3 COMPILER CONSTRUCTION AND BOOTSTRAPPING

By now the reader may have realized that developing translators is a decidedly non-trivial exercise. If one is faced with the task of writing a full-blown translator for a fairly complex source language, or an emulator for a new virtual machine, or an interpreter for a low-level intermediate language, one would probably prefer not to implement it all in machine code.

Fortunately one rarely has to contemplate such a radical step. Translator systems are now widely available and well understood. A fairly obvious strategy when a translator is required for an old language on a new machine, or a new language on an old machine (or even a new language on a new machine), is to make use of existing compilers on either machine, and to do the development in a high level language. This chapter provides a few examples that should make this clearer.

3.1 Using a high-level host language

If, as is increasingly common, one's dream machine *M* is supplied with the machine coded version of a compiler for a well-established language like C, then the production of a compiler for one's dream language *X* is achievable by writing the new compiler, say *XtoM*, in C and compiling the source (*XtoM.C*) with the C compiler (*CtoM.M*) running directly on *M* (see Figure 3.1). This produces the object version (*XtoM.M*) which can then be executed on *M*.

```
.--------------------------.          .--------------------------.
|           XtoM.C         |          |           XtoM.M         |
| X      ----------> M-Code|          | X     -----------> M-code|
|                          |          |                          |
`-------.          .--------------------------.          .-------'
        |          |          CtoM.M          |          |
        |     C    | C      --------->  M-code |  M-code  |
        |          |                          |          |
        `------------------.          .------------------'
                           |          |       .----------.
                           |  M-code  |       |M-machine |
                           |          |       `----------'
                           `----------'
                           .----------.
                           |M-machine |
                           `----------'
```

Figure 3.1 Use of C as an implementation language

Even though development in C is much easier than development in machine code, the process is still complex. As was mentioned earlier, it may be possible to develop a large part of the compiler source using compiler generator tools - assuming, of course, that these are already available either in executable form, or as C source that can itself be compiled easily. The hardest part of the development is probably that associated with the back end, since this is intensely machine dependent. If one has access to the source code of a compiler like *CtoM* one may be able to use this to good avail. Although commercial compilers are rarely released in source form, source code is available for many compilers produced at academic institutions or as components of the GNU project carried out under the auspices of the Free Software Foundation.

3.2 Porting a high level translator

The process of modifying an existing compiler to work on a new machine is often known as **porting** the compiler. In some cases this process may be almost trivially easy. Consider, for example, the fairly common scenario where a compiler *XtoC* for a popular language *X* has been implemented in C on machine *A* by writing a high-level translator to convert programs written in *X* to C, and where it is desired to use language *X* on a machine *M* that, like *A*, has already been blessed with a C compiler of its own. To construct a two-stage compiler for use on either machine, all one needs to do, in principle, is to install the source code for *XtoC* on machine *M* and recompile it.

Such an operation is conveniently represented in terms of T-diagrams chained together. Figure 3.2(a) shows the compilation of the *X* to C compiler, and Figure 3.2(b) shows the two-stage compilation process needed to compile programs written in *X* to *M*-code.

```
  (a)  .--------------------------.          .--------------------------.
       |           XtoC.C         |          |           XtoC.M         |
       | X      ---------->     C |          | X     ----------->     C |
       |                          |          |                          |
       `-------.          .--------------------------.          .-------'
               |          |          CtoM.M          |          |
               |     C    | C      --------->  M-code |  M-code  |
               |          |                          |          |
               `------------------.          .------------------'
                                  |          |
                                  |  M-code  |
                                  |          |
                                  `----------'

  (b)  .--------------------------.          .--------------------------.
       |           XtoC.M         |          |           CtoM.M         |
PROG.X | X     ----------->     C |  PROG.C  | C     ----------> M-code | PROG.M
       |                          |          |                          |
       `-------.          .-------'          `-------.          .-------'
               |          |                          |          |
               |  M-code  |                          |  M-code  |
               |          |                          |          |
               `----------'                          `----------'
               .----------.                          .----------.
               |M-machine |                          |M-machine |
               `----------'                          `----------'
```

Figure 3.2 Porting and using a high-level translator

The portability of a compiler like *XtoC.C* is almost guaranteed, provided that it is itself written in "portable" C. Unfortunately, or as Mr. Murphy would put it, "interchangeable parts don't" (more explicitly, "portable C isn't"). Some time may have to be spent in modifying the source code of *XtoC.C* before it is acceptable as input to *CtoM.M*, although it is to be hoped that the developers of *XtoC.C* will have used only standard C in their work, and used pre-processor directives that allow for easy adaptation to other systems.

If there is an initial strong motivation for making a compiler portable to other systems it is, indeed, often written so as to produce high-level code as output. More often, of course, the original implementation of a language is written as a self-resident translator with the aim of directly producing machine code for the current host system.

3.3 Bootstrapping

All this may seem to be skirting around a really nasty issue - how might the first high-level language have been implemented? In ASSEMBLER? But then how was the assembler for ASSEMBLER produced?

A full assembler is itself a major piece of software, albeit rather simple when compared with a compiler for a really high level language, as we shall see. It is, however, quite common to define one language as a subset of another, so that subset 1 is contained in subset 2 which in turn is contained in subset 3 and so on, that is:

```
Subset 1    ⊆       Subset 2    ⊆       Subset 3
of ASSEMBLER        of ASSEMBLER        of ASSEMBLER
```

One might first write an assembler for subset 1 of ASSEMBLER in machine code, perhaps on a load-and-go basis (more likely one writes in ASSEMBLER, and then hand translates it into machine code). This subset assembler program might, perhaps, do very little other than convert mnemonic opcodes into binary form. One might then write an assembler for subset 2 of ASSEMBLER in subset 1 of ASSEMBLER, and so on.

This process, by which a simple language is used to translate a more complicated program, which in turn may handle an even more complicated program and so on, is known as **bootstrapping**, by analogy with the idea that it might be possible to lift oneself off the ground by tugging at one's boot-straps.

3.4 Self-compiling compilers

Once one has a working system, one can start using it to improve itself. Many compilers for popular languages were first written in another implementation language, as implied in section 3.1, and then rewritten in their own source language. The rewrite gives source for a compiler that can then be compiled with the compiler written in the original implementation language. This is illustrated in Figure 3.3.

```
.---------------------------.          .---------------------------.
|       PasToM.2.Pas        |          |        PasToM.2.M         |
| Pascal ---------> M-code  |          | Pascal ---------> M-code  |
|                           |          |                           |
`-------.          .---------------------------.          .-------'
        |          |       PasToM.1.Fort        |          |
        |  Pascal  | Pascal  ------->  M-code   |  M-code  |
        |          |                            |          |
        `------------------.          .------------------'
                           |          |
                           | Fortran  |
                           |          |
                           `----------'
```

Figure 3.3 First step in developing a self-compiling compiler

Clearly, writing a compiler by hand not once, but twice, is a non-trivial operation, unless the original implementation language is close to the source language. This is not uncommon: Oberon compilers could be implemented in Modula-2; Modula-2 compilers, in turn, were first implemented in Pascal (all three are fairly similar), and C++ compilers were first implemented in C.

Developing a self-compiling compiler has four distinct points to recommend it. Firstly, it constitutes a non-trivial test of the viability of the language being compiled. Secondly, once it has been done, further development can be done without recourse to other translator systems. Thirdly, any improvements that can be made to its back end manifest themselves both as improvements to the object code it produces for general programs and as improvements to the compiler itself. Lastly, it provides a fairly exhaustive self-consistency check, for if the compiler is used to compile its own source code, it should, of course, be able to reproduce its own object code (see Figure 3.4).

```
.--------------------------.          .--------------------------.
|       PasToM.2.Pas       |          |        PasToM.2.M        |
| Pascal ---------> M-code |          | Pascal  --------> M-code |
|                          |          |                          |
`-------.          .--------------------------.          .-------'
        |          |        PasToM.2.M        |          |
        |  Pascal  | Pascal  --------> M-code |  M-code  |
        |          |                          |          |
        `------------------.          .------------------'

                           |          |                ^
                           |  M-code  |    should be   |
                           |          |<-- identical --'
                           `----------'
```

Figure 3.4 A self-compiling compiler must be self-consistent

Furthermore, given a working compiler for a high-level language it is then very easy to produce compilers for specialized dialects of that language.

3.5 The half bootstrap

Compilers written to produce object code for a particular machine are not intrinsically portable. However, they are often used to assist in a porting operation. For example, by the time that the first Pascal compiler was required for ICL machines, the Pascal compiler available in Zürich (where Pascal had first been implemented on CDC mainframes) existed in two forms (Figure 3.5).

```
.--------------------------.             .--------------------------.
|       PasToCDC.Pas       |             |       PasToCDC.CDC       |
| Pascal -------> CDC-code |             | Pascal  ------> CDC-code |
|                          |             |                          |
`-------.          .-------'             `-------.          .-------'
        |          |                             |          |
        |  Pascal  |                             | CDC-code |
        |          |                             |          |
        `----------'                             `----------'
```

Figure 3.5 Two versions of the original Zürich Pascal compiler

The first stage of the transportation process involved changing *PasToCDC.Pas* to generate ICL machine code - thus producing a cross compiler. Since *PasToCDC.Pas* had been written in a high-level language, this was not too difficult to do, and resulted in the compiler *PasToICL.Pas*.

Of course this compiler could not yet run on any machine at all. It was first compiled using *PasToCDC.CDC*, on the CDC machine (see Figure 3.6(a)). This gave a cross-compiler that could run on CDC machines, but still not, of course, on ICL machines. One further compilation of *PasToICL.Pas*, using the cross-compiler *PasToICL.CDC* on the CDC machine, produced the final result, *PasToICL.ICL* (Figure 3.6(b)).

```
(a)  .--------------------------.          .--------------------------.
     |       PasToICL.Pas       |          |       PasToICL.CDC       |
     | Pascal  ------> ICL-code |          | Pascal  ------> ICL-code |
     |                          |          |                          |
     `-------.          .--------------------------.          .-------'
             |          |       PasToCDC.CDC       |          |
             |  Pascal  | Pascal  ------> CDC-code | CDC-code |
             |          |                          |          |
             `------------------.          .------------------'
                                |          |
                                | CDC-code |
                                |          |
                                `----------'

(b)  .--------------------------.          .--------------------------.
     |       PasToICL.Pas       |          |       PasToICL.ICL       |
     | Pascal  ------> ICL-code |          | Pascal -------> ICL-code |
     |                          |          |                          |
     `-------.          .--------------------------.          .-------'
             |          |       PasToICL.CDC       |          |
             |  Pascal  | Pascal  ------> ICL-code | ICL-code |
             |          |                          |          |
             `------------------.          .------------------'
                                |          |
                                | CDC-code |
                                |          |
                                `----------'
```

Figure 3.6 The production of the first ICL Pascal compiler by half bootstrap

The final product (*PasToICL.ICL*) was then transported on magnetic tape to the ICL machine, and loaded quite easily. Having obtained a working system, the ICL team could (and did) continue development of the system in Pascal itself.

This porting operation was an example of what is known as a *half bootstrap* system. The work of transportation is essentially done entirely on the donor machine, without the need for any translator in the target machine, but a crucial part of the original compiler (the back end, or code generator) has to be rewritten in the process. Clearly the method is hazardous - any flaws or oversights in writing *PasToICL.Pas* could have spelled disaster. Such problems can be reduced by minimizing changes made to the original compiler. Another technique is to write an emulator for the target machine that runs on the donor machine, so that the final compiler can be tested on the donor machine before being transferred to the target machine.

3.6 Bootstrapping from a portable interpretive compiler

Because of the inherent difficulty of the half bootstrap for porting compilers, a variation on the full bootstrap method described above for assemblers has often been successfully used in the case of Pascal and other similar high-level languages. Here most of the development takes place on the target machine, after a lot of preliminary work has been done on the donor machine to produce an interpretive compiler that is almost portable. It will be helpful to illustrate with the well-known example of the Pascal-P implementation kit mentioned in section 2.5.

Users of this kit typically commenced operations by implementing an interpreter for the P-machine. The bootstrap process was then initiated by developing a compiler (*PasPtoM.PasP*) to translate Pascal-P source programs to the local machine code. This compiler could be written in Pascal-P source, development being guided by the source of the Pascal-P to P-code compiler supplied as part of the kit. This new compiler was then compiled with the interpretive compiler (*PasPtoP.P*) from the kit (Figure 3.7(a)) and the source of the Pascal to M-code compiler was then compiled by this new compiler, interpreted once again by the P-machine, to give the final product, *PasPtoM.M* (Figure 3.7(b)).

```
(a) .--------------------------.          .--------------------------.
    |      PasPtoM.PasP        |          |        PasPtoM.P         |
    | Pascal-P -------> M-code |          | Pascal-P -------> M-code |
    |                          |          |                          |
    `-------.          .--------------------------.          .-------'
            |          |        PasPtoP.P         |          |
            | Pascal-P | Pascal-P  ----->  P-code |  P-code  |
            |          |                          |          |
            `------------------.          .------------------'
                               |          |
                               |  P-code  |
                               |          |
                               `----------'
                               .-----------. --.
                               |  P-code   |   |
                               |Interpreter|   |
                               |  M-code   |   P-Machine
                               `-----------'   |
                               .-----------.   |
                               | M-Machine |   |
                               `-----------' --'

(b) .--------------------------.          .--------------------------.
    |      PasPtoM.PasP        |          |        PasPtoM.M         |
    | Pascal-P -------> M-code |          | Pascal-P -------> M-code |
    |                          |          |                          |
    `-------.          .--------------------------.          .-------'
            |          |        PasPtoM.P         |          |
            | Pascal-P | Pascal-P  ----->  M-code |  M-code  |
            |          |                          |          |
            `------------------.          .------------------'
                               |          |
                               |  P-code  |
                               |          |
                               `----------'
                               .-----------. --.
                               |  P-code   |   |
                               |Interpreter|   |
                               |  M-code   |   P-Machine
                               `-----------'   |
                               .-----------.   |
                               | M-Machine |   |
                               `-----------' --'
```

Figure 3.7 Developing a native code compiler from the P-compiler

The Zürich P-code interpretive compiler could be, and indeed was, used as a highly portable development system. It was employed to remarkable effect in developing the UCSD Pascal system, which was the first serious attempt to implement Pascal on microcomputers. The UCSD Pascal team went on to provide the framework for an entire operating system, editors and other utilities - all written in Pascal, and all compiled into a well-defined P-code object code. Simply by providing an alternative interpreter one could move the whole system to a new microcomputer system virtually unchanged.

3.7 A P-code assembler

There is, of course, yet another way in which a portable interpretive compiler kit might be used. One might commence by writing a P-code to M-code assembler, probably a relatively simple task. Once this has been produced one would have the assembler depicted in Figure 3.8.

```
.--------------------------.
|          PToM.M          |
| P-code ---------> M-code |
|                          |
`-------.          .-------'
        |          |
        |  M-code  |
        |          |
        `----------'
```

Figure 3.8 A P-code to M-code assembler

The P-codes for the P-code compiler would then be assembled by this system to give another cross compiler (Figure 3.9(a)), and the same P-code/M-code assembler could then be used as a back-end to the cross compiler (Figure 3.9(b)).

```
(a) .--------------------------.          .--------------------------.
    |         PasPtoP.P        |          |         PasPtoP.M        |
    | Pascal-P -------> P-code |          | Pascal-P -------> P-code |
    |                          |          |                          |
    `-------.          .--------------------------.          .-------'
            |          |          PtoM.M          |          |
            |  P-code  | P-code ---------> M-code |  M-code  |
            |          |                          |          |
            `------------------.          .------------------'
                               |          |
                               |  M-code  |
                               |          |
                               `----------'

(b) .--------------------------.          .--------------------------.
    |        PasPtoP.M         |          |          PtoM.M          |
    | Pascal-P ------>  P-code |--------->| P-code -------->  M-code |
    |                          |          |                          |
    `-------.          .-------'          `-------.          .-------'
            |          |                          |          |
            |  M-code  |                          |  M-code  |
            |          |                          |          |
            `----------'                          `----------'
```

Figure 3.9 Two-pass compilation and assembly using a P-code compiler

Exercises

3.1 Draw the T-diagram representations for the development of a P-code to M-code assembler, assuming that you have a C++ compiler available on the target system.

3.2 Later in this text we shall develop an interpretive compiler for a small language called Clang, using C++ as the host language. Draw T-diagram representations of the various components of the system as you foresee them.

Further reading

A very clear exposition of bootstrapping is to be found in the book by Watt (1993). The ICL bootstrap is further described by Welsh and Quinn (1972). Other early insights into bootstrapping are to be found in papers by Lecarme and Peyrolle-Thomas (1973), by Nori *et al.* (1981), and Cornelius, Lowman and Robson (1984).