Compiler design in the Programming language

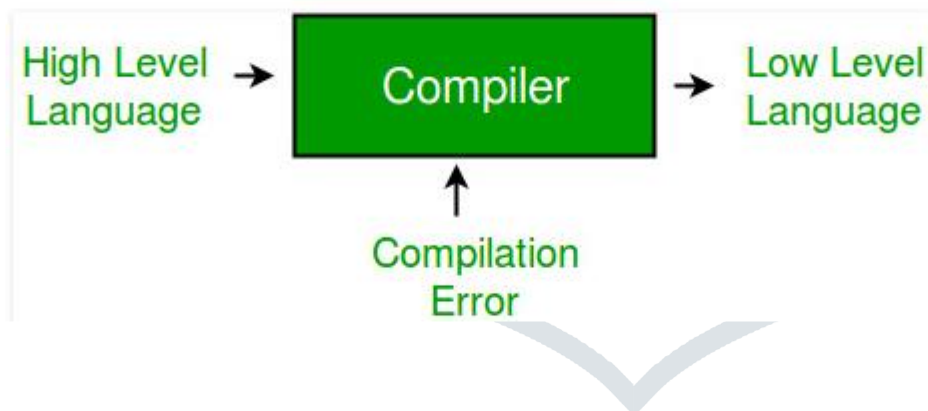
Muthu Dayalan
Senior software Developer & Researcher
Chennai,Tamilnadu

Abstract - **compiler design is used to translate a code written in one programming language to other languages, while maintaining the meaning of the codes. The high-level language is usually composed by a developer, and compiler converts it to a machine language that could be understood by the processor. There are two major phases of a compiler, which is based on they compile. These phases include analysis phases and the synthesis phase. The lexical phase is composed of taking the modified code source originating from the language processor. The context-free grammar is mandated to producing production rules, which are followed by the syntax analyzer.**

Index Terms - **Compiler design, programming language, syntax, lexical**

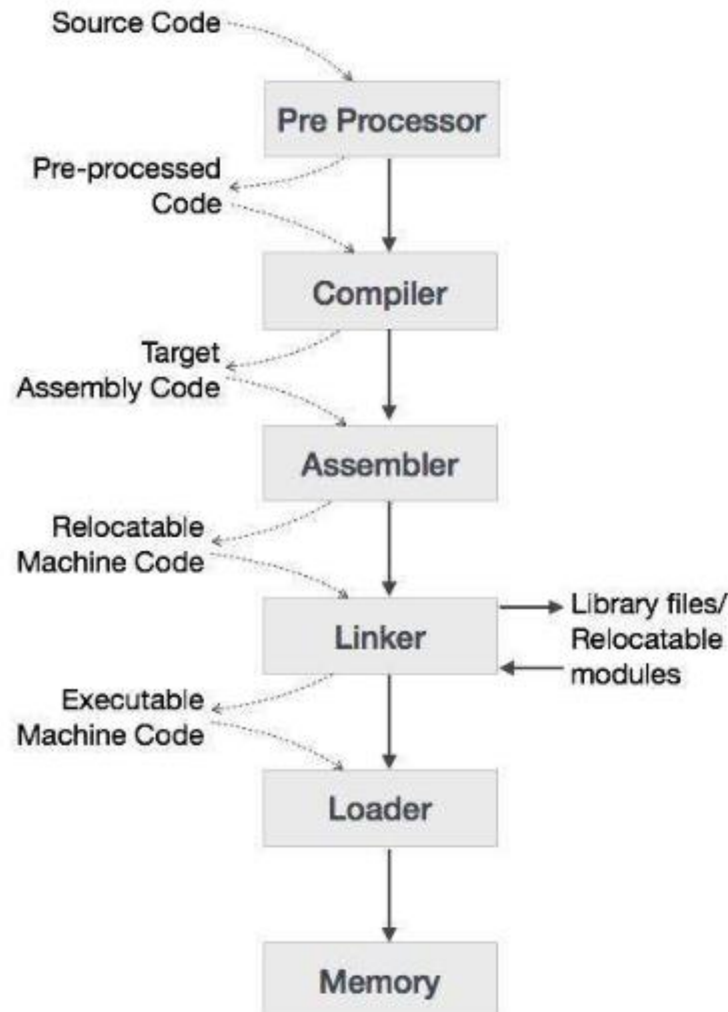
I. Introduction

In computer programming, the compiler design implies a computer program, which is used to transform source code written in high-level language in to low-level language or the machine language. Generally, compiler design is used to translate a code written in one programming language to other languages, while maintaining the meaning of the codes. The design is important as it produces an efficient end code for optimal execution in terms of time and memory [4]. The high-level language is usually composed by a developer, and compiler converts it to a machine language that could be understood by the processor. Additionally, the compiler is used to notify the programmer the errors made. The figure below shows the function of a compiler.



II. Language Processing System Using a Complier

The computer is made up of an assembly of software and hardware. The hardware could comprehend a language, which is difficult for us to understand. The computer programs are written in high-end languages, which is not complicated to be understood by human being. However, for these programs to be understood and executed by machines, they pass through a series of transformation - the language procedure systems shown below.



The procedure system is discussed below:

Pre-processor: the pre-processor is considered as a component of the compiler, which is applied for the macro-processing language extension and augmentation aspects [8].

Interpreter: this is similar to the compiler, which is used to convert high-level language in to low-level machine language. While the interpreter reads and transforms codes line by line, compiler reads the code at once to create a machine code.

Assembler: this component is used to translate the assembly language code in to a language that could be understood by the machine. The assembler produces an output referred to as object file, which implies a combination of a machine instructions and data storing these instructions [1].

Linker: the linker is used to merge several object files, in a manner that it could create an executable file. The major task performed by the linker is to search for the required module or routine in a particular program, and set the location of the memory where these codes could be loaded. As a result, an absolute reference is created for the program [14].

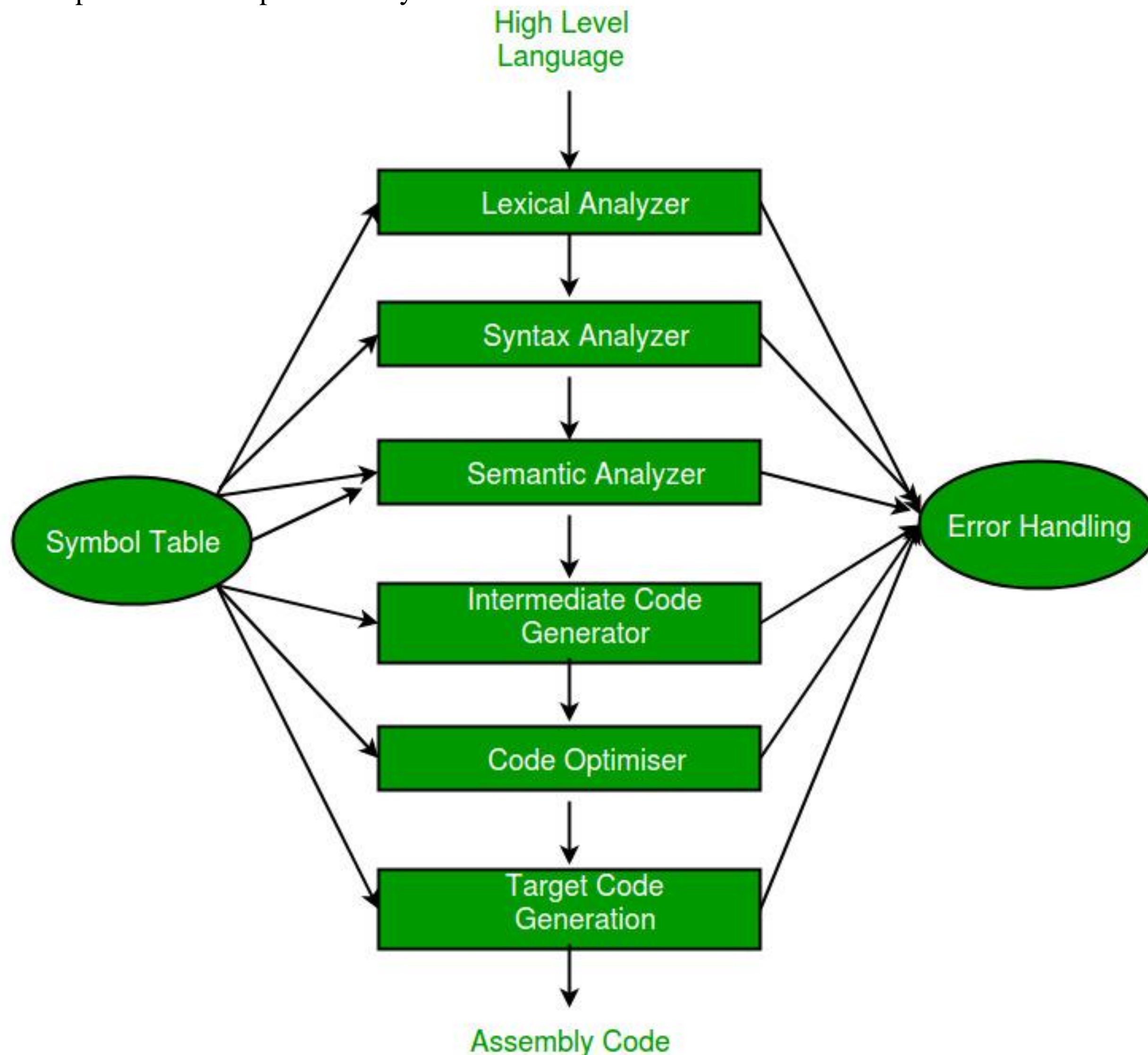
Loader: this section of the operating, which loads various executable files in to a memory, and prepares them for execution. By calculating the data and instructions of the program, the loader is able to create a memory space for it. It then develops various register for execution initiation.

III. Phases of Compiler

There are two major phases of a compiler, which is based on they compile. These phases include analysis phases and the synthesis phase. The two phases in turn has many parts. The output of one phase is the input of the following phase, which works in a coordinated manner [13].

Analysis phase

In this phase, an intermediate representation is generated from a particular source code. It is divided into Lexical Analyzer, Syntax Analyzer and semantic Analyzer. The second phase is the synthesis phase, in which helps in the generation of the target program through the help of the source code representation and symbol table. This phase is divided on to intermediate code generator code optimizer and code generator sub-phases. These phases are comprehensively discussed below.



Lexical Phase

This is the first phase, which functions as a test scanner. Its main task is to scan the source code in for of source character, and concert these characters in to a stream of lexemes.

Syntax Analysis

This is the second phase, in which the token from the lexical phase is taken and converted in to a parse tree, also referred to as the syntax tree. The arrangement of the token are checked in terms of source code grammar. This involves analyzing whether the present tokens are correct syntactically [2].

Semantic Analysis

Once the parse tree has been constructed in the syntax analysis phase, the semantic analysis checks whether it obeys the language rules. For instance, checking whether the assigned values are within the compatible data types, and addition of the string to the integer. It also checks the condition of the identifiers, their expression and type. The output of this phase is the annotated syntax tree [7].

Intermediate code generator

The nest step involve the generation of the intermediate code by the compiler, for the target machine. This is a program which could be used by the abstract machine, which is somewhere between the high-end and machine language. The code should be easy to translate in to a target machine code.

Code Optimization

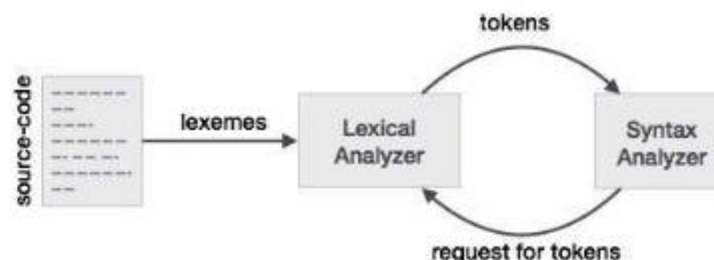
The nest phase involves the optimization of the intermediate code. This process involves the removal of the unnecessary codes, and aligning them in a way that they can executed using the least CPU memory possible [12].

Code Generator

This phase involves the mapping of the optimized representation from the intermediate code, with the targeted machine language [9]. The intermediate code is then translated in to a machine code that can be located. These functions are performed by a sequence of instructions.

IV. Compiler Design: Lexical Analysis

This is the first phase of the compiler design, which is composed of taking the modified code source originating from the language processor. These codes are written in sentences, and the work of the lexical analysis is to delete any inherent comments or whitespace and make them develop a series of tokens. In case there is an invalid token the analyzer displays an error [6]. The lexical and syntax analyzers works together in order to read various characters originating from the source code, to evaluate the legal tokens and send the data to the syntax in case it is needed. The lexical analysis architecture is displayed below.



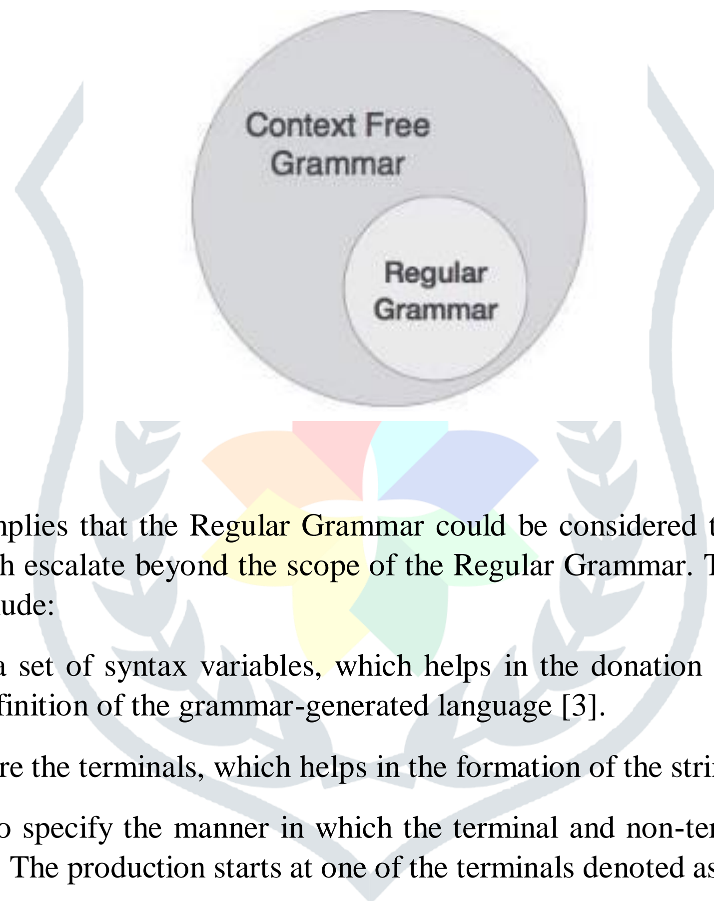
Tokens are composed of lexemes, which are a sequence of alphanumeric characters. For any lexeme to be considered as a valid token, there are various predetermine rules that must be observed. These rules are determined by grammar rules through a pattern. The patterns determine which could be considered as a token and what should not. In programming language, there are various components that should be considered as a language. These incudes the constants, keywords, numbers, strings, punctuation symbols as well as operators.

Under the specification of the tokens in compiler design, there are various terms understood by the compiler design. An alphabet is understood as any set of symbol. {0,1} considered as a binary alphabets. The following are considered as a hexadecimal alphabets are {0,1,2,3,4,5,6,7,8,9, A, B, C, D, E, F, }, while {a-z, A-Z} are considered as a set of English language alphabets. The arrangement of a finite number of alphabets is

considered as strings. There are also the special symbols such as arithmetic symbols, punctuation symbols, alignment symbols, and special alignment symbols [2]. The language is formed by a finite arrangement of finite set of strings from a finite set of alphabets.

V. Compiler Design: Syntax Analysis

This is the second phase of the compiler. While the lexical analyzer is mandated with identifying the token using the pattern rules and the regular expressions, it does not have the capacity to check the syntax. This is due to the regular expressions limitations [7]. Since the balancing tokens like the parenthesis could not be checked by the regular expressions, this phase uses the specialized context-free grammar (CFG) which has the capacity to recognize the push-down automata. It is important to note that the CFG is a superset of the Regular Grammar as shown in the figure below.



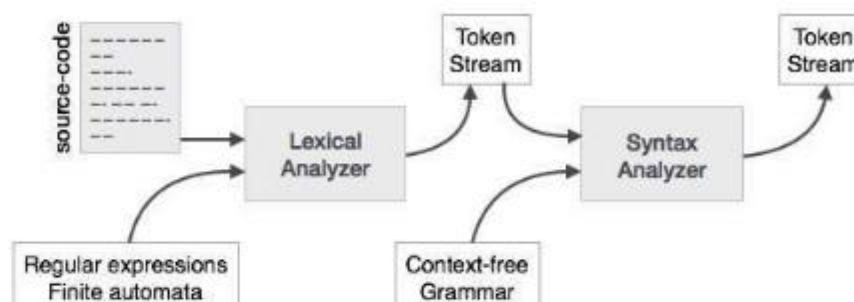
The graph above implies that the Regular Grammar could be considered to be context-free. However, there are other issues, which escalate beyond the scope of the Regular Grammar. There are four sections in the syntax grammar. These include:

Non-terminals: these are a set of syntax variables, which helps in the donation of the sets of strings. These terminals are vital in the definition of the grammar-generated language [3].

Terminal symbols: these are the terminals, which helps in the formation of the strings.

Productions: these helps to specify the manner in which the terminal and non-terminal could be combined in the formation of the strings. The production starts at one of the terminals denoted as the start symbol.

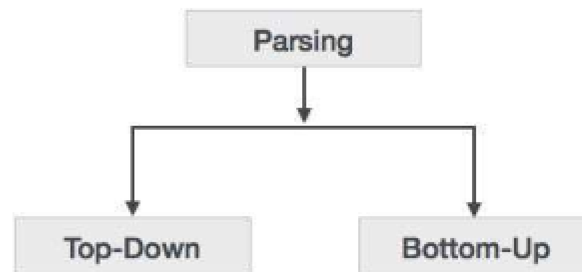
The lexical analyzer produces an output referred to as the token streams [4]. These tokens streams are used as input in the syntax analyzer. The analysis of the source code is done against the set production rules, so that any inherent errors could be detected. The output produced by this phase is known as the parse phase. The procedure undertaken in this phase is shown in the graph below.



In this case, therefore, there are two major tasks carried out by the passer. These are parsing of the code, and evaluating the errors and the generation of the parse tree in form of the output of this phase. However, it is vital to note that parsers are expected to parse the whole code, even if there are some errors which are identified in the program [10].

VI. Compiler Design: Types of Parsing

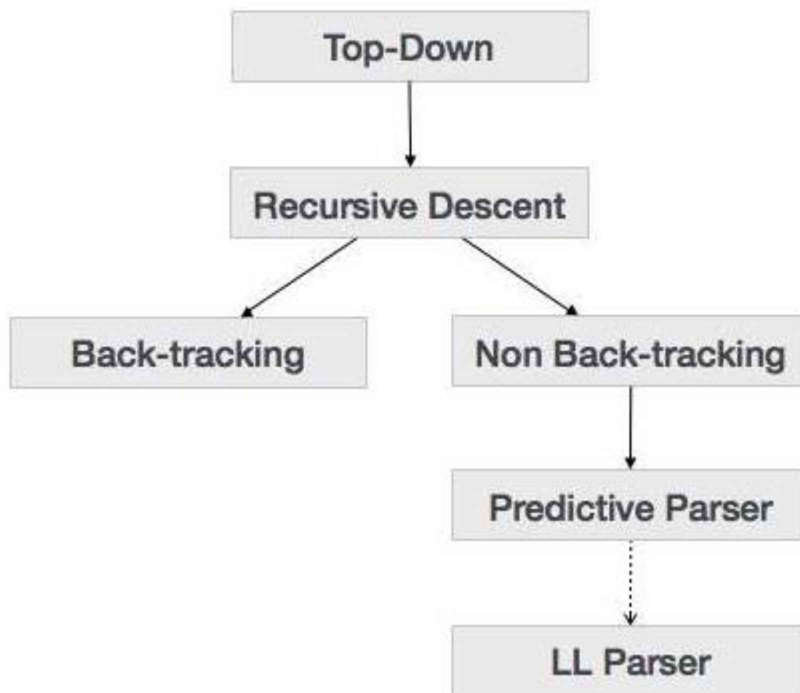
The context-free grammar is mandated to producing production rules, which are followed by the syntax analyzer. The implementation of the production rules involves dividing the parsing in to two major categories. These are the top-down parsing and the bottom-up parsing.



Top-down Parsing

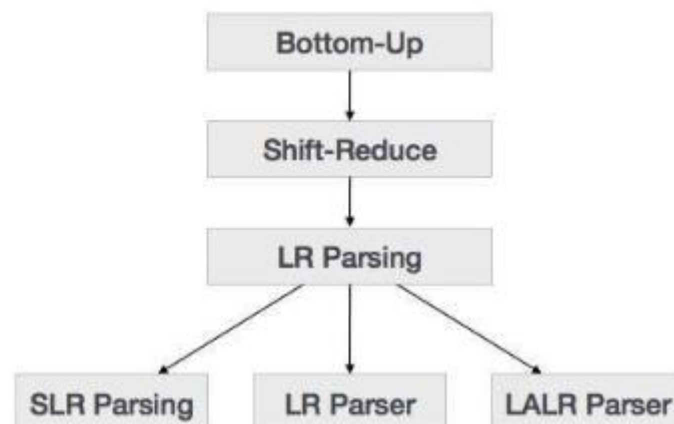
This process involves the construction of the parse tree by the parse from the start symbol, and then the transformation of the start symbol to the input. This process has two sections. The first section involves recursive descent parsing, which involves the application of the recursive procedures to process the inputs. The second is backtracking, which implies that in case of the failure of one of the derivation of production, the whole process is restarted by the use of the same production rules. To determine the right production process, backtracking may be done more than once. Under the general functionality of the top-down parser, the input is parsed leading to the construction of the parse tree from the root node [5]. In the recursive descent parsing, the construction of the parse tree is done from the top, while the input readings are done from the left to the right. However, it is hard to avoid the back-tracking on the used grammar. The predictive parse implies the parser which is used in the prediction of the production, which is applied to replace the input string. The overall task is accomplished by the look-ahead pointer, which is applied in the prediction of the nest input symbol.

The various types of this parsing is shown in the figure below.



Bottom-up Parsing

This is the reverse, which start with the input symbol, which makes an effort to construct the parse tree upwards to the start symbol. Starting from a sentence, the production rules are applied in reverse directions until the start symbols are reached [11]. There are two different steps applied in the shift-reduce parsing. These include the shift-step and the reduce-step. While the shift-step implies the advancement of the input pointer towards the next input pointer, the reduce-step implies the replacement of the complete grammar rule (RHS) with the LHS. This is shown in the image below.



The LR parser is considered as the most effective syntax analysis technique. The reason is that it is a non-recursive shift-reduce using a wide array of context-free grammar [6]. It is expected that the parser should have the capacity to detect and report any error, which is detected within the program. In case the error is encountered, then the parser should have the capacity to deal with it and continue with the parsing process to the rest of the input.

References

[1] Becker, B. A., Glanville, G., Iwashima, R., McDonnell, C., Goslin, K., & Mooney, C. (2016). Effective compiler error message enhancement for novice programming students. *Computer Science Education*, *26*(2-3), 148-175.

[2] Chong, F. T., Franklin, D., & Martonosi, M. (2017). Programming languages and compiler design for realistic quantum hardware. *Nature*, *549*(7671), 180.

[3] Heim, B. (2018, February). Compiler and language design for quantum computing (keynote). In *Proceedings of the 27th International Conference on Compiler Construction* (pp. 2-2). ACM.

[4] Jaco, D. C. A., & Sandoval, M. A. M. (2016, September). Design of a prototype programming language in spanish and its compiler through tools Jflex and Java Cup. In *2016 IEEE Central America and Panama Student Conference (CONESCAPAN)* (pp. 1-6). IEEE.

[5] Koeplinger, D., Feldman, M., Prabhakar, R., Zhang, Y., Hadjis, S., Fiszel, R., ... & Olukotun, K. (2018, June). Spatial: A language and compiler for application accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (pp. 296-311). ACM.

[6] Kundra, D., & Sureka, A. (2016, December). An experience report on teaching compiler design concepts using case-based and project-based learning approaches. In *2016 IEEE Eighth International Conference on Technology for Education (T4E)* (pp. 216-219). IEEE.

[7] Lee, Y., Jeong, J., & Son, Y. (2017). Design and implementation of the secure compiler and virtual machine for developing secure IoT services. *Future Generation Computer Systems*, *76*, 350-357.

[8] Lopes, L., & Martins, F. (2016). A safe-by-design programming language for wireless sensor networks. *Journal of Systems Architecture*, *63*, 16-32.

[9] Mogensen, T. Æ. (2017). *Introduction to compiler design*. Springer.

[10] Pu, J., Bell, S., Yang, X., Setter, J., Richardson, S., Ragan-Kelley, J., & Horowitz, M. (2017). Programming heterogeneous systems from an image processing DSL. *ACM Transactions on Architecture and Code Optimization (TACO)*, *14*(3), 26.

[11] Park, J., Esmaeilzadeh, H., Zhang, X., Naik, M., & Harris, W. (2015, August). Flexjava: Language support for safe and modular approximate programming. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (pp. 745-757). ACM.

[12] Stewart, G., Gowda, M., Mainland, G., Radunovic, B., Vytiniotis, D., & Patterson, D. (2015). Ziria: language for rapid prototyping of wireless PHY. *ACM SIGCOMM Computer Communication Review*, *44*(4), 357-358.

[13] Wei, R., Schwartz, L., & Adve, V. (2017). DLVM: A modern compiler infrastructure for deep learning systems. *arXiv preprint arXiv:1711.03016*.

[14] Wimmer, C., Jovanovic, V., Eckstein, E., & Würthinger, T. (2017, February). One compiler: deoptimization to optimized code. In *Proceedings of the 26th International Conference on Compiler Construction* (pp. 55-64). ACM.