Bootstrapping

A compiler is a complex enough program that we would like to write it in a friendlier language than assembly language. In the UNIX programming environment, compilers are usually written in C. Even C compilers are written in C. Using the facilities offered by a language to compile itself is the essence of *bootstrapping*. Here we shall look at the use of bootstrapping to create compilers and to move them from one machine to another by modifying the back end. The basic ideas of bootstrapping have been known since the mid 1950's (Strong et al. [1958]).

Bootstrapping may raise the question, "How was the first compiler compiled?" which sounds like, "What came first, the chicken or the egg?" but is easier to answer. For an answer we consider how Lisp became a programming language. McCarthy [1981] notes that in late 1958 Lisp was used as a notation for writing functions; they were then hand-translated into assembly language and run. The implementation of an interpreter for Lisp occurred unexpectedly. McCarthy wanted to show that Lisp was a notation for describing functions "much neater than Turing machines or the general recursive definitions used in recursive function theory," so he wrote a function *eval*[*e*, *a*] in Lisp that took a Lisp expression *e* as an argument. S. R. Russell noticed that *eval* could serve as an interpreter for Lisp, hand-coded it, and thus created a programming language with an interpreter. As mentioned in Section 1.1, rather than generating target code, an interpreter actually performs the operations of the source program.

For bootstrapping purposes, a compiler is characterized by three languages: the source language S that it compiles, the target language T that it generates code for, and the implementation language I that it is written in. We represent the three languages using the following diagram, called a *T-diagram*, because of its shape (Bratman [1961]).

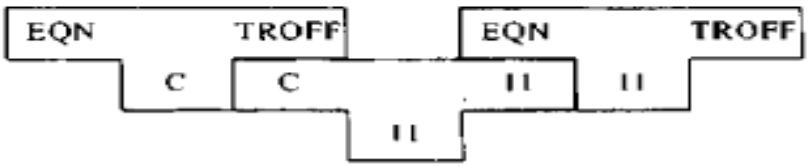
Within text, we abbreviate the above T-diagram as $S_I T$. The three languages S, I, and T may all be quite different. For example, a compiler may run on one machine and produce target code for another machine. Such a compiler is often called a *cross-compiler*.

Suppose we write a cross-compiler for a new language L in implementation language S to generate code for machine N; that is, we create $L_S N$. If an existing compiler for S runs on machine M and generates code for M, it is characterized by $S_M M$. If $L_S N$ is run through $S_M M$, we get a compiler $L_M N$, that is, a compiler from L to N that runs on M. This process is illustrated in Fig. 11.1 by putting together the T-diagrams for these compilers.

When T-diagrams are put together as in Fig. 11.1, note that the implementation language S of the compiler $L_S N$ must be the same as the source language of the existing compiler $S_M M$ and that the target language M of the existing compiler must be that same as the implementation language of the translated form $L_M N$. A trio of T-diagrams such as Fig. 11.1 can be thought of as an equation

$$L_S N + S_M M = L_M N$$

Example 11.1. The first version of the EQN compiler (see Section 12.1) had C as the implementation language and generated commands for the text formatter TROFF. As shown in the following diagram, a cross-compiler for EQN, running on a PDP-11, was obtained by running EQN $_C$TROFF through the C compiler C_{11} 11 on the PDP-11.

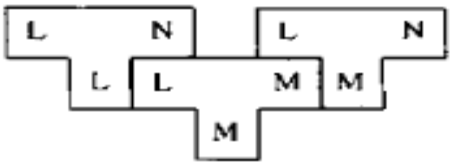


□

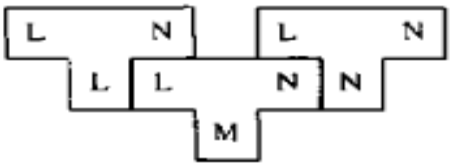
One form of bootstrapping builds up a compiler for larger and larger subsets of a language. Suppose a new language L is to be implemented on machine M. As a first step we might write a small compiler that translates a subset S of L into the target code for M; that is, a compiler $S_M M$. We then use the subset S to write a compiler $L_S M$ for L. When $L_S M$ is run through $S_M M$, we obtain an implementation of L, namely, $L_M M$. Neliac was one of the first languages to be implemented in its own language (Huskey, Halstead, and McArthur [1960]).

Wirth [1971] notes that Pascal was first implemented by writing a compiler in Pascal itself. The compiler was then translated "by hand" into an available low-level language without any attempt at optimization. The compiler was for a subset "(>60 per cent)" of Pascal; several bootstrapping stages later a compiler for all of Pascal was obtained. Lecarme and Peyrolle-Thomas [1978] summarize methods that have been used to bootstrap Pascal compilers.

For the advantages of bootstrapping to be realized fully, a compiler has to be written in the language it compiles. Suppose we write a compiler $L_L N$ for language L in L to generate code for machine N. Development takes place on a machine M, where an existing compiler $L_M M$ for L runs and generates code for M. By first compiling $L_L N$ with $L_M M$, we obtain a cross-compiler $L_M N$ that runs on M, but produces code for N:



The compiler $L_L N$ can be compiled a second time, this time using the generated cross-compiler:



The result of the second compilation is a compiler $L_N N$ that runs on N and generates code for N. The are a number of useful applications of this two-step process, so we shall write it as in Fig. 11.2.

Example 11.2. This example is motivated by the development of the Fortran H compiler (see Section 12.4). "The compiler was itself written in Fortran and bootstrapped three times. The first time was to convert from running on the IBM 7094 to System/360 — an arduous procedure. The second time was to optimize itself, which reduced the size of the compiler from about 550K to about 400K bytes" (Lowry and Medlock [1969]).

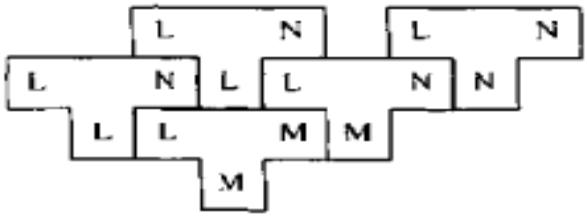
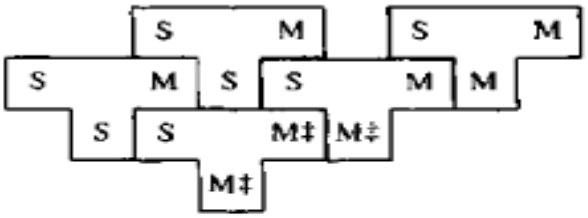


Fig. 11.2. Bootstrapping a compiler.

Using bootstrapping techniques, an optimizing compiler can optimize itself. Suppose all development is done on machine M. We have $S_S M$, a good optimizing compiler for a language S written in S, and we want $S_M M$, a good optimizing compiler for S written in M.

We can create $S_{M‡} M‡$, a quick-and-dirty compiler for S on M that not only generates poor code, but also takes a long time to do so. (M‡ indicates a poor implementation in M. $S_{M‡} M‡$ is a poor implementation of a compiler that generates poor code.) However, we can use the indifferent compiler $S_{M‡} M‡$ to obtain a good compiler for S in two steps:



First, the optimizing compiler $S_S M$ is translated by the quick-and-dirty compiler to produce $S_{M‡}M$, a poor implementation of the optimizing compiler, but one that does produce good code. The good optimizing compiler $S_M M$ is obtained by recompiling $S_S M$ through $S_{M‡}M$. □