Communications on Applied Electronics (CAE) – ISSN : 2394-4714
Foundation of Computer Science FCS, New York, USA
Volume 7 – No. 7, October 2017 – www.caeaccess.org

Qualitative Assessment of Compiled, Interpreted and Hybrid Programming Languages

Ampomah Ernest Kwame
Computer Science Department,
Christian Service University College
Santasi-Kumasi, Ghana

Ezekiel Mensah Martey
Department of Information and Communication Technology,
Christ Apostolic University College
Kwadaso-Kumasi, Ghana

Abilimi Gilbert Chris
Computer Science Department,
Christian Service University College
Santasi-Kumasi, Ghana

ABSTRACT
Programmers use programming languages to develop software. But how efficiently programmers can write software depends on the translation mode that is used by them. In computing, the general modes of execution for modern high-level languages are interpretation, compilation and hybrid. The selection of these general modes of execution is dependent on the choice of programming language. In this paper, the study compared compiled, interpreted and hybrid programming languages under translation process, execution, efficiency, portability, maintainability, and security. The paper contains idea about how software development are influenced by the decision to use compiler, interpreter or hybrid as a mode of translation hence purpose of this research. It was observed that compiled, interpreted and hybrid programming languages have their strengths and weakness, hence the need for programmers to critically analyzed their goal and how the various programming languages will help to achieve that goal before choosing a language.

Keywords
Compiled, Interpreted, Hybrid, Programming Language,

1. INTRODUCTION
Computers have become one of the most essential devices needed for human existence. However, they are useless without a computer program. A computer requires a program to be able to perform a task. A computer program can be described as a set of instructions that follow the rules of the programming language been used. A program has statements and may contain variables that tell the computer what it should do with the variables [18]. In recent times, almost all programs are written in high-level programming languages. High-level language programs cannot be run directly on by the central processing unit (CPU) of the computer rather it has to be translated into machine language, hence, the need to translate high level language instructions called source code into machine code which can be understood by the computer [23]. Translation of source code into machine code is accomplished by using compiler or interpreter. As a result, there are compiled, interpreted and hybrid high level programming languages. This paper compared these three kinds of high level languages qualitatively pointing out their mode of operation, merits and demerits

1.1 Programming Language
Programming languages are used to write computer programs such as applications, systems programs and utilities. Programmers and developers use programming language to develop software programs and other sets of instructions for computers to execute. There are three levels of programming language and these are machine language (which is classified as Low-level language), Assembly language (which is also classified Low-level language) and High-level language [25].

1.1.1 Machine Language
Machine language comprises of collection of binary digits or bits that the computer reads and interprets without the need for translation. Hence, it is the language a computer understands [16]. Machine language is designed to be recognized by a computer and can be described as basic type of low-level programming language. The machine language program is written in a binary code of 0s and 1s that represent electric impulses or off and on electrical states respectively. A group of such digits is called an instruction and it is translated into a command that the CPU understands [1]. Every computer has its own kind of machine language, and the computer can directly execute a program without the need for translation if only the program is expressed in that language. A computer can execute programs written in other languages if they are first translated into machine language by a language translator [9]. Instructions contained in machine-language causes the CPU of the computer to perform operations such as an arithmetic calculation or storing data in the random access memory of the computer.

1.1.2 Assembly Language
Assembly Language is an intermediary programming language between machine language and a high-level language. It is classified as low-level programming language that is made up of instructions that are mnemonic codes for corresponding machine language instructions [13]. Assembly language are used by programmers to directly create instruction code programs without having to worry about the various instruction code set combinations on the processor. The mnemonics enables the programmer to use English-style words to represent individual instruction codes and they can easily be translated to the machine language instruction codes by an assembler [28]. Assembly language programs are non-portable; a program must be rewritten to run on a different machine. An assembly language program has three components that are used to define the program operations and these are Opcode mnemonics, Data sections and Directives [8]. To help facilitate writing the instruction codes, assemblers equate mnemonic words with instruction code functions, such as moving or adding data elements. Instead of having to know what each byte of instruction code represents, the assembly language programmer can use easier-to-

remember mnemonic codes, such as push, mov, sub and call, to represent the instruction codes [7].

1.1.3 High Level Language

A high-level language is a programming language that permits development of a program in a more user-friendly programming context. High level language resembles human language or notation used in mathematics [24]. Programming in machine and assembly language are tiresome and the process is error-prone. Hence, most programming are done using a high-level programming languages. Using high level languages make programs easier to read, write, and maintain than assembly and machine languages. A high-level language has a higher level of abstraction from the computer and it allows programmers to concentrate on the programming logic rather than the underlying hardware components such as memory addressing and register utilization [20] A high-level language does not require addressing hardware constraints when developing a program. However, high-level language programs must be translated into machine language by a compiler or interpreter before the computer can execute the program [6].

2. METHODOLOGY

A comparison between interpreted, compiled and hybrid programming languages were done on the basis of the following parameters: translation process, portability, execution efficiency, maintainability, safety.

2.1 Translation process

Translation process is the process that a source code written in a high level programming language undergoes before it is converted into a low level machine language. Translation process is different for the various translators for converting source code to machine code.

2.1.1 Compiled Language

A compiler translates the source code to machine code for a particular platform. The source code is translated into the computer's native language up front by the compiler, before the execution of the program [27]. Program compilation is a two-step process which are the compiling step and the linking step [22]. Compiling step produces an intermediate file, often called as object code file. This file has instruction codes that represent the core of the application functions. Every line of the source code are matched up with one or more instruction codes pertaining to the specific processor on which the application will run. Examples of compiler programming languages include C, C++. The linking step uses a linker to link the object code file with other object files needed by the application, and creates the final executable output file. The linker's output is an executable file that runs only on the operating system for which the program is written [7]. As shown in figure 2.1, a compiler internal architecture consists of a front-end layer, an intermediate layer, and a back-end [12]. The compilation process is categorized into several phases with well-defined interfaces. The phases operate in sequence, and each phase uses the output from the preceding phase as its input. A common division into phases is as follows [26]:

- Lexical analysis: this phase comprises of the initial part of reading and analyzing the program text: The text is read and divided into tokens, each of which corresponds to a symbol in the programming language.
- Syntax analysis: in this phase the list of tokens produced by the lexical analysis are taken and arranged in a tree-structure known as the syntax tree that reflects the structure of the program. This phase is often called parsing.
- Type checking: during this phase the syntax tree are analyzed to determine if the program violates certain consistency requirements.
- Intermediate code generation: there is translation of the program to a simple machine independent intermediate language.
- Register allocation: in this phase, there is translation of symbolic variable names used in the intermediate code into numbers, and each of them corresponds to a register in the target machine code.
- Machine code generation: the intermediate language is translated to assembly language for a specific machine architecture.
- Assembly and linking: this phase translate the assembly-language code into binary representation, and the addresses of variables, functions and others are determined

2.1.2 Interpreted Language

Translation of source code of an interpreted language does not happen in advance. Translation occurs concurrently with the execution of the program. The interpreter starts interpreting each instruction instantly upon execution. Hence, the source code of an interpreted language is executed directly on the target platform by an interpreter [10]. The interpreter converts source code into machine code line by line each time the program is executed. Figure 2.2 shows the structure of an interpreter for a textual programming language. The translation steps of an interpreter are as follow [22]:

- Lexical analysis which involves transforming stream of characters into a stream of recognized tokens.
- Syntactical analysis which follows lexical analysis and produces an abstract representation of a program.
- The semantics of a program is then analyzed, based on the constructions of the source language to generate an annotated representation.
- The abstract representation is evaluated on some given input data. The result of executing the interpreter is the program's output data. Python is an example of an interpreted programming language.

2.1.3 Hybrid Language

Hybrid language make use of both compilation and interpretation to execute the source code. The source code is first compiled into what is known as byte code. Bytecode describes computer object code that a program known as a virtual machine processed rather than the computer's processor [17]. Bytecode is kept in a dot (.) class file format. Virtual machine (VM) which is an interpreter uses the dot class as input to produces output by executing the bytecode. The byte code is interpreted by a Virtual Machine which runs separately on the host computer. Java and C-Sharp are some

examples of hybrid programming language [5].

2.2. Portability

Portability measures the easiness an application can be moved from one computer environment to another. That is the usability of the same application in different environments.

2.2.1 Compiled Languages

A compiler translates source code into a machine code that is specific to the target machine. Hence it has a low portability [18]. For instance, a compiled program for Intel Core 2 Duo will not work for Intel Pentium 4. A programmer must produce several versions of source code for the same application when getting a product out to the market. This causes the programmer to spend more time and resources on source code maintenance and updates.

2.2.2 Interpreted Language

An interpreter converts source code into machine code one line at a time, each time the program is executed. The source code is executed directly on the target platform by the interpreter [23]. Hence Interpreted languages are highly portable across different kinds of hardware. Executable shipped to every platform is the same.

2.2.3 Hybrid Language

Output of a Hybrid Language compiler is known as byte code. This is a non- executable code. Run-time system known as Virtual Machine which is an interpreter executes the bytecode [10]. The Translation of a source code into bytecode permits programmers to run a program in different kinds of environments since only the VM needs to be implemented for the different platforms. Hence Hybrid Language is portable.

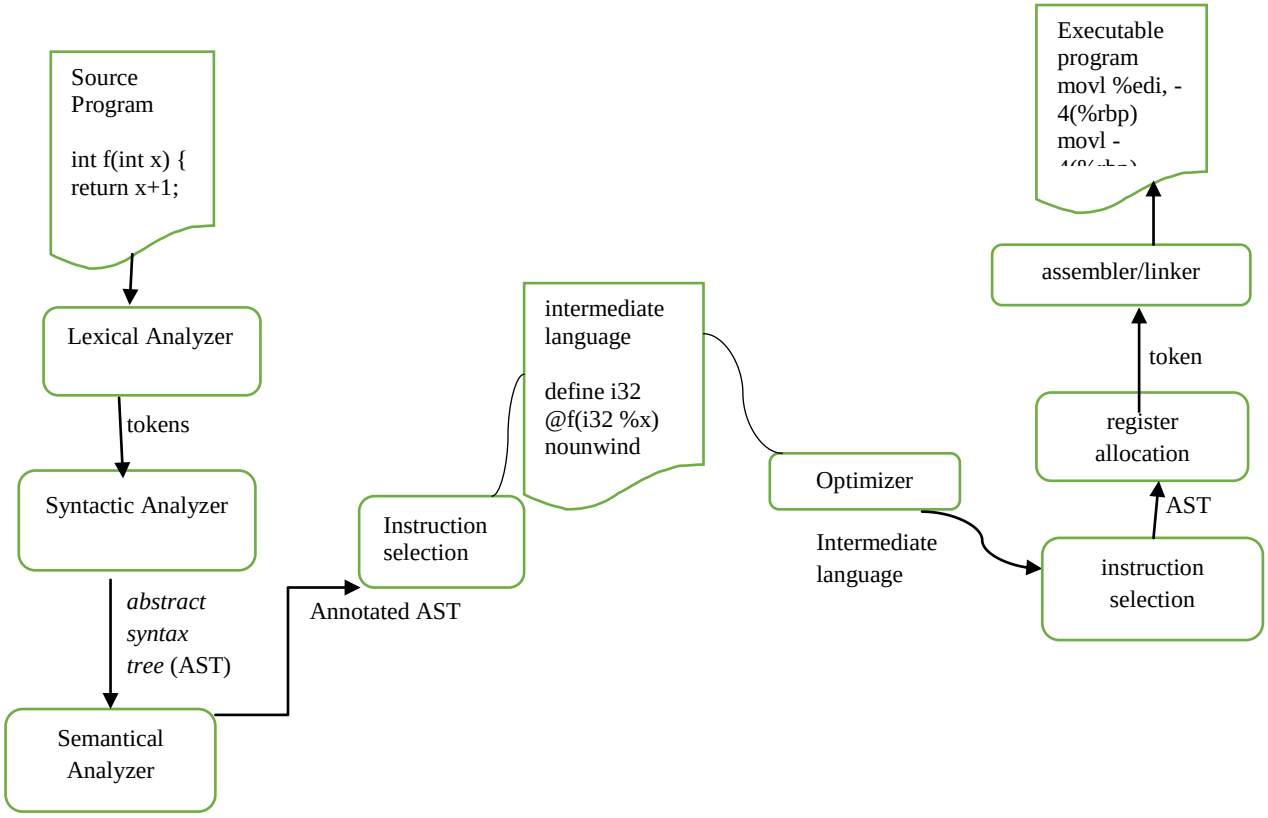

Figure 1: Architecture of Compiler [13]

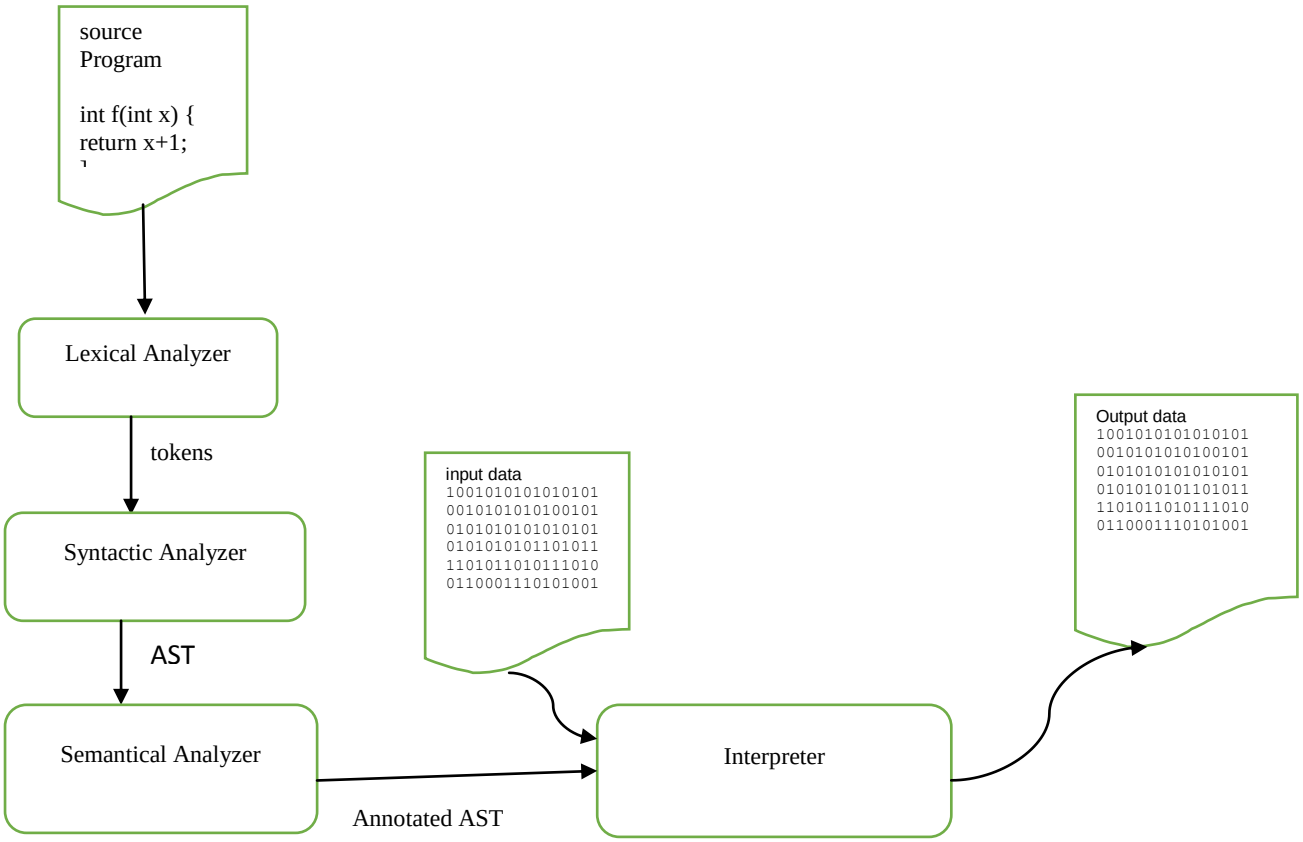

Figure 2: The architecture of an interpreter [13]

2.3 Execution Efficiency

The time during which a program is running is termed as Execution or run time. Hence, execution efficiency of a programming language is the ability of programs written in that language to execute within a reasonable amount of time.

2.3.1 Compiled Language

The compiler translate the source code into platform specific machine code [27]. Since the source code is converted into platform specific machine code, the execution of the code does not require any more translation, and hence compiled languages run significantly faster and more efficiently.

2.3.2 Interpreted Language

An interpreter reads each line of the source code and execute the required task [23]. The time used by interpreter to read and execute the source code delays the execution of the program. Also the source program is always translated fresh every time it is used [21], and this slow the process causing execution to take more time**.**

2.3.3. Hybrid Language

In hybrid languages, during the initially running of the program the source code is compiled into bytecode [15]. Hybrid programs tend to perform slower than equivalent programs written in compiled languages since the bytecode must be interpreted by the VM when the program is run. However, the execution speed of hybrid languages are significantly higher compared to interpreted languages since the bytecode is optimized for the interpreter.

2.4 Maintainability

Maintainability is a measures of the easiness to understand, repair or improve a computer program or application.

2.4.1 Compiled Language

Compiled language is static typed language and hence errors in programs are detected at the compilation stage. It provide better documentation in the form of type signatures for programmer intent [2].Maintainability of compiled language programs is relatively easy.

2.4.2 Interpreted Language

Interpreted language is dynamic typed language and perform type checking at runtime therefore errors are detected at runtime. It also has a poor documentation [3]. Maintainability of interpreted language programs are therefore difficult.

2.4.3 Hybrid Language

Hybrid language is a hybrid typed language and performs type checking at both compilation stage and run time [14]. It has good documentation and hence, maintainability is relatively not as difficult as with interpreted languages.

2.5 Security

Security refers to ability to prevent unauthorized modification of a program's source code. Security is about confidentiality, integrity, availability and authentication.

2.5.1 Compiled Language

The source code of compiled language can be kept private [19]. The process of compilation breaks down the source code into very low-level binary codes and the instructions are

reordered [11]. There is clear distinction between executable file and the source code, and that makes it somewhat more secure.

2.5.2 Interpreted Language

For interpreted language, there is no distinction between the executable file and the source code [11]. The source code is public and available to everybody making it less secure as malicious users can tamper with the source code.

2.5.3 Hybrid Language

The source code is compiled into intermediate bytecode which is translated by the VM, hence the source code is distinct from the bytecode [5]. This enhances the security of the source code since it is not directly available to the public

3. EXPERIMENTAL ANALYSIS

Magnetic bubble sort algorithm [4] was implemented on C++, Python and Java platforms which are compiled, interpreted and hybrid programming languages respectively. The Central Processing Unit (CPU) execution time for the Magnetic bubble sort algorithm on these three languages were compared. Visual Studio profilers was used to determine the CPU execution time for C++ and Java and Python profiler was used to obtain the execution time on python. Five different data sizes 10000, 20000, 50000, 100000 and 150000 were used.

Table 1: CPU Execution Time for Magnetic Bubble Sort Algorithm on C++, Python and Java

Data Size	C++	Python	Java
10000	3.72	9.08	4.96
20000	7.25	14.46	8.68
50000	18.19	43.21	20.59
100000	49.79	110.23	55.75
150000	98.32	179.63	108.32

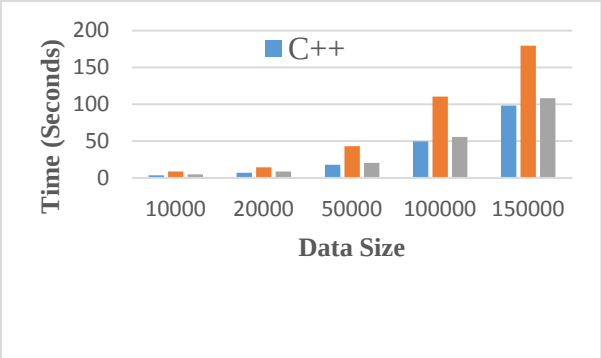

Figure3: CPU Execution Time for Magnetic Bubble Sort on Different Platforms

4. CONCLUSION

The study shown that both compiler and interpreter programming languages have varying advantages and disadvantages when used by programmers to write programs. The interpreter technique is slow and inefficient, since lines of code are repeated and translated while the program is running. However, due to the fact that anytime interpreted language program is run, the interpreter refers to the source code it is a relatively easy to modify and rerun a piece of code or to move the code to a computer environment different where it was developed and run. Interpreters are very good development tools since it can be easily edited, and are therefore ideal for beginners in programming and software development. However they are not good for professional developers due to the slow execution nature of the interpreted code. On the other hand the compiler technique translate the whole code into a single machine code program and run this machine code. Execution of code is very fast when using compiler technique, however, the code cannot be executed on any other platform apart from the one the code was developed on it. The hybrid language combines both techniques and minimizes the disadvantages associated with each of the two techniques while maintaining the advantages they have to some degree. Due to the high execution speed of both compiled and hybrid languages, they are good for professional software developers.

5. REFERENCES

[1] Aggeliki K. (2011). Machine Language vs High-Level Languages, Bright Hub Engineering; Retrieved: 2nd April, 2017. http://www.brighthubengineering.com/consumer-appliances-electronics/115635-machine-language-vs-high-level-languages/

[2] Agrawal, R., DeMichiel, L. G., & Lindsay, B. G. (1991). Static type checking of multi-methods Vol. 26, No. 11, pp. 113-128. ACM.

[3] Alpern, B., Cocchi, A., & Grove, D. (2001). Dynamic Type Checking in Jalapeño. In Java Virtual Machine Research and Technology Symposium.

[4] Appiah, O., & Martey, E. M. (2015). Magnetic Bubble Sort Algorithm. *International Journal of Computer Applications IJCA*, *122*(21), 24-28. doi:10.5120/21850-5168

[5] Arnold, K., Gosling, J., & Holmes, D. (2005). The Java programming language. Addison Wesley Professional.

[6] Bates, P. C., & Wileden, J. C. (1983). High-level debugging of distributed systems: The behavioral abstraction approach. Journal of Systems and Software, 3(4), 255-264.

[7] Blum Richard, (2005). Professional Assembly Language, Wiley Publishing, Inc. ISBN: 0-7645-7901-0

[8] Crary, K. (2003). Toward a foundational typed assembly language (Vol. 38, No. 1, pp. 198-212). ACM.

[9] Eck David J., (2011). Introduction to Programming Using Java, Hobart and William Smith Colleges, Geneva, NY 14456.

[10] Gosling, J., & McGilton, H. (1995). The Java language environment. Sun Microsystems Computer Company, 2550.

[11] Grillmeyer, O. (1998). Compilers and Interpreters. In Exploring Computer Science with Scheme (pp. 319-372). Springer New York.

[12] Jiménez, M., Palomera, R., & Couvertier, I. (2014). Assembly Language Programming. In Introduction to Embedded Systems (pp. 155-218). Springer New York.

[13] João Costa Seco, 2014, Interpretation and Compilation of

Programming Languages, Retrieved: 22nd May, 2017. http://docentes.fct.unl.pt/sites/default/files/jrcs/files/ln01-overview.pdf

[14] Knowles, K., & Flanagan, C. (2010). Hybrid type checking. ACM Transactions on Programming Languages and Systems (TOPLAS), 32(2), 6.

[15] Krauss Aaron, (2015). Programming Concepts: Compiled and Interpreted Languages; Retrieved: 16th May, 2017. https://thesocietea.org/2015/07/programming-concepts-compiled-and-interpreted-languages/

[16] Manson, K. S. (2006). U.S. Patent No. 7,085,708. Washington, DC: U.S. Patent and Trademark Office.

[17] McInnes James, (2014). How is source code typically kept secret?; Retrieved: 5th June 2017. https://www.quora.com/How-is-source-code-typically-kept-secret

[18] Morgan, C. (1994). Programming from specifications. Prentice Hall.

[19] Najjar, W. A., Bohm, W., Draper, B. A., Hammes, J., Rinker, R., Beveridge, J. R., ... & Ross, C. (2003). High-level language abstraction for reconfigurable computing. Computer, 36(8), 63-69.

[20] Organ, D. V., Deome, M. E., Techasaratoole, R., & Greene, V. N. (2001). Capturing and displaying computer program execution timing." Washington, DC: U.S.

[21] Redish, K. A., & Smyth, W. F. (1986). Program style analysis: A natural by-product of program compilation. Communications of the ACM, 29(2), 126-133.

[22] Rouse Margaret, (2005). Bytecode; retrieved: 16th May, 2017 http://whatis.techtarget.com/definition/bytecode

[23] Sanner, M. F. (1999). Python: a programming language for software integration and development. J Mol Graph Model, 17(1), 57-61.

[24] Schmidt, J. W. (1977). Some high level language constructs for data of type relation. ACM Transactions on Database Systems (TODS), 2(3), 247-261.

[25] Stroustrup, B. (2013). The C++ programming language. Pearson Education.

[26] Torben Ægidius Mogensen, (2010). Basics of Compiler Design, Anniversary edition, University of Copenhagen, ISBN 978-87-993154-0-6.

[27] Watson, D. (2017). Compilers and Interpreters. In A Practical Approach to Compiler Construction (pp. 13-36). Springer International Publishing.

[28] Xi, H., & Harper, R. (2001). A dependently typed assembly language. ACM SIGPLAN Notices, 36(10), 169-180.