Bootstrapping (compilers)

In computer science, **bootstrapping** is the technique for producing a self-compiling compiler — that is, compiler (or assembler) written in the source programming language that it intends to compile. An initial core version of the compiler (the *bootstrap compiler*) is generated in a different language (which could be assembly language); successive expanded versions of the compiler are developed using this minimal subset of the language.

Many compilers for many programming languages are bootstrapped, including compilers for BASIC, ALGOL, C, C#, D, Pascal, PL/I, Factor, Haskell, Modula-2, Oberon, OCaml, Common Lisp, Scheme, Go, Java, Rust, Python, Scala, Nim, Eiffel, and more.

Contents

Advantages

Bootstrapping a compiler has the following advantages:[1][2]

- it is a non-trivial test of the language being compiled, and as such is a form of dogfooding.
- compiler developers and bug reporting part of the community only need to know the language being compiled.
- compiler development can be performed in the higher-level language being compiled.
- improvements to the compiler's back-end improve not only general-purpose programs but also the compiler itself.
- it is a comprehensive consistency check as it should be able to reproduce its own object code.

Note that some of these points assume that the language runtime is also written in the same language.

The chicken and egg problem

If one needs to compile a compiler for language X (written in language X), there is the issue of how the first compiler can be compiled. The different methods that are used in practice to solving this chicken or the egg problem include:

- Implementing an interpreter or compiler for language X in language Y. Niklaus Wirth reported that he wrote the first Pascal compiler in Fortran.
- Another interpreter or compiler for X has already been written in another language Y; this is how Scheme is often bootstrapped.
- Earlier versions of the compiler were written in a subset of X for which there existed some other compiler; this is how some supersets of Java, Haskell, and the initial Free Pascal compiler are bootstrapped.
- A compiler supporting non-standard language extensions or optional language features can be written without using those extensions and features, to enable it being compiled with another compiler supporting the same base language but a different set of extensions and features. The main parts of the C++ compiler clang were written in a subset of C++ that can be compiled by both g++ and Microsoft Visual C++. Advanced features are written with some GCC extensions.
- The compiler for X is cross compiled from another architecture where there exists a compiler for X; this is how compilers for C are usually ported to other platforms. Also this is the method used for Free Pascal after the initial

bootstrap.
- Writing the compiler in X; then hand-compiling it from source (most likely in a non-optimized way) and running that on the code to get an optimized compiler. Donald Knuth used this for his WEB literate programming system.

Methods for distributing compilers in source code include providing a portable bytecode version of the compiler, so as to *bootstrap* the process of compiling the compiler with itself. The T-diagram is a notation used to explain these compiler bootstrap techniques.[2] In some cases, the most convenient way to get a complicated compiler running on a system that has little or no software on it involves a series of ever more sophisticated assemblers and compilers.[3]

History

Assemblers were the first language tools to bootstrap themselves.

The first high-level language to provide such a bootstrap was NELIAC in 1958. The first widely used languages to do so were Burroughs B5000 Algol in 1961 and LISP in 1962.

Hart and Levin wrote a LISP compiler in LISP at MIT in 1962, testing it inside an existing LISP interpreter. Once they had improved the compiler to the point where it could compile its own source code, it was self-hosting.[4]

> The compiler as it exists on the standard compiler tape is a machine language program that was obtained by having the S-expression definition of the compiler work on itself through the interpreter.
>
> — AI Memo 39[4]

This technique is only possible when an interpreter already exists for the very same language that is to be compiled. It borrows directly from the notion of running a program on itself as input, which is also used in various proofs in theoretical computer science, such as the proof that the halting problem is undecidable.

Current efforts

Due to security concerns regarding the Trusting Trust Attack and various attacks against binary trustworthiness, multiple projects are working to reduce the effort for not only bootstrapping from source but also allowing everyone to verify that source and executable correspond.

These include the Bootstrappable builds project [5] and the Reproducible builds project [6]

List of languages having self-hosting compilers

The following programming languages have self-hosting compilers:

- Ada
- BASIC
- BCPL
- BlitzMax
- Burroughs Algol
- C
- C++ (compilers: Visual C++, clang, probably others)
- C# and Visual Basic .NET via Microsoft Roslyn
- Ciao
- Cobol
- CoffeeScript
- Common Lisp
- Crystal
- Curry

- D
- Delphi
- Eiffel
- F#
- FASM
- Factor
- Forth
- Free Pascal
- Go
- Haskell
- LiveScript
- Java
- Mercury
- Modula-2
- Nemerle
- Nim
- Oberon
- OCaml
- Pascal
- Raku (compilers: Rakudo and Niecza are both self-hosting)
- PL/I
- Python
- Pyret[7]
- Rust
- Scheme
- Scala
- Smalltalk
- SML
- Tcl[8]
- TypeScript
- Virgil[9]
- Umple
- XPL

See also

- Self-hosting
- Self-interpreter
- Tombstone diagram
- Metacompiler

References

1. Compilers and Compiler Generators: An Introduction With C++. Patrick D. Terry 1997. International Thomson Computer Press. ISBN 1-85032-298-8
2. "Compiler Construction and Bootstrapping" by P.D.Terry 2000. HTML (http://www.oopweb.com/Compilers/Documents/Compilers/Volume/cha03s.htm) Archived (https://web.archive.org/web/20091123154911/http://www.oopweb.com/Compilers/Documents/Compilers/Volume/cha03s.htm) 2009-11-23 at the Wayback Machine. PDF (http://webster.cs.ucr.edu/AsmTools/RollYourOwn/CompilerBook/CHAP03.PDF) Archived (https://web.archive.org/web/20101214135219/http://webster.cs.ucr.edu/AsmTools/RollYourOwn/CompilerBook/CHAP03.PDF) December 14, 2010, at the Wayback Machine.
3. "Bootstrapping a simple compiler from nothing" (http://homepage.ntlworld.com/edmund.grimley-evans/bcompiler.html) Archived (https://web.archive.org/web/20100303235322/http://homepage.ntlworld.com/edmund.grimley-evans/bcompiler.html) March 3, 2010, at the Wayback Machine by Edmund GRIMLEY EVANS 2001