The exercises in this book are intended to deepen your understanding of ML and improve your programming skills. But such exercises cannot turn you into a programmer, let alone a software engineer. A project is more than a large programming exercise; it involves more than programming. It demands careful preparation: background study, analysis of requirements, design. The finished program should be evaluated fairly but thoroughly.

Each suggestion is little better than a hint, but with a little effort, can be developed into a proper proposal. Follow the attached references and prepare a project description including a statement of objectives, a provisional timetable and a list of required resources. The next stage is to write a detailed requirements analysis, listing all functions in sufficient detail to allow someone else to carry out eventual testing. Then specify the basic design; ML functors and signatures can describe the main components and their interfaces.

The preparatory phases outlined above might be done by the instructor, a student or a team of students. This depends upon the course aims, which might be concerned purely with ML, with project management, or with demonstrating some methodology of software engineering. The final evaluation might similarly be done by the instructor, the implementor or another team of students.

The evaluation should consider to what extent the program meets its objectives. Testing can be driven by the requirements analysis. Many projects are easy to do, but hard to do efficiently. Evaluation thus should consider performance as well as correctness; profiling tools can identify performance bottlenecks. Students might be expected to find and use the standard library modules — arrays, low-level word operations, etc. — that are appropriate for efficiency.

Some of the suggested projects have been done by Cambridge students, though not necessarily in ML. The others are there because they are interesting (at least to me) and are of suitable difficulty. They are intended to be particularly appropriate for ML — though practically anything can be done in ML unless it requires unsafe programming or is embedded in a system that mandates some other language. So, feel free to adopt project suggestions from other sources.

*Unlimited precision integer arithmetic* gives exact answers and never fails
due to overflow. (Some ML systems provide this by default.) Knuth (1981) describes the algorithms, which apart from division are straightforward. He also suggests improved algorithms for operating on rational numbers.

**Unlimited precision real arithmetic** yields answers that are correct to any desired precision, automatically determining the precision required for intermediate calculations. Much effort has been devoted to finding the most efficient representation of a real number (Boehm and Cartwright, 1990). Ménissier-Morain (1995) recommends a convergent series of rationals of the form \( p/B^q \). Computational stunts on this theme may be amusing, though definitely not easy (Gourdon and Salvy, 1993). You could also develop the numerical examples of Section 5.15.

The **polynomial arithmetic** example of Chapter 3 can be extended in several directions. You could provide additional operations, allow more than one variable or even implement a better GCD algorithm. See Davenport et al. (1993) or Knuth (1981). Unlimited precision integers are required.

**Emulators** can be fun: you bring an obsolete machine and its quaint software back to life. My personal favourite is the DEC PDP-8. The basic model can address 4096 12-bit words and has an instruction set with eight opcodes. The manuals are out of print, but information is available on the World Wide Web (Jones, 1995). Details such as the precise treatment of interrupts can be tricky. Emulated software must run fast enough to keep up with the user’s typing!

**Advanced tautology checkers** include ordered binary decision diagrams and the Davis-Putnam proof procedure. OBDDs have applications in hardware and systems verification; Bryant (1992) is a classic description but Moore (1994) may be more appropriate for functional programming. Davis-Putnam is back in favour after many years; early books mention it (Chang and Lee, 1973), but the latest algorithms are described only in technical reports (Zhang and Stickel, 1994).

**Theorem provers** can be built in various ways upon the foundation provided in Chapter 10. The tableau method is easy to implement (Beckert and Posegga, 1995). Model elimination is also fairly straightforward (Stickel, 1988a). Andrews (1989) describes the matrix method in the context of higher-order logic; it is equally applicable to first-order logic. Only the ablest student should try implementing the resolution method (Stickel, 1988b); the refinements necessary for high performance demand complex data structures (Butler and Overbeek, 1994).

Consider writing a **parser generator**: a simple LR(0) one, an SLR one, or
perhaps an LALR(1) version with sophisticated error recovery. Good compiler
texts, such as Aho et al. (1986), describe the necessary techniques.

**Compiling** projects are always popular. Select a small subset of ML and write
an interpreter for it. The SECD machine yields call-by-value semantics, while
graph reduction yields call-by-need. Field and Harrison (1988) describe such
implementation methods, as well as type checking. Unless the syntax is trivial,
use a parser generator such as ML-Yacc (Tarditi and Appel, 1994).

You should be familiar at least with Chapters 2–5, preferably also Chapters 7
and 8, before attempting any substantial project. Good luck!
Project Suggestions
BIBLIOGRAPHY


Bibliography


Bibliography


