In earlier chapters we have developed a general framework for the design of a compiler. We have considered how the task and its data structures could be decomposed, what tools and strategies are available to the compiler writer, and what problems might be encountered. Given a source language, target machine and performance goals for the generated code we can design a translation algorithm. The result of the design is a set of module specifications.

This chapter is concerned with issues arising out of the implementation of these specifications. We first discuss the decisions that must be made by the implementors and the criteria that guide these decisions. Unfortunately, we can give no quantitative relationship between decisions and criteria! Compiler construction remains an art in this regard, and the successful compiler writer must simply develop a feel for the inevitable compromises. We have therefore included three case studies of successful compilers that make very different architectural decisions. For each we have tried to identify the decisions made and show the outcome.

14.1. Implementation Decisions

Many valid implementations can generally be found for a set of module specifications. In fact, an important property of a module is that it hides one or more implementation decisions. By varying these decisions, one obtains different members of a ‘family’ of related programs. All of the members of such a family carry out the same task (defined by the module specifications) but generally satisfy different performance criteria. In our case, we vary the
pass structure and data storage strategies of the compiler to satisfy a number of criteria presented in Section 14.1.1. Despite this variation, however, the module specifications remain unchanged. This point is an extremely important one to keep in mind, especially since many implementation languages provide little or no support for the concept of a module as a distinct entity. With such languages it is very easy to destroy the modular decomposition during development or maintenance, and the only protection one has against this is eternal vigilance and a thorough understanding of the design.

14.1.1. Criteria **Maintainability**, **performance** and **portability** are the three main criteria used in making implementation decisions. The first is heavily influenced by the structure of the program, and depends ultimately on the quality of the modular design. Unfortunately, given current implementation languages, it is sometimes necessary to sacrifice some measure of maintainability to achieve performance goals. Such tradeoffs run counter to our basic principles. We do not lightly recommend them, but we recognize that in some cases the compiler will not run at all unless they are made. We do urge, however, that all other possibilities be examined before such a decision is taken.

Performance includes memory requirements, secondary storage requirements and processing time. Hardware constraints often place limits on performance tradeoffs, with time the only really free variable. In Sections 14.1.2 and 14.1.3 we shall be concerned mainly with tradeoffs between primary and secondary storage driven by such constraints.

Portability can be divided into two sub-properties often called **rehostability** and **retargetability**. Rehosting is the process of making the compiler itself run on a different machine, while retargeting is the process of making it generate code for a different machine. Rehostability is largely determined by the implementation language and the performance tradeoffs that have been made. Suppose, for example, that we produce a complete design for a Pascal compiler, specifying all modules and interfaces carefully. If this design is implemented by writing a FORTRAN program that uses only constructs allowed by the FORTRAN standard, then there is a good chance of its running unchanged on a wide variety of computers. If, on the other hand, the design is implemented by writing a program in assembly language for the Control Data Cyber series then running it on another machine would involve a good deal of effort.

Even when we fix both the design and the implementation language, performance considerations may affect rehostability. For example, consider the use of bit vectors (say as parser director sets or error matrices, or as code generator decision table columns) when the implementation language is Pascal. One possible representation is a set, another is a packed array of Boolean. Unfortunately, some Pascal implementations represent all sets with the same number of bits. This usually precludes large sets, and the bit vectors must be implemented as arrays of sets or packed arrays of Boolean.