20 Lisp: Final Thoughts

Both Lisp and Prolog are based on formal mathematical models of computation: Prolog on logic and theorem proving, Lisp on the theory of recursive functions. This sets these languages apart from more traditional languages whose architecture is just an abstraction across the architecture of the underlying computing (von Neumann) hardware. By deriving their syntax and semantics from mathematical notations, Lisp and Prolog inherit both expressive power and clarity.

Although Prolog, the newer of the two languages, has remained close to its theoretical roots, Lisp has been extended until it is no longer a purely functional programming language. The primary culprit for this diaspora was the Lisp community itself. The pure lisp core of the language is primarily an assembly language for building more complex data structures and search algorithms. Thus it was natural that each group of researchers or developers would "assemble" the Lisp environment that best suited their needs. After several decades of this the various dialects of Lisp were basically incompatible. The 1980s saw the desire to replace these multiple dialects with a core Common Lisp, which also included an object system, CLOS. Common Lisp is the Lisp language used in Part III.

But the primary power of Lisp is the fact, as pointed out many times in Part III, that the data and commands of this language have a uniform structure. This supports the building of what we call *meta-interpreters*, or similarly, the use of *meta-linguistic abstraction*. This, simply put, is the ability of the program designer to build interpreters within Lisp (or Prolog) to interpret other suitably designed structures in the language. We saw this many time in Part III, including building a Prolog interpreter in Lisp, the design of the expert system interpreter `lisp-shell`, and the ID3 machine learning interpreter used for data mining. But Lisp is, above all, a practical programming language that has grown to support the full range of modern techniques. These techniques include functional and applicative programming, data abstraction, stream processing, delayed evaluation, and object-oriented programming.

The strength of Lisp is that it has built up a range of modern programming techniques as extensions of its core model of functional programming. This set of techniques, combined with the power of lists to create a variety of symbolic data structures, forms the basis of modern Lisp programming. Part III is intended to illustrate that style.

Partly as a result of the Lisp diaspora that produced Common Lisp, was the creation of a number of other functional programming languages. With the desire to get back to the semantic foundations on which McCarthy created Lisp (R*ecursive functions of symbolic expressions and their computation by machine*, 1960), several important functional language developments began. Among these we mention Scheme, SML, and OCaml. Scheme, a small, sometimes called *academic* Lisp, was developed by Guy Steele and Gerald Sussman in the 1970s. Scheme chose static, sometimes called *lexical*, scope over the dynamic scope of Common Lisp. For references on Scheme see Gerald Sussman and Guy Steele. *SCHEME: An Interpreter for Extended Lambda Calculus,* AI Memo 349, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, December 1975 and *The Scheme Programming Language* by R. Kent Dybvig (1996).

Standard ML (SML) is a general-purpose functional language with compile-time type checking. Type inference procedures use compile-time checking to limit run-time errors. For further information see Robin Milner, Mads, Tofte, Robert Harper, and David MacQueen. (1997). *The Definition of Standard ML (Revised)*. Objective Caml or Ocaml is an object-oriented extension to the functional language Caml. It has an interactive interpreter, a byte-code compiler, and an optimized native-code compiler. It integrates object-orientation with functional programming with SML-like type inference. The language is maintained by INRIA; for further details see *Introduction to Objective Caml* by Jason Hickey (2008) and *Practical OCaml* by Joshua Smith (2006).

In designing the algorithms of Part III, we have been influenced by Abelson and Sussman's book *The Structure and Interpretation of Computer Programs* (1985). Steele (1990) offers an essential guide to using Common Lisp. Valuable tutorials and textbooks on Lisp programming include *Lisp* (Winston and Horn 1984), *Common LispCraft* (Wilensky 1986), *Artificial Intelligence Programming*, Charniak et al. (1987), *Common Lisp Programming for Artificial Intelligence* (Hasemer and Domingue 1989), *Common Lisp: A Gentle Introduction to Symbolic Computation* (Touretzky 1990), *On Lisp: Advanced Techniques for Common Lisp* (Graham 1993), and *ANSI Common Lisp* (Graham 1995).

A number of books explore the use of Lisp in the design of AI problem solvers. *Building Problem Solvers* (Forbus and deKleer 1993) is an encyclopedic treatment of AI algorithms in Lisp and an invaluable reference for AI practitioners. Also, see any of a number of general AI texts that take a more Lisp-centered approach to the basic material, including *The Elements of Artificial Intelligence Using Common Lisp* by Steven Tanimoto (1990). Finally, we mention *Practical Common Lisp* an introductory book on Common Lisp by Peter Seibel (2004).