# UNIT 2 PROGRAMMING LANGUAGES FOR ARTIFICIAL INTELLIGENCE

**CONTENTS**



## 1.0 INTRODUCTION

Artificial intelligence researchers have developed several specialized programming languages for artificial intelligence:

> IPL was the first language developed for artificial intelligence. It includes features intended to support programs that could perform general problem solving, including lists, associations, schemas (frames), dynamic memory allocation, data types, recursion, associative retrieval, functions as arguments, generators (streams), and cooperative multitasking.
>
> Lisp is a practical mathematical notation for computer programs based on lambda calculus. Linked lists are one of Lisp languages'

Major data structures, and Lisp source code is itself made up of lists. As a result, Lisp programs can manipulate source code as a data structure, giving rise to the macro systems that allow programmers to create new syntax or even new domain-specific programming languages embedded in Lisp. There are many dialects of Lisp in use today, among them are Common Lisp, Scheme, and Clojure.
Prolog is a declarative language where programs are expressed in terms of relations, and execution occurs by running *queries* over these relations. Prolog is particularly useful for symbolic reasoning, database and language parsing applications. Prolog is widely used in AI today.
A STRIP is a language for expressing automated planning problem instances. It expresses an initial state, the goal states, and a set of actions. For each action preconditions (what must be established before the action is performed) and post conditions (what is established after the action is performed) are specified.
Planner is a hybrid between procedural and logical languages. It gives a procedural interpretation to logical sentences where implications are interpreted with pattern-directed inference.

AI applications are also often written in standard languages like C++ and languages designed for mathematics, such as MATLAB and Lush. This unit will deal only on IPL, Lisp and Prolog.

## 2.0 OBJECTIVES

At the end of this unit, you should be able to:

describe the history of IPL
discuss the similarities between lisp and prolog programming
list the areas where lisp can be used.

## 3.0 MAIN CONTENT

### 3.1 IPL Programming Language

Information Processing Language (IPL) is a programming language developed by Allen Newell, Cliff Shaw, and Herbert Simon at RAND Corporation and the Carnegie Institute of Technology from about 1956. Newell had the role of language specifier-application programmer, Shaw was the system programmer and Simon took the role of application programmer-user.

The language includes features intended to support programs that could perform general problem solving, including lists, associations, schemas (frames), dynamic memory allocation, data types, recursion, associative retrieval, functions as arguments, generators (streams), and cooperative multitasking. IPL pioneered the concept of list processing, albeit in an assembly-language style.

### 3.1.1 A taste of IPL

An IPL computer has:

1. A set of *symbols*. All symbols are addresses, and name cells. Unlike symbols in later languages, symbols consist of a character followed by a number, and are written H1, A29, 9-7, 9-100.

   Cell names beginning with a letter are *regional*, and are absolute addresses.
   Cell names beginning with "9-" are *local*, and are meaningful within the context of a single list. One list's 9-1 is independent of another list's 9-1.
   Other symbols (e.g., pure numbers) are *internal*.

2. A set of *cells*. Lists are built from several cells holding mutual references. Cells have several fields:

   P, a 3-bit field used for an operation code when the cell is used as an instruction and unused when the cell is data.
   Q, a 3-valued field used for indirect reference when the cell is used as an instruction and unused when the cell is data.
   SYMB, a symbol used as the value in the cell.

3. A set of *primitive processes*, which would be termed *primitive functions* in modern languages.

### 3.1.2 History of IPL

The first application of IPL was to demonstrate that the theorems in *Principia Mathematica* which were laboriously proven by hand, by Bertrand Russell and Alfred North Whitehead, could in fact be proven by computation. According to Simon's autobiography *Models of My Life*, this first application was developed first by hand simulation, using his children as the computing elements, while writing on and holding up note cards as the registers which contained the state variables of the program.

IPL was used to implement several early artificial intelligence programs, also by the same authors: the Logic Theory Machine (1956), the General Problem Solver (1957), and their computer chess program NSS (1958).
Several versions of IPL were created: IPL-I (never implemented), IPL-II (1957 for JOHNNIAC), IPL-III (existed briefly), IPL-IV, IPL-V (1958, for IBM 650, IBM 704, IBM 7090, many others. Widely used), IPL-VI. However the language was soon displaced by Lisp, which had far more powerful features, a simpler syntax, and the benefit of automatic garbage collection.

## 3.2 Lisp Programming Language

Lisp (or LISP) is a family of computer programming languages with a long history and a distinctive, fully parenthesized syntax. Originally specified in 1958, Lisp is the second-oldest high-level programming language in widespread use today; only FORTRAN is older (by one year). Like FORTRAN, Lisp has changed a great deal since its early days, and a number of dialects have existed over its history. Today, the most widely known general-purpose Lisp dialects are Common Lisp, Scheme, and Clojure.

Lisp was originally created as a practical mathematical notation for computer programs, influenced by the notation of Alonzo Church's lambda calculus. It quickly became the favored programming language for artificial intelligence (AI) research. As one of the earliest programming languages, Lisp pioneered many ideas in computer science, including tree data structures, automatic storage management, dynamic typing, and the self-hosting compiler.

The name *LISP* derives from "LISt Processing". Linked lists are one of Lisp languages' major data structures, and Lisp source code is itself made up of lists. As a result, Lisp programs can manipulate source code as a data structure, giving rise to the macro systems that allow programmers to create new syntax or even new domain-specific languages embedded in Lisp.

The interchange ability of code and data also gives Lisp its instantly recognizable syntax. All program code is written as *s-expressions*, or parenthesized lists. A function call or syntactic form is written as a list with the function or operator's name first, and the arguments following; for instance, a function f that takes three arguments might be called using (f arg1 arg2 arg3).

### 3.2.1 History

Interest in artificial intelligence first surfaced in the mid 1950. Linguistics, psychology, and mathematics were only some areas of application for AI. Linguists were concerned with natural language processing, while psychologists were interested in modelling human information and retrieval. Mathematicians were more interested in automating the theorem proving process. The common need among all of these applications was a method to allow computers to process symbolic data in lists.

IBM was one of the first companies interested in AI in the 1950s. At the same time, the FORTRAN project was still going on. Because of the high cost associated with producing the first FORTRAN compiler, they decided to include the list processing functionality into FORTRAN. The FORTRAN List Processing Language (FLPL) was designed and implemented as an extension to FORTRAN.

In 1958 John McCarthy took a summer position at the IBM Information Research Department. He was hired to create a set of requirements for doing symbolic computation. The first attempt at this was differentiation of algebraic expressions. This initial experiment produced a list of language requirements, most notably was recursion and conditional expressions. At the time, not even FORTRAN (the only high-level language in existence) had these functions.

It was at the 1956 Dartmouth Summer Research Project on Artificial Intelligence that John McCarthy first developed the basics behind Lisp. His motivation was to develop a list processing language for Artificial Intelligence. By 1965 the primary dialect of Lisp was created (version 1.5). By 1970 special-purpose computers known as Lisp Machines, were designed to run Lisp programs. 1980 was the year that object-oriented concepts were integrated into the language. By 1986, the X3J13 group formed to produce a draft for ANSI Common Lisp standard. Finally in 1992, X3J13 group published the American National Standard for Common Lisp.

**Since 2000**

After having declined somewhat in the 1990s, Lisp has recently experienced a resurgence of interest. Most new activity is focused around open source implementations of Common Lisp, and includes the development of new portable libraries and applications. This interest can be measured partly by sales from the print version of *Practical Common Lisp* by Peter Seibel, a tutorial for new Lisp programmers published in 2004.

It was briefly Amazon.com's second most popular programming book. It is available free online.http://en.wikipedia.org/wiki/Lisp_ (programming_language) - cite_note-15

Many new Lisp programmers were inspired by writers such as Paul Graham and Eric S. Raymond to pursue a language others considered antiquated. New Lisp programmers often describe the language as an eye-opening experience and claim to be substantially more productive than in other languages. This increase in awareness may be contrasted to the "AI winter" and Lisp's brief gain in the mid-1990s.

Dan Weinreb lists in his survey of Common Lisp implementations eleven actively maintained Common Lisp implementations. Scieneer Common Lisp is a new commercial implementation forked from CMUCL with a first release in 2002.

The open source community has created new supporting infrastructure: Cliki is a wiki that collects Common Lisp related information, the Common Lisp directory lists resources, #lisp is a popular IRC channel (with support by a Lisp-written Bot), lisppaste supports the sharing and commenting of code snippets, Planet Lisp collects the contents of various Lisp-related Blogs, on LispForum user discuss Lisp topics, Lisp jobs is a service for announcing job offers and there is a new weekly news service (Weekly Lisp News). Common-lisp.net is a hosting site for open source Common Lisp projects.

50 years of Lisp (1958–2008) has been celebrated at LISP50@OOPSLA. There are several regular local user meetings (Boston, Vancouver, Hamburg,), Lisp Meetings (European Common Lisp Meeting, European Lisp Symposium) and an International Lisp Conference.

The Scheme community actively maintains over twenty implementations. Several significant new implementations (Chicken, Gambit, Gauche, Ikarus, Larceny, and Ypsilon) have been developed in the last few years. The Revised Report on the Algorithmic Language Scheme standard of Scheme was widely accepted in the Scheme community. The Scheme Requests for Implementation process has created a lot of quasi standard libraries and extensions for Scheme. User communities of individual Scheme implementations continue to grow. A new language standardization process was started in 2003 and led to the RRS Scheme standard in 2007. Academic use of Scheme for teaching computer science seems to have declined somewhat. Some universities are no longer using Scheme in their computer science introductory courses.

There are several new dialects of Lisp: Arc, Nu, and Clojure.

### 3.2.2 Connection to artificial intelligence

Since its inception, Lisp was closely connected with the artificial intelligence research community, especially on PDP-10http://en. wikipedia.org/wiki/Lisp_(programming_language)- cite_note-5 systems. Lisp was used as the implementation of the programming language Micro Planner which was used in the famous AI system SHRDLU. In the 1970s, as AI research spawned commercial offshoots, the performance of existing Lisp systems became a growing issue.

### 3.2.3 Areas of Application

Lisp totally dominated Artificial Intelligence applications for a quarter of a century, and is still the most widely used language for AI. In addition to its success in AI, Lisp pioneered the process of *Functional Programming*. Many programming language researchers believe that functional programming is a much better approach to software development, than the use of Imperative Languages (Pascal, C++, etc).

Below is a short list of the areas where Lisp has been used:

Artificial Intelligence

- AI Robots
- Computer Games (Craps, Connect-4, BlackJack)
- Pattern Recognition

Air Defense Systems
Implementation of Real-Time, embedded Knowledge-Based Systems
List Handling and Processing
Tree Traversal (Breath/Depth First Search)
Educational Purposes (Functional Style Programming)

### 3.2.4 Syntax and semantics

#### Symbolic expressions

Lisp is an expression-oriented language. Unlike most other languages, no distinction is made between "expressions" and "statements"; all code and data are written as expressions. When an expression is *evaluated*, it produces a value (in Common Lisp, possibly multiple values), which then can be embedded into other expressions. Each value can be any data type.

McCarthy's 1958 paper introduced two types of syntax: S-expressions (Symbolic expressions, also called "sexps"), which mirror the internal representation of code and data; and M-expressions (Meta Expressions), which express functions of S-expressions. M-expressions never found favour, and almost all Lisps today use S-expressions to manipulate both code and data.

The use of parentheses is Lisp's most immediately obvious difference from other programming language families. As a result, students have long given Lisp nicknames such as *Lost in Stupid Parentheses*, or *Lots of Irritating Superfluous Parentheses*.[23] However, the S-expression syntax is also responsible for much of Lisp's power: the syntax is extremely regular, which facilitates manipulation by computer. However, the syntax of Lisp is not limited to traditional parentheses notation. It can be extended to include alternative notations. XMLisp, for instance, is a Common Lisp extension that employs the metaobject-protocol to integrate S-expressions with the Extensible Markup Language (XML).

The reliance on expressions gives the language great flexibility. Because Lisp functions are themselves written as lists, they can be processed exactly like data. This allows easy writing of programs which manipulate other programs (metaprogramming). Many Lisp dialects exploit this feature using macro systems, which enables extension of the language almost without limit.

## 3.3 Prolog Programming Language

Prolog is a general purpose logic programming language associated with artificial intelligence and computational linguistics.

Prolog has its roots in first-order logic, a formal logic, and unlike many other programming languages, Prolog is declarative: 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.

The language was first conceived by a group around Alain Colmerauer in Marseille, France, in the early 1970s and the first Prolog system was developed in 1972 by Colmerauer with Philippe Roussel.

Prolog was one of the first logic programming languages, and remains among the most popular such languages today, with many free and commercial implementations available. While initially aimed at natural language processing, the language has since then stretched far into other areas like theorem proving, expert systems, games, automated answering systems, ontologies and sophisticated control systems. Modern Prolog environments support creating graphical user interfaces, as well as administrative and networked applications.

### 3.3.1 History of Prolog

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 Alain Colmerauer and Phillipe Roussel. The first implementations of Prolog were interpreters; however, David H. D. Warren created the Warren Abstract Machine, an early and influential Prolog compiler which came to define the "Edinburgh Prolog" dialect which served as the basis for the syntax of most modern implementations.

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 prove with Horn clauses of the form:

### 3.3.2 Prolog 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*. 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 extra logical features.

### 3.3.2.1 Data Types

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

An atom is a general-purpose name with no inherent meaning. Examples of atoms include x, blue, 'Taco', and 'some atom'.

Numbers can be floats or integers.

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. Examples of compound terms are truck_year('Mazda', 1986) and '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, []. For example [1,2,3] or [red,green,blue].

*Strings*: A sequence of characters surrounded by quotes is equivalent to a list of (numeric) character codes, generally in the local character encoding, or Unicode if the system supports Unicode. For example, "to be, or not to be".

### 3.3.2.2 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 predicate ,/2 (meaning a 2-arity 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(tom).
which is equivalent to the rule?
cat(tom) :- true.
The built-in predicate true/0 is always true.
Given the above fact, one can ask:
*is tom a cat?*
?- cat(tom).
Yes
*what things are cats?*
?- cat(X).
X = tom
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 = tom

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) 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.

### 3.3.2.3 Evaluation

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.

### 3.3.2.4 Loops and recursion

Iterative algorithms can be implemented by means of recursive predicates.

### 3.3.2.5 Negation

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

is evaluated as follows: Prolog attempts to prove the legal(X). If a proof for that goal can be found, the original goal (i.e., \+ legal(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 ?- illegal(X). can now not be used to enumerate all things that are illegal.

### 3.3.2.6 Examples

Here follow some example programs written in Prolog.
Hello world
An example of a query:
?- write('Hello world!'), nl.
Hello world!
true.

?-

### 3.3.2.7 Criticism

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. Most applications are small by industrial standards, with few exceeding 100,000 lines of code. 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. 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."

Software developed in Prolog has been criticised for having a high performance penalty compared to conventional programming languages. However, advances in implementation methods have reduced the penalties to as little as 25%-50% for some applications.

### 3.3.2.8 Types

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

## 4.0 CONCLUSION

IPL, Lisp and Prolog considered in this unit are among other specialized programming languages for artificial intelligence.

## 5.0 SUMMARY

In this unit, you learnt that:

IPL is the pi1ered concept of list processing.

Lisp is the second-oldest high-level programming language in widespread use today

Prolog was 1 of the first logic programming languages, and remains among the most popular such languages today, with many free and commercial implementations available.

## 6.0 TUTOR-MARKED ASSIGNMENT

1. Describe Prolog programming Language.
2. Describe Lisp programming Language.
3. List three (3) areas where Lisp can be used.

## 7.0 REFERENCES/FURTHER READING

Crevier, D. (1993). *AI: The Tumultuous Search for Artificial Intelligence*. New York, NY: BasicBooks, ISBN 0-465-02997-3

McCarthy, J. (1979). *History of Lisp.* "LISP prehistory - Summer 1956 through Summer 1958."

Nilsson, N. (1998). *Artificial Intelligence: A New Synthesis*. Morgan: Kaufmann Publishers, ISBN 978-1-55860-467-4.

Poole, D. Mackworth, A. Goebel, R. (1998). *Computational Intelligence: A Logical Approach*. New York: Oxford University Press, ISBN 0195102703, http://www.cs.ubc.ca/spider/poole/ ci.html

Russell, S. J.& Norvig, Peter (2003). *Artificial Intelligence: A Modern Approach* (2nd ed.), Upper Saddle River, New Jersey: Prentice Hall, ISBN 0-13-790395-2, http://aima.cs.berkeley.edu/

Sebesta, R.W. (1996). *Concepts of Programming Languages,* (Third Edition). Menlo Park, California: Addison-Wesley Publishing Company.