Unification is an operation which checks whether two terms are equal or can be made equal by suitably instantiating their uninstantiated variables.

Unification is governed by the following rules:

- An (uninstantiated) variable can be unified with any term by being instantiated to this term.
- Constants are unifiable if they are identical.
- Structures are unifiable if they have the same functor and the same arity and if corresponding arguments are also unifiable.

When a variable is instantiated to a term by unification, the instantiation applies to the particular clause involved. It is not possible to perform a new instantiation for this variable within the clause, and any subsequent unification with a different term will fail. Variable instantiation can only be undone in the course of backtracking.
The mechanism for trying to satisfy goals or finding multiple solutions is called backtracking.

For example, if the database is:
- `eats(john, orange).
eats(john, banana).
eats(john, apple).`

- Question: What are all the things that John eats?
- Query: `?- eats(john, What).`?
- Answer: What = orange
- Other possible solutions: What = banana
- Other possible solutions: What = apple
- Other possible solutions: no
The Correspondence Between Logical and Imperative Concepts

Prolog  Imperative language

set of clauses ----- program

predicate; set of clauses ----- procedure definition;
with same name and arity     nondeterministic case statement

clause; axiom ----- one branch of a nondeterministic case statement;
if statement; series of procedure calls

goal invocation ----- procedure call

unification ----- parameter passing; assignment;
dynamic memory allocation

backtracking ----- conditional branching; iteration;
continuation passing

logical variable ----- pointer manipulation

recursion ----- iteration
A Prolog program consists of a set of clauses, where each clause is either:

- A fact about the given information, or
- A rule about how the solution may relate to or be inferred from the given facts

The Prolog program is viewed as the conjunction of formulas it defines. It is storehouse of facts and rules, which are used to answer questions

The query succeeds if it and the program are simultaneously satisfiable
Programming in Prolog

- Describing known facts about objects
- Defining rules about objects and their relationship
- Asking (the right kind of) questions about objects and their relationship. Prolog deduce new facts from the database constructed by a programmer
Computation in Prolog

- Computation is reduced to deduction
- Declarative statement like "P if Q and R and S" can be interpreted procedurally as "To solve P, solve Q and R and S"
- Computation is carried out not by specifying an algorithm and prescribing steps (i.e., control flow) by a programmer, but rather
  - Partly by logical declarative semantics of Prolog
  - Partly by what new facts Prolog can infer from the given ones
  - And (only) partly by explicit control information supplied by the programmer (e.g., cut)
Backward Chaining

- In Prolog, inference is done by backward chaining (backward reasoning)
- P implies Q interprets to
  - When goal Q, goal P
  - (And when goal not P, goal not Q)
  - In other words, to show that \( p(X) \) holds, find rules of the form \( p(X) :- q(X) \) and show that \( q(X) \) holds
- In contrast to forward chaining:
  - When assert P, assert Q
  - (And when assert not Q, assert not P)
Facts

- A fact is a Prolog clause without a clause body.
- A fact represents unconditional knowledge, i.e. it describes a relation between objects which always holds true, regardless of any other relations.
- For example, John likes Mary: "likes(john, mary)." is a fact.
  - john and mary are objects.
  - "likes" is a relationship.
- Objects and relationship begin with a lower-case letter.
- The full stop character (.) must always appear at the end of a fact.
- The names of the object within round brackets are called arguments.
- If two facts have the same predicates and their corresponding arguments each are the same, they are said to match.
Questions

- If Prolog finds a fact from its database that matches the fact in the question, it outputs yes:
  
  ?- likes(john, mary).

  yes

- Otherwise no:

  ?- likes(mary, john).

  no
Variables

- Prolog is a dynamically typed language
- Variables are scoped locally to clauses, need not be declared and can hold any kind of value
- Variable must begin with a capital letter:
  - ?- X is 1+1, Y is 2*X.
    - X = 2, 
    - Y = 4
- When asked a question containing variable (goal), Prolog searches through all its facts to find an object that the variable can stand for (i.e, satisfy the goal):
  - ?- likes(john, Is_there_something_that_john_likes).
    - mary
- Variables do not vary (NB. resatisfying goals!):
  - ?- X is 1, X is 2.
    - no
Rules

- A rule is a general statement about objects and their relationships
- A rule is a Prolog clause containing a clause body
- A rule represents conditional knowledge, i.e. it is an implication in the form "clause head $\iff$ clause body" (the clause head is true if the clause body is true)
- John likes all people: list all people/facts => inefficient!
  - likes(john, mary).
  - likes(john, david).
  - ...
- "John likes any object provided it is a person"
  - likes(john, X) :- person(X).
- Example:
  - a :- b, c, d.
  - a is true if b and c and d are true, or $b \land c \land d \rightarrow a$
Conjunctions

- Means "and"
- The database:
  - likes(mary, food).
  - likes(mary, wine).
  - likes(john, wine).
  - likes(john, mary).
- Question: "Do John and Mary like each other?" (two goals to be satisfied)
  - ?- likes(john, mary), likes(mary, john).
    no
Conjunctions (cont.)

- Backtracking: repeated attempts to satisfy and resatisfy goals in a conjunction

- The database:
  - likes(mary, food).
  - likes(mary, wine).
  - likes(john, wine).
  - likes(john, mary).

- Question: ”Is there anything that John and Mary both like?”
  - ?- likes(mary, X), likes(john, X).
    
    X = wine

- Backtracking is chronological, i.e., control goes back to the most recently made choice and tries the next clause
The Early History: Before Prolog

- The insight that deduction could be used as computation was developed in the 1960’s through the work of Cordell Green and others: to consider a logical description of a problem as a program that could be executed efficiently.

- The work on Prolog was preceded by the Absys system. Absys (from Aberdeen System) was designed and implemented at the University of Aberdeen in 1967. For reasons that are unclear but that are probably cultural, Absys did not become widespread.
The Early History: Prolog

- Prolog was created as a competitor to LISP
- The name (abbreviation) is ambiguous:
  - The concept of logic programming in general, and
  - The specific programming language
- Evolved out of research at the University of Aix-Marseille and Edinburgh in the late 60’s and early 70’s
- Developed by Alain Colmerauer (+ Phillipe Roussel) and Robert Kowalski
- Colmerauer was interested in language processing and Kowalski in logic and theorem proving
- Prolog was the offspring of a successful marriage between these two
- Originally, the artificial intelligence group in Marrseille, consisted of Bob Pasero, Roussel, and Colmerauer and leded by Colmerauer was developing a natural language (French) question-answering system
- The first discussions between Kowalski and Colmerauer in 1971 were about using logic to represent grammar and using resolution to parse sentences
- Later in 1972 the idea of programming in predicate logic was born
Q-Systems, the Forerunner of Prolog

- Q-systems, a programming language which Colmerauer had developed for the machine translation project, played a prominent part in Prolog’s born

- Q-Systems were essentially a set of rewriting rules on sequences of complex symbols for processing natural languages
Q-Systems (cont.)

- The idea of unification was presented already in Q-Systems, although it worked in one direction
- Implemented by Colmerauer in ALGOL in 1969
- Developed in FORTRAN by Michel van Caneghem and Françoise Stellin
The Q-systems were very efficient and they are still (1996) used in a program which every day translates Canadian weather reports from English to French. An example of such a translation made in 1978:

*Cloudy with a chance of showers today and thursday => Nuageux avec possibilité d’averses aujourd’hui et jeudi.*
A primitive natural-language communication system was developed

User

Cats kill mice.
Tom is a cat who does not like mice who eat cheese.
Jerry is a mouse who eats cheese.
Max is not a mouse.
What does Tom do?

Computer

Tom does not like mice who eat cheese.
Tom kills mice.

User

Who is a cat?

Computer

Tom.

User

What does Jerry eat?

Computer

Cheese.

User

Who does not like mice who eat cheese?

Computer

Tom.

User

What does Tom eat?

Computer

What cats who do not like mice who eat cheese eat.
The logical formulae created made use of:

(i) constants representing elements,
   Tom, Jerry, Max, Cheese;
(ii) constants representing sets,
   Cats, Mice, MiceWhoEatCheese, CatsWhoDoNotLikeMiceWhoEatCheese;
(iii) constants representing binary relations between sets,
   Kill, DoesNotLike, Eat;
(iv) a functional symbol of arity 1 and two relational symbols of
   arity 2 and 3,
   \textbf{The}(a): represent the set consisting only of the element a
   \textbf{Subset}(x,y): expressed the inclusion of set x in set y
   \textbf{True}(r,x,y): expressed that the sets x and y were in the
   relation r
Prolog 0, an automated deductive system, was created in the fall of 1972 in connection with the development of the man-machine communication system.

Nondeterminism was processed by backtracking only a single branch of the search tree was stored at any given time in the memory (for efficiency reasons).

As opposed to a management system of several branch calculations simultaneously resident in memory, whose effect would have been to considerably increase the memory size required for execution of the deductions.
A number of built-in predicates were added to the system:
- Predicates to trace an execution,
- COPY to copy a term
- BOUM to split or create an identifier into a, or from a string of characters
- DIF to process the symbolic (that is, syntactic) equality

In current versions of Prolog, there is only a simple cut operation for search space reductions. In preliminary Prolog, there were different double punctuation marks at the end of each clause to perform different types of search space reductions:
- .. performed a cut after the head of the clause
- .; performed a cut after execution of the whole rule
- ;. performed a cut after production of at least one answer
- ;; had no effect (no cut)

These operators were abandoned later because of problems they cause to their users.
Adaptation of the resolution method comprising some very novel elements

New logical axioms

Phillipe Roussel implemented the interpreter in ALGOL-W on the IBM 360-67 machine

ALGOL-W was selected as the implementation languages because it was the only high-level language the developers had access to that enabled them to create structured objects dynamically, while also being equipped with garbage collection
Prolog 0 was created at the same time as its application.

The man-machine communication system was the first large application written Prolog program with 610 clauses.

An example of a text submitted to the system:
- Every psychiatrist is a person.
- Every person he analyzes is sick.
- Jacques is a psychiatrist in Marseille.
- Is Jacques a person?
- Where is Jacques?
- Is Jacques sick?

The answers obtained for the three questions at the end:
- Yes.
- In Marseille.
- I don't know.
All the inferences were made from pronouns (he, she, they, etc.), articles (the, a, etc.), subjects and complement relations with or without prepositions (from, to, etc.)

The system knew only about pronouns, articles, and prepositions (the vocabulary was encoded by 164 clauses)

Besides the communication system, two other applications were developed using this initial version of Prolog:
- A symbolic computation system
- A general problem-solving system called Sugiton
Syntax of Preliminary Prolog

- Example of syntax in preliminary Prolog:
  
  +APPEND(NIL,*Y,*Y).
  +APPEND(*A.*X,*Y,*A.*Z)
  -APPEND(*X,*Y,*Z).

- An asterisk used in front of the variables (in the Q-systems an asterisk was used after the variables)

- The same program written in normalized Prolog:
  
  append([],Y,Y).
  append([A|X],Y,[A|I|Z] :-
    append(X,Y,Z).
The Improved Prolog, i.e., Prolog I (1973)

- The next factors were considered in the improved version to simplify things: the choice of syntax, basic primitives, and the interpreter's computing methods.
- This version included all the new basic features of current Prologs.
- This system was first given the name "Marseille Prolog" and then Prolog I.
- This system’s operational semantics and its built-ins are essentially the same as in modern Prolog systems.
- The system had reasonable performance and was very influential in convincing people that programming in logic was a viable idea.
- This was the first version used outside Marseille and many different applications were developed using this version.
The Improved Prolog (cont.)

Some differences with the previous version:

- No more delayed evaluation (DIF, BOUM)
- All the classical built-in predicates like input and output
- A single operator for backtracking management, the search space cut operator (unique cut operation) "!", at that time written “/”
- The meta-call concept to use a variable instead of a literal
- Use of the predicates ANCESTOR and STATE, which have disappeared in present Prologs, to access ancestor literals and the current resolvent (considered as a term), for programmers wishing to define their own resolution mechanism
The Distribution of Prolog 1974-1975

- The fact that the interpreter was written in FORTRAN was very helpful in the distribution of the language
- The first applications:
  - Plan generation system
  - Symbolic computation system
  - Speech recognition system
- The first places it was used in:
  - Budapest
  - University of Warsaw
  - Toronto
  - University of Waterloo
  - University of Edinburgh
  - University of Montreal
  - IRIA, research center in Paris
Prolog II and Prolog III

- After Prolog I was released, Colmerauer’s group experimented with various mutations of Prolog and eventually released Prolog II with more built-in procedures.

- Prolog III: ”The Prolog III programming language extends Prolog by redefining fundamental process at its heart: unification” (Colmerauer)

- Later, Prolog became widely used in North America, Europe, Japan, etc.
The First Compiler

- The parser for the previous interpreter was very slow. It took about a second to parse each clause and users were beginning to complain.
- The first compiler, DEC-10 Prolog developed in 1977 by David Warren (with the help of Fernando Pereira and Luis Pereira).
- This structure-sharing system was fast and resulted in the widely acceptance of Prolog.
- It was competitive in performance to Lisp systems of the day and was for many years the highest performance Prolog system.
- Its syntax and semantics became the de facto standard, the "Edinburgh standard".
- An attempt to commercialize this system failed because of the demise of the DEC-10/20 machines and because of bureaucratic problems with the British government, which controlled the rights of all software developed with public funds.
The Warren Abstract Machine (WAM)

- David Warren presented the WAM in October 1983
- The original WAM is designed with an emulated implementation in mind (as opposed to native code)
- WAM = sequential control (call/return/jump instructions) + unification (get/put/unify instructions) + backtracking (try/retry/trust instructions) + optimizations (to use as little memory as possible)
Prolog systems can be divided into two categories: **structure-sharing** (introduced by Boyer and Moore) or **structure-copying** (introduced by Bruynooghe). The distinction is based on how compound terms are represented.

In a structure-sharing representation, all compound terms are represented as a pair of pointers (called a *molecule*): one pointer to an array containing the values of the term’s variables, and another pointer to a representation of the term’s non-variable part (the *skeleton*).

In a structure-copying representation, a concrete copy of a compound term is made whenever the compound term is matched against a variable.

It is faster to create terms in a structure-sharing representation. On the other hand, the unification of clauses is faster in a structure-copying representation. Memory usage of both techniques is similar in practice.

Early systems were mostly structure-sharing (e.g., DEC-10 Prolog and other earlier systems). Modern systems are mostly structure-copying (including WAM and WAM-based systems).

Later, the *program-sharing* approach was introduced, originated from *structure-sharing* but needs only one pointer to represent a dynamic instance of a structure [9].
The Influence of the WAM

- The development of the WAM in 1983 was very encouraging for Prolog developers.
- The WAM is simple on the outside (a small, clean instruction set) and complex on the inside (the instructions do complex things). This simultaneously helped and hindered implementation technology.
- Because the WAM is complex on the inside, for a long time many people used it “as is” and were happy with its level of performance.
- Because the WAM is simple on the outside, it was a perfect environment for extensions.
- After a few years, people were extending the WAM left and right.
- WAM became the de facto standard implementation technique.
Bridging the Gap Between DEC-10 (1977) Prolog and the WAM (1983)

- C-Prolog interpreter, developed at Edinburgh in 1982 by Fernando Pereira, Luis Damas, and Lawrence Byrd:
  - An important early system which is based on EMAS Prolog, a System completed in 1980 by Luis Damas
  - C-Prolog was one of the best interpreters, and is still a very usable system. It did much to create a Prolog programming community and to establish the Edinburgh standard. It is cheap, robust, portable (it is written in C), and fast enough for real programs

- Several other compiled systems including Prolog-X and NIP (New Implementation of Prolog). David Bowen, Lawrence Byrd, William Clocksin, and Fernando Pereira at Edinburgh were the main contributors in these
Many WAM variants have been developed for new logic languages, new computation models, and parallel systems. Three significant examples are:

- The CHIP (Constraint Handling In Prolog) constraint system, which interfaces the WAM with three constraint solvers
- The clp(FD) constraint system, which implements a glass box approach that allows constraint solvers to be written at the user level
- The SLG-WAM, which extends the WAM with memorization (a technique that caches already-computed answers to a predicate)
Prolog Performance

- The WAM was a large step towards the efficient execution of Prolog
- From the viewpoint of theorem proving, Prolog is extremely fast
- But there is still a large gap between the efficiency of the WAM and that of imperative language implementations
- As people started using Prolog for standard programming tasks, the gap became apparent and people started to optimize their systems
Limitations of Prolog Performance

- WAM instructions are too coarse-grained to perform much optimization.
- The majority of predicates written by human programmers are intended to give at most one solution, i.e., they are deterministic. Although these predicates are in reality case statements, they are too often compiled in an inefficient manner using the full generality of backtracking (which implies saving the machine state and repeated failure and state restoration).
- The single-assignment nature of Prolog (i.e., a variable can only be assigned one value in forward execution) needs to be handled well.
- Prolog has dynamic typing (variables may contain values of any type) and dynamic memory allocation (all data objects are allocated at run-time). Both of these cost execution time. They should be compiled statically wherever possible.
- Programming style has a large effect on a program’s efficiency. Prolog programming is at a high level of abstraction, so it hides many details of the implementation from the programmer, making it difficult to improve efficiency when it is important to do so (for example, using cut can make a big difference).
- The apparent need for architectural support. So-called “general-purpose” architectures are in fact optimized for imperative languages. To run Prolog equally well, either the compiler must do more work, or the architecture should be modified. Fast memory system is an important architectural need because of Prolog’s dynamic.
Other Execution Models

Some developments in Prolog implementation are based on novel models of execution very different from the WAM:

- **The Vienna Abstract Machine (VAM)**
  - Developed by Andreas Krall at the Technische Universität Wien (Vienna, Austria), the VAM is an execution model faster than the WAM
  - The idea is that instead of building predicate’s arguments before the predicate is called, perform argument setup and argument unification at the same time
  - For example, the call p(X,[a,b,c],Y) to the definition p(A,_,B)

- **BinProlog**
  - Developed by Paul Tarau at the Universite de Moncton (Canada), the idea of BinProlog is to transform each Prolog clause into a binary clause, i.e., a clause containing only one body goal. The body goal is given an extra argument, which represents its success continuation, i.e., the sequence of goals to be executed if the body completes successfully
  - The binary clause transformation can be the basis of a system that uses very little memory yet compiles and executes very quickly
Prolog Evolution, the Systems View

- Number of different software implementations of Prolog since WAM in 1983, many of them substantially compatible with the Edinburgh standard
- Including MProlog, IF/Prolog, SNI-Prolog, MU-Prolog, NU-Prolog, Quintus, BIM, IBM Prolog, SEPIA, ECLiPSe, SB-Prolog, XSB, SICStus, and Aquarius
- Almost all commercial systems support modules
- Almost all commercial systems have a full-featured foreign language interface, and many of them (including Quintus, BIM, IF/Prolog, SNI-Prolog, SICStus, and ECLiPSe) allow arbitrarily nested calls between Prolog and C
- In what follows, for each system are listed its most important contributions to implementation technology
MProlog

- The first commercial Prolog system developed in Hungary starting in 1978
- The implementation is based on Warren’s pre-WAM three-stack model of DEC-10 Prolog
- A full-featured structure-sharing system with all Edinburgh built-ins, debugging, foreign language interface, sophisticated I/O and garbage collection for the symbol table and code area (but not for the stacks)
IF/Prolog and SNI-Prolog

- Developed at InterFace Computer GmbH, in Munich, Germany, was commercialized in 1983
- The first release was an interpreter, a WAM-based compiler released in 1985
- Siemens-Nixdorf Informationssysteme AG bought the IF/Prolog sources in 1986, when it was ported and extended and then became SNI-Prolog
- SNI-Prolog was completely redesigned from scratch in 1990
- The current system conforms to the ISO Prolog standard, supports constraints, has been ported to more platforms and has improved system behavior (more flexible interfaces and less memory usage)
- Both SNI-Prolog and IF/Prolog have extensive C interoperability.
- They have configurable memory management and garbage collection of all Prolog memory areas
MU-Prolog and NU-Prolog were developed at Melbourne University.

Neither system does garbage collection.

MU-Prolog is a structure-sharing interpreter. The original version (1.0) was written by John Lloyd in Pascal. Version 2.0 was written by Naish and completed in 1982.

NU-Prolog is a WAM-based emulator written in C and completed in 1985.

NU-Prolog is interesting for its pioneering implementation of logical negation, quantifiers and if-then-else, through extensions to the WAM.

NU-Prolog was the basis for many implementation experiments, e.g., related to parallelism, databases, and programming environments.
Quintus Prolog

- Quintus Prolog, developed in Palo Alto in 1985, is the first widely known and probably the best-known commercial Prolog system. Its syntax and semantics became a de facto standard.
- It is close to the Edinburgh syntax and is highly compatible with C-Prolog.
- It has several distinguishing features, e.g.
  - It generates the most compact code (compact code is significant specially for applications with large databases).
  - Source-level debugging and an Emacs interface.
  - Large set of libraries of useful utilities.
BIM Prolog (ProLog by BIM)

- Was developed by the BIM company in Everberg, Belgium in 1985
- Has WAM execution model
- Only ran on a few machines
- Several notable contributions:
  - It was the first WAM-based system to do heap and symbol table garbage collection and provide modules
  - It was the first system to provide a source-level graphical debugger and an external database interface
IBM Prolog

- IBM Prolog was developed at IBM Paris
- The first version, a structure-sharing system, was written in 1983–1984
- It was commercialized in 1985 as VM/Prolog
- A greatly rewritten and extended version was commercialized in 1989 as IBM Prolog
- The system is WAM-based and has a foreign language interface
- Only ran on a few machines
SICStus Prolog

- Was developed at SICS (Swedish Institute of Computer Science) near Stockholm
- The first version was distributed in 1986, but the popularity came with the release 0.5 in 1987
- SICStus was cheap, robust, fast, highly compatible with the “Edinburgh standard” and it has been ported to many machines. These properties made it very popular in the early 1990’s.
Due to faster machines and improved compilation technology, the performance of Prolog increased in 1993 about two orders of magnitude since DEC-10 Prolog
- System performance was improved as Prolog was better understood

Interest in building hardware architectures optimized for Prolog was started in the early 1980’s

This interest was catalyzed by two events:
- The start of the Japanese Fifth Generation Project in 1982
- The development of the WAM in 1983

Several project including:
- ICOT (Japanese Institute for New Generation Technology) and the PSI (Personal Sequential Inference) Machines 1985
- ECRC (European Computer-Industry Research Centre) and the KCM (Knowledge Crunching Machine) started in 1989
- The PLM (The Programmed Logic Machine) and the VLSI-BAM (The VLSI Berkeley Abstract Machine) in 1991
Prolog Standards

- Edinburgh Prolog
  - The syntax and semantics characteristics of Warren’s system became the de facto standard since 1977

- ISO Prolog
  - In 1995, ISO Prolog standardization made to ensure its individual parts remains fixed
Influence on Other Languages

- Visual Prolog
  - Strongly typed object-oriented dialect of Prolog, first developed as Borland’s Turbo Prolog in 1980
- Mercury
- Oz
- Erlang
- Strand
- KL0
- KL1
- Datalog
The Simplification Principle

- The main principle in compiling Prolog, which underlies every Prolog compiler, is to simplify each occurrence of one of its basic operations (namely, unification and backtracking).

- Compiling Prolog is feasible because this simplification is so often possible. For example, unification is often used purely as a parameter passing mechanism. Most such cases are easily detected and compiled into efficient code.

- Simplification can be done statically and locally (each functor separately) as in WAM.

- Or it can be done dynamically (e.g., clause selection) and globally (e.g., global analysis and unification of a complete term as a whole).
Prolog Applications

- The earliest applications were in the domain of natural-language programming. NLP is supported by a built-in mechanism for parsing context-free grammars.
- Today used in knowledge-based or expert systems, computer algebra, etc.
- Symbolic computing, including:
  - Relational databases
  - Mathematical logic
  - Abstract problem solving
  - Understanding natural languages
  - Architectural design
  - Symbolic equation solving
  - Biochemical structure analysis
  - Many areas of artificial intelligence, etc.
Prolog and the Mainstream

- As measured by the number of users, commercial systems, and practical applications, Prolog is by far the most successful logic programming language. Its closest competitors are surely the special-purpose constraint languages.
- However, logic programming in particular and declarative programming in general remain outside of the mainstream of computing.
- Two important factors that hinder the widespread acceptance of Prolog are:
  - **Compatibility.** Existing code works and investment in it is large. Therefore people will not easily abandon it for new technology.
  - **Public perception.** To the commercial computing community, the terms “Prolog” and “logic programming” are at best perceived as useful in an academic or research setting, but not useful for industry.
- The ideas of logic programming will continue to be used in those application domains for which it is particularly suited. This includes domains in which program complexity is beyond what can be managed in the imperative paradigm.
Conclusion

- “… Prolog is essentially a theorem prover… Our contribution was to transform that theorem prover into a programming language.” Colmerauer and Roussel [3]
- In summary, Prolog was basically a subset of Planner that restricted programs to clausal form using backward chaining. Carl Hewitt [5]
- ”Prolog would have had its place in computer science regardless of its discoverers”. Jacques Cohen [2]
References