# A Survey of Compilers

**HOME ASSIGNMENT(SURVEY)**

# Subject : Compiler Design

Guide :Prof. Gauri Krishna Barse

**GROUP DETAILS : Group – 7**

|  |  |  |
| --- | --- | --- |
| NAME | Roll Number | GR number |
| Atharva Dixit | C-38 | 1710321 |
| Aditya Holkar | C-57 | 1710349 |
| Upendra Kadre | C-69 | 1710505 |
| Swapnil Katti | C-72 | 1710322 |

A Survey of Compiler

INTRODUCTION

Compilers are important tools because they are a central piece of infrastructure for building other software. Virtually every program that runs on a computer, ranging from operating systems over web browsers to small scripts written by end-users, has been processed by a compiler or a compiler-like tool. Because compilers are such a central piece of infrastructure, they are very widely distributed. For example, popular compilers of widespread programming languages, such as GCC for C/C++, are run by millions of users. Beyond these direct users of compilers, typically developers, much more people indirectly rely on compilers when executing compiled programs.

Given the importance of compilers, there is a huge interest in implementing them correctly. However, reaching this goal is non-trivial since compilers are complex pieces of software. A typical compiler consists of a pipeline of interacting components that address, e.g., the lexical analysis of the source language, parsing the source language into an intermediate representation, applying semantic checks, optimizing the code, and generating code in the target language. Currently, the optimization phase and implementation of parallelism and object-oriented features make it more complex. Because of this complexity, traditional computer science curricula typically dedicate at least one entire course to the art of constructing compilers. In contrast to other complex software, the domain that compilers deal with is particularly rich. Both the input and the output of compilers usually are programs written in Turing-complete languages. Moreover, both the inputs and output domains are infinitely large, since programs can be arbitrarily long. As a result, reasoning about the behaviour of a compiler is all but trivial.

# History of compilers

Software for early computers was primarily written in assembly language. It is usually more productive for a programmer to use a high-level language, and programs written in a high-level language can be reused on different kinds of computers. Even so, it took a while for compilers to become established, because they generated code that did not perform as well as hand-written assembler, they were daunting development projects in their own right, and the very limited memory capacity of early computers created many technical problems for practical compiler implementations.

### First Compiler

The first practical compiler was written by Corrado Bohm, in 1951. And the first implemented compiler was written by Grace Hopper, who also coined the term "compiler", [1] [2] referring to her A-0 system which functioned as a loader or linker, not the modern notion of a compiler. The first Autocode and compiler in the modern sense were developed by Alick Glennie in 1952 at the University of Manchester for the Mark 1 computer.[3][4] The FORTRAN team led by John W. Backus at IBM introduced the first commercially available compiler, in 1957, which took 18 person-years to create.[5]

### Self-hosting Compilers

Like any other software, there are benefits from implementing a compiler in a high-level language. In particular, a compiler can be self-hosted – that is, written in the programming language it compiles (NOT COMPILER COMPILER). Building a self-hosting compiler is a bootstrapping problem, i.e. the first such compiler for a language must be either hand written machine code or compiled by a compiler written in another language, or compiled by running the compiler in an interpreter.

Some of the example are as follow:

* Corrado Böhm PhD dissertation

First compiler to be define in its own language. The language was interesting in itself, because every statement was a special case of an assignment statement (including input and output).

* NELIAC

The Navy Electronics Laboratory International ALGOL Compiler or NELIAC was a dialect and compiler implementation of the ALGOL 58 programming language developed by the Naval Electronics Laboratory in 1958. It was the world's first self-compiling compiler - the compiler was first coded in simplified form in assembly language (the bootstrap), then re-written in its own language and compiled by the bootstrap, and finally re-compiled by itself, making the bootstrap obsolete.

* Lisp

Another early self-hosting compiler was written for Lisp by Tim Hart and Mike Levin at MIT in 1962. [6] 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.

* Forth

It is the extensible programming language features of Forth and Lisp that enable them to generate new versions of themselves or port themselves to new environments.

#### CFG & Parsers

A parser is an important component of a compiler. It parses the source code of a computer programming language to create some form of internal representation. A context-free grammar provides a simple and precise mechanism for describing how programming language constructs are built from smaller blocks.

If a programming language designer is willing to work within some limited subsets of context-free grammars, more efficient parsers are possible. Mainly there are two parser

* LR parser

An LR parser is a parser that reads input from Left to right (as it would appear if visually displayed) and produces a Rightmost derivation. The term LR(k) parser is also used, where k refers to the number of unconsumed lookahead input symbols that are used in making parsing decisions.

Knuth proved that LR(k) grammars can be parsed with an execution time essentially proportional to the length of the program, and that every LR(k) grammar for k > 1 can be mechanically transformed into an LR(1) grammar for the same language. In other words, it is only necessary to have one symbol lookahead to parse any deterministic context-free grammar (DCFG). [7]

* LL parser

The class of grammars which are parsable in this way is known as the LL grammars. LL grammars are an even more restricted class of context-free grammars than LR grammars. Nevertheless, they are of great interest to compiler writers, because such a parser is simple and efficient to implement.

LL(k) grammars can be parsed by a recursive descent parser which is usually coded by hand, although a notation such as META II might alternatively be used. LR parsing can handle a larger range of languages than LL parsing, and is also better at error reporting, i.e. it detects syntactic errors when the input does not conform to the grammar as soon as possible.

##### MetaCompilers

Metacompilers differ from parser generators, taking as input a program written in a metalanguage. Their input consists grammar analysing formula and transform operations that produce abstract syntax trees, or simply output reformatted text strings that may be stack machine code.

Many can be programmed in their own metalanguage enabling them to compile themselves, making them self-hosting extensible language compilers.

Mainly there are three Metacompiler:

* META II

It was made by Dewey Val Schorre , first released in 1964. It was the documented metacompiler. Able to define its own and other language. Lexical analysis was performed by built token recognizing functions: .ID, .STRING, and .NUMBER. Quoted strings in syntax formula recognize lexemes that are not kept. [8]

* TREE-META

TREE-META, a second generation Schorre metacompiler, appeared around 1968. It extended the capabilities of META II, adding unparse rules separating code production from the grammar analysis. Tree transform operations in the syntax formula produce abstract syntax trees that the unparse rules operate on. The unparse tree pattern matching provided peephole optimization ability.

* CWIC

CWIC, described in a 1970 ACM publication is a third generation Schorre metacompiler that added lexing rules and backtracking operators to the grammar analysis. LISP 2 was married with the unparse rules of TREEMETA in the CWIC generator language. With LISP 2 processing, CWIC can generate fully optimized code.

CWIC compiled to 8 bit byte addressable machine code instructions primarily designed to produce IBM System/360 code.

Later generations are not publicly documented. One important feature would be the abstraction of the target processor instruction set, generating to a pseudo machine instruction set, macros, that could be separately defined or mapped to a real machine's instructions.

Optimizations applying to sequential instructions could then be applied to the pseudo instruction before their expansion to target machine code.

###### Compiler Optimization

Compiler optimization is the process of improving the quality of object code without changing the results it produces.

The developers of the first FORTRAN compiler aimed to generate code that was better than the average hand-coded assembler, so that customers would actually use their product. In one of the first real compilers, they often succeeded. [9]

Mainly two optimizer are used to optimize runtime:

* Peephole Optimization

Peephole optimization is a very simple but effective optimization technique. It was invented by William M. McKeeman and published in 1965 in CACM.[10] It was used in the XPL compiler that McKeeman helped develop.

* Capex COBOL Optimizer

Capex Corporation developed the "COBOL Optimizer" in the mid 1970s for COBOL. This type of optimizer depended, in this case, upon knowledge of 'weaknesses' in the standard IBM COBOL compiler, and actually replaced (or patched) sections of the object code with more efficient code.

For example, on the IBM System/360 hardware the CLI instruction was, depending on the particular model, between twice and 5 times as fast as a CLC instruction for single byte comparisons.[11][12]

Modern compilers typically provide optimization options, so programmers can choose whether or not to execute an optimization pass.

Factors Affecting Optimization

* The architecture of the target CPU
* The architecture of the machine
* Intended use of generated code
  + Debugging
  + Special purpose use
  + General Purpose use
  + Embedded system

Reference

[1] [Maurice V. Wilkes](https://en.wikipedia.org/wiki/Maurice_V._Wilkes). 1968. Computers Then and Now. Journal of the Association for Computing Machinery, 15(1):1–7, January. p. 3 (a comment in brackets added by editor), "(I do not think that the term compiler was then [1953] in general use, although it had in fact been introduced by Grace Hopper.)"

[2]The World's First COBOL Compilers [Archived](https://web.archive.org/web/20111013021915/http:/www.computerhistory.org/events/lectures/cobol_06121997/index.shtml) 13 October 2011 at the [Wayback Machine](https://en.wikipedia.org/wiki/Wayback_Machine" \o "Wayback Machine)

[3]Knuth, Donald E.; Pardo, Luis Trabb. "Early development of programming languages". Encyclopedia of Computer Science and Technology. 7: 419–493.

[4] Peter J. Bentley (2012). Digitized: The Science of Computers and how it Shapes Our World. Oxford University Press. p. 87. ISBN 9780199693795. Archived from the original on 29 August 2016.

[5] Backus et al. "The FORTRAN automatic coding system", Proc. AFIPS 1957 Western Joint Computer Conf., Spartan Books, Baltimore 188–198

[6]T. Hart and M. Levin "The New Compiler", AIM-39[permanent dead link] CSAIL Digital Archive – Artificial Intelligence Laboratory Series

[7] Knuth, Donald. "On the Translation of Languages from Left to Right" (PDF). Archived from the original (PDF) on 15 March 2012. Retrieved 29 May 2011.

[8]"META II a syntax-oriented compiler writing language". acm.org.

[9] "Comp.compilers: Re: History and evolution of compilers". iecc.com.

[10]McKeeman, W.M. Peephole optimization. Commun. ACM 8, 7 (July 1965), 443–444

[11] http://www.bitsavers.org/pdf/ibm/360/A22\_6825-1\_360instrTiming.pdf

[12] "Software engineering for the Cobol environment". acm.org.