Bootstrapping in compiler design pdf

What is bootstrapping in compiler. What is bootstrapping in compiler design. Bootstrapping in compiler design in hindi. Bootstrapping in compiler design with example.

You're Reading a Free Preview Pages 6 to 11 are not shown in this preview. It is an approach for making a self-compiling compiler that is a compiler written in the source programming language that it determine to compile. A bootstrap compiler can compile the compiler and thus you can use this compiled compiler to compile everything else and the future versions of itself.Uses of BootstrappingThere are various uses of bootstrapping which are as follows −It can allow new programming languages and compilers to be developed starting from actual ones.It allows new features to be combined with a programming language and its compiler.It also allows new optimizations to be added to compilers.It allows languages and compilers to be transferred between processors with different instruction setsAdvantages of BootstrappingThere are various advantages of bootstrapping which are as follows −Compiler development can be performed in the higher-level language being compiled.It is a non-trivial test of the language being compiled.It is an inclusive consistency check as it must be capable of recreating its object code.For bootstrapping, a compiler is defined by three languages −S→Source language it compiles.T→Target language it generates.I→Implementation language that it is written inThese languages can be represented using a T-diagram asCross CompilerA compiler is characterized by three languages as its source language, its object language, and the language in which it is written. These languages may be quite different. A compiler can run on one machine and produce target code for another machine.

Such a compiler is known as a cross-compiler.If it can write a cross-compiler for a new language 'L' in execution languages 'S' to generate a program for machine 'N'.i.e. LsNIf a current compiler for S runs on machine M and generates a program for M, it is defined by SMM.If LSN runs through SMM, we get a compiler LMN, i.e., a compiler from L to N that runs on M.Example − Create a cross compiler using bootstrapping when SsM runs on SAA.Solution − First of all, it represents two compilers with T-diagram.When SsM runs on SAA, SAM will be generated. Bootstrapping in Compiler Design/Construction [Term Paper/Journal] Submitted in Fulfillment of the Requirement for the Completion of COMPILER DESIGN – CSE415 by Shabnam Sidhu (B.Tech.CSE) [email] : shabnam719[at]gmail[dot]com RK2202B29-11203047 School of Computer Science and Engineering Under the Guidance of Asst. Prof. Harshpreet Singh Department of Computer Science Lovely Professional University Punjab-India Submission Date : 7th April, 2015 Page No.: Bootstrapping in Compiler Design/Construction by Shabnam Sidhu School of Computer Science and Engineering Lovely Professional University, IN This paper represents my own work and is formatted based upon “Standard IEEE Format for Research Journals” in accordance with University regulations. /s/. Shabnam Sidhu Contents 1.
 Page No.: 0 Chapter - 1 Introduction: Bootstrapping is the process of writing a compiler (or assembler) in the target programming language which it is intended to compile. Applying this technique leads to a self-hosting compiler. Introduction To Compiler: A compiler is set of programs that transforms or converts, source code written in a programming language to system language. System language is basically the object code which is basically in binary form. Main objective of this is to create an executable program. A compiler is likely to perform many or all of the following operations: lexical analysis, preprocessing, parsing, semantic analysis (Syntax-directed translation), code generation, and code optimization. Compilation: Compilers enabled the development of programs that are machine-independent. The first higher-level language, in the 1950s, machine-dependent assembly language was widely used. While assembly language produces more reusable and relocatable programs than machine code on the same architecture, it has to be modified or rewritten if the program is to be executed on different computer hardware architecture. Structure: A compiler consists of three main parts: the front-end, the middle-end, and the back-end. Front End: Programs are checked here in term of Syntax and Semantics of respective programming language. Type checking is also performed by collecting type information. The front-end then generates an intermediate representation of the source code for processing by the middle-end. Middle End: Here optimization takes place. Few transformations for optimization are removal of useless or unreachable code, discovery and relocation of computation code etc.

Bootstrapping in Compiler Design/Construction

[Term Paper/Journal]

Submitted in Fulfillment of the
Requirement for the Completion of

COMPILER DESIGN – CSE415

by

Shabnam Sidhu *(B.Tech.CSE)*
[email] : shabnam719[at]gmail[dot]com

RK2202B29-11203047
School of Computer Science and Engineering

Under the Guidance of

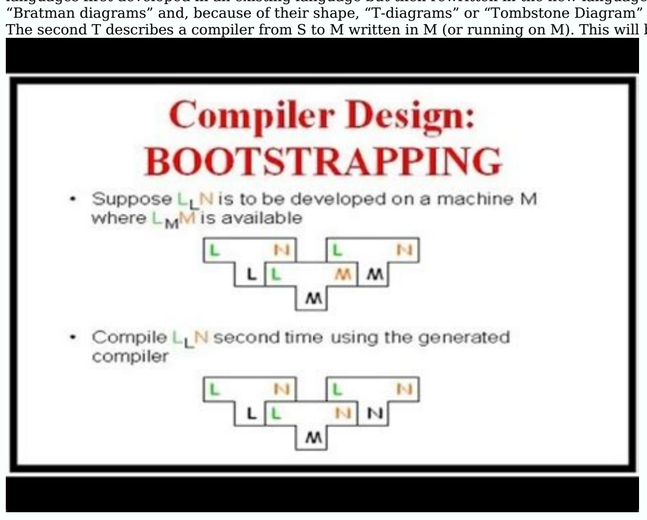
Asst. Prof. Harshpreet Singh

Department of Computer Science
Lovely Professional University
Punjab-India

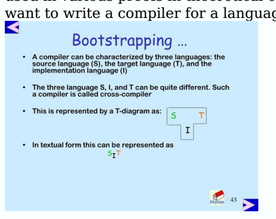
Submission Date : 7th April, 2015

Page No.:

Page No.: 1 Back End: It gives the output as assembly code from optimized code from middle-end. Memory allocation and register allocation are also performed here for process register (required for some program variables). Page No.: 2 Chapter – 2 Structure of Compiler: In a compiler, • linear analysis • is called LEXICAL ANALYSIS or SCANNING and • is performed by the LEXICAL ANALYZER or LEXER, • hierarchical analysis • is called SYNTAX ANALYSIS or PARSING and • is performed by the SYNTAX ANALYZER or PARSER. • During the analysis, the compiler manages a SYMBOL TABLE by • recording the identifiers of the source program • collecting information (called ATTRIBUTES) about them: storage allocation, type, scope, and (for functions) signature. • When the identifier x is found by the lexical analyzer • generates the token id • enters the lexeme x in the symbol-table (if it is not already there) • associates to the generated token a pointer to the symbol-table entry x.
This pointer is called the LEXICAL VALUE of the token. • During the analysis or synthesis, the compiler may DETECT ERRORS and report on them. • However, after detecting an error, the compilation should proceed allowing further errors to be detected. • The syntax and semantic phases usually handle a large fraction of the errors detectable by the compiler. Page No.: 3 Chapter – 3 Compiler Construction and Bootstrapping: All but the smallest of compilers have more than two phases. However, these phases are usually regarded as being part of the front end or the back end. The point at which these two ends meet is open to debate. The development of compilers for new programming languages first developed in an existing language but then rewritten in the new language and compiled by itself, is another example of the bootstrapping notion. Definition: The notion of implementing a language, by means of using (part of) that language’s own implementation. Notation: T-diagram T-Diagram: The Bootstrapping notation is also called “Bratman diagrams” and, because of their shape, “T-diagrams” or “Tombstone Diagram” which is used to represent complex diagrams. Understanding T-Diagrams : The first T describes a compiler from L to N written in S.
The second T describes a compiler from S to M written in M (or running on M). This will be your compiler compiler. Applying the second T to the first T compiles the first T so that it runs on machine M. The result is thus a compiler from L to N running on machine M.



Page No.: 4 Chapter – 4 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. 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. Bootstrapping Concept: The 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. In other words, you want to write a compiler for a language A, targeting language B (the machine language) and written in language B.



The most obvious approach is to write the compiler in language B.

But if B is machine language, it is a horrible job to write any non-trivial compiler in this language. Instead, it is customary to use a process called “bootstrapping”, referring to the seemingly impossible task of pulling oneself up by the bootstraps. Use of Bootstrapping: As mentioned, bootstrapping means that a compiler can compile itself. What are the pros and cons of bootstrap-ping?
Page No.: 5 The implemented language becomes well tested, since the developers are using it on a large application (the compiler).
 The developers are motivated to make a high quality implementation, since they are using it themselves. The developers are motivated to create a good development environment, since they are using it themselves. There is also a negative factor: since the tool must be able to build itself, there is more work to do significant changes to it – the implementation must be good enough to be useable. Page No.: 6 Chapter - 5 Compiling Compilers: The basic idea in bootstrapping is to use compilers to compile themselves or other compilers. We do, however, need a solid foundation in form of a machine to run the compilers on. Developing a self-compiling compiler has four distinct points to recommend it: 1: It constitutes a non-trivial test of the viability of the language being compiled.
2: Once it has been done, further development can be done without recourse to other translator systems. 3: 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. 4: 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 Full Bootstrap: The bootstrapping process relies on an existing compiler for the desired language, albeit running on a different machine. It is, hence, often called “half bootstrapping”. When no existing compiler is available, e.g.: When a new language has been designed, we need to use a more complicated process called “full bootstrapping”. - A common method is to write a QAD (“quick and dirty”) compiler using an existing language. This compiler needs not generate code for the desired target machine (as long as the generated code can be made to run on some existing platform), nor does it have to generate good code. The important thing is thatit allows programs in the new language to be executed. Additionally, the “real” compiler is written in the new language and will be bootstrapped using the QAD compiler Page No.: 7 Half Bootstrap: We discussed full bootstrapping which is required when we have no access to a compiler for our language at all. If we have access to a compiler for our language on a different machine HM but want to develop one for TM, we'll use Half Bootstrap Incremental Bootstrap: It is also possible to build the new language and its compiler incrementally. The first step is to write a compiler for a small subset of the language, using that same subset to write it. This first compiler must be bootstrapped in one of the ways described earlier, but thereafter the following process is done repeatedly: 1) Extend the language subset slightly. 2) Extend the compiler so it compiles the extended subset, but without using the new features. 3) Use the previous compiler to compile the new. In each step, the features introduced in the previous step can be used in the compiler. Even when the full language is compiled, the process can be continued to improve the quality of the compiler. Problem and Solution: 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 rules. 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. Advantages: Bootstrapping a compiler has the following advantages: • it is a non-trivial test of the language being compiled. Page No.: 8 • compiler developers only need to know the language

being compiled. • compiler development can be done 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.
Page No.: 9 REFERENCES [1]. 28compilers%29 [2]. [3]. [4]. [5].
[6]. Page No.: 10