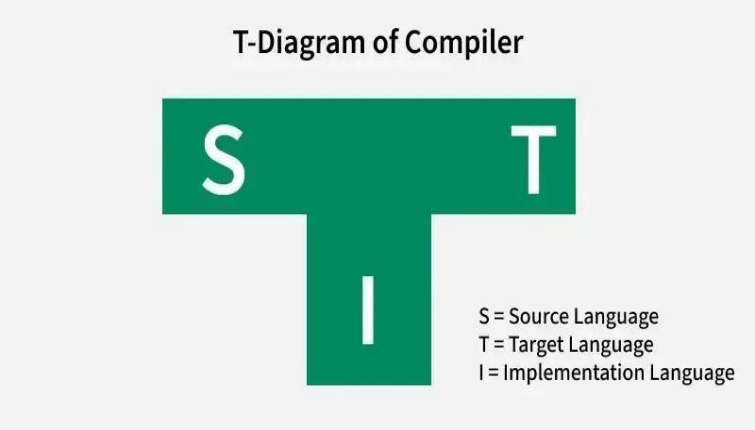
1.2 BOOTSTRAPPING

Bootstrapping is an important technique in compiler design, where a basic compiler is used to create a more advanced version of itself. This process helps in building compilers for new programming languages and improving the ones already in use. By starting with a simple compiler, bootstrapping allows gradual improvements and makes the compiler more efficient over time.

- Bootstrapping relies on the idea of a self-compiling compiler, where each iteration improves the compiler's ability to handle more complex code.
- It simplifies the development cycle, allowing incremental improvements and faster deployment of more robust compilers.
- Many successful programming languages, including C and Java, have used bootstrapping techniques during their development.



A compiler can be represented using a T Diagram. In this diagram, the source language of the compiler is positioned at the top-left, the target language (the language produced by the compiler) is placed at the topright, and the language in which the compiler is implemented is shown at the bottom.

1.2.1 Working of Bootstrapping

Bootstrapping is the process of creating compilers. It involves a methodology where a slightly more complicated compiler is created using a simple language (such as assembly language). This slightly more complicated compiler, in turn, is used to create an even more advanced compiler, and this process continues until the desired result is achieved.

Here's a step-by-step look at how bootstrapping works in compiler design:

Step 1: Start with a Basic Compiler

The first step is to create a basic compiler that can handle the most essential features of a programming language. This simple compiler is often written in assembly language or machine language to make it easier to build.

Step 2: Use the Basic Compiler to Create a More Advanced Version

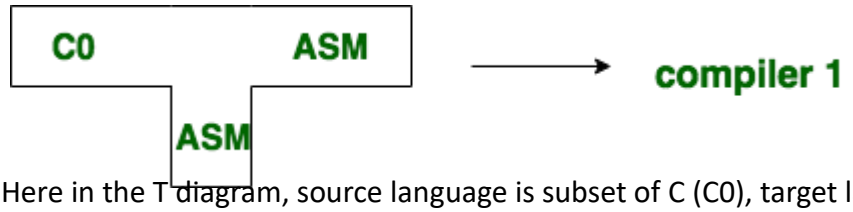
Once the basic compiler is ready, it is used to compile a more advanced version of itself. This new version can handle more complex features, like better error checking and optimizations.

Step 3: Gradually Improve the Compiler

With each new version, the compiler becomes more capable. The process is repeated, and each iteration adds more features, making the compiler stronger and more efficient.

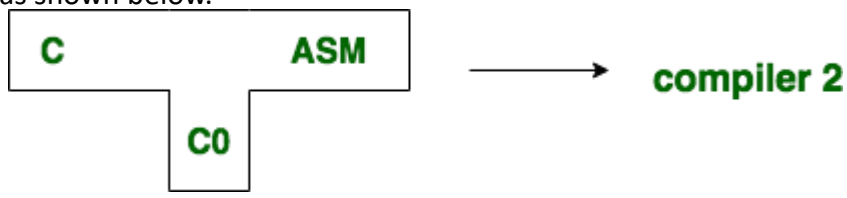
For example, let's assume a compiler which takes C language as input and generates an assembly language as an output.

1. To generate this compiler we first write a compiler for a small subset of C language i.e. C0 in assembly language. Subset of C means C language with less functionality.



Here in the T diagram, source language is subset of C (C0), target language is Assembly language and implementation language is also Assembly language.

2. Then using the C0 language, we create a compiler for the C language. This compiler C0, takes C language as source language and generates assembly language as target language as shown below.



By the help bootstrapping we generated compiler for C language in C language itself i.e. self-compiling compiler.

1.2.2 Cross-Compilation Using Bootstrapping

Cross-compilation is a process where a compiler runs on one platform (host) but generates machine code for a different platform (target). This is useful when the target platform is not powerful enough to run the full compiler or when the target architecture is different from

the host system. Using bootstrapping in crosscompilation can help create a compiler that runs on one system (the host) but produces code for another system (the target).

For example, suppose we want to write a cross compiler for new language X. To create a cross-compiler for a new language X that generates code in language Z, we start by using an existing compiler Y (running on machine M) to compile a simple version of language X into language Z. The first step is to create a basic compiler, XYZ, using Y that translates X code into Z code, all while running on machine M. This results in a cross-compiler XMZ, which is capable of generating target code in language Z for the source code in language X, but it runs on machine M. This method allows us to create a compiler for a new language without needing to run it on the target system directly.

Advantages

1. **Improved Efficiency:** Bootstrapping makes the development process faster. Once you have a basic compiler, you can use it to create more advanced versions, making the entire process of building a complex compiler much quicker. It's like building a tool that helps you build better versions of itself.

2. **Portability:** Bootstrapping helps create compilers that can work across different systems. Once you've bootstrapped a compiler, it can be used to generate code for various platforms, making it more flexible and portable.

3. **Reduced Dependency:** With bootstrapping, you don't need to rely on other compilers or external tools. As long as you have a simple starting compiler, you can use it to build more complex versions and handle all your compiling needs. This reduces the need for other software or external dependencies.

1.2.3 Challenges and Limitations

1. **Initial Effort**: Writing the very first version of the compiler is tough. Before bootstrapping can even begin, you need to create a simple, working compiler from scratch, which requires a lot of time and effort.

2. **Complexity of Self-Compilation**: A compiler that can compile itself sounds great, but it's not easy to make. Building a self-compiling compiler adds complexity because you have to ensure that each version can handle more features and still work as expected.

3. **Time Consumption**: Bootstrapping is not something that happens overnight. It takes time and resources, especially during the early stages. You have to repeatedly build and improve versions of the compiler, which can be a slow process in the beginning.