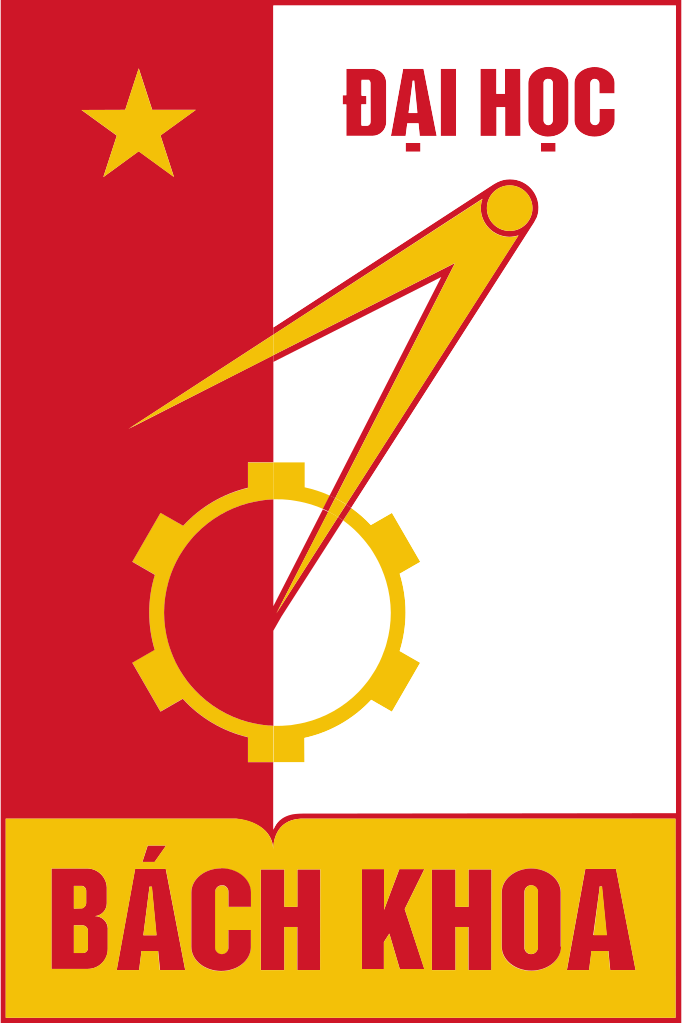
# TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI

--🙢🕮🙠--



# Final Report

**Name : Vũ Hoàng Tùng**

**Mssv : 20194880**

**Class : ICT03-K64**

THE FINAL REPORT

1. **Phases of a compiler** A compiler is a piece of software that can turn high-level programming language source code into machine code that can be run by a computer. This procedure consists of several stages or phases, each of which carries out a distinct function. We will go over the phases of a compiler and talk about the tasks they perform in this report.

* Tasks of a compiler  
    
  The main tasks of a compiler are as follows:

-Lexical Analysis: also known as scanning, which breaks down the source code into tokens such as keywords, identifiers, operators, and literals. These tokens are the smallest meaningful components of a program

-Syntax Analysis stage : also known as parsing, the compiler ensures that the sequence of tokens generated by the Lexical Analysis adheres to the programming language's grammar rules. It also creates a parse tree, which represents the program's syntactic structure.  
- Semantic Analysis: Incompatible assignments, type mismatches, undefined variables, and other semantic errors are examined in this phase. It also manages the symbol table and checks type.  
- Code Generation: The optimized intermediate code is transformed during this phase into machine code that the target computer can execute. The operating system or the hardware can further optimize the generated code.

* Overview of phases of a compiler  
    
  The following is a brief overview of the phases of a compiler:
* Lexical Analysis: The source code is scanned and converted into tokens.
* Syntax Analysis: The tokens are parsed and a parse tree is constructed.
* Semantic Analysis: The parse tree is checked for semantic errors and a symbol table is generated.
* Code Generation: The optimized intermediate code is translated into machine code.

In conclusion, converting high-level source code into machine code requires the use of a sophisticated software tool known as a compiler. Compiler phases are interdependent and must be carried out in a particular order to guarantee accurate and efficient translation. For software developers who want to create programs that are both effective and dependable, it is essential to comprehend the phases and tasks of a compiler.

1. **Lexical analysis**

* Tasks of a scanner  
  The scanner, or lexical analyzer, performs several crucial tasks to convert the source code into a sequence of tokens that can be processed by the compiler  
  The main tasks of a scanner are as follows:  
  - Reading the source code: The scanner reads the source code character by character.  
  - Skipping white space: The scanner skips white space such as spaces, tabs, and line breaks, which are not significant in the program.
* Recognizing tokens: The scanner recognizes tokens such as keywords, identifiers, operators, and literals.
* Handling errors: The scanner detects and handles lexical errors such as misspelled keywords or undefined symbols.
* Data structure  
    
  A finite automaton, also known as a state machine, is typically the data structure utilized in lexical analysis. A mathematical model with a transition function, an output function, a set of states, and a set of input symbols is called a finite automaton. By switching between states in response to the input symbols, the scanner employs the finite automaton to identify tokens.
* Implementation (description of important funtions like readIdentkeyword, skipComment…)  
    
   The implementation of a scanner involves writing code that performs the tasks described above. Here, we will describe some of the important functions used in lexical analysis.  
    
  - readIdentKeyword: This function determines whether a sequence of characters is an identifier or a keyword after reading it. The symbol table is used to keep track of keywords and identifiers that have already been seen.

- skipComment: The comments in the source code are skipped by this function. Remarks are not huge in the program and are regularly encased in extraordinary images, for example,/\* \*/or//. The skipComment capability identifies these images and skirts the characters for the rest of the remark is reached.

- getNextToken: The following token is read from the source code by this function. It uses the finite automaton to identify the token based on the input symbols after omitting comments and white space.

- handleError: This capability recognizes and handles lexical mistakes. It attempts to recover by skipping to the next token after notifying the user of the error.

All in all, lexical examination is a significant period of a compiler that includes perusing the source code and changing over it into a grouping of tokens. The scanner handles errors, skips comments and white space, and recognizes tokens among other functions. A finite automaton and the writing of functions like readIdentKeyword, skipComment, getNextToken and handleError are typically used to implement a scanner.

1. **Syntax analysis**The Syntax Analysis stage, also known as parsing, is the second step in a compiler. Its primary responsibility is to ensure that the sequence of tokens produced by the scanner adheres to the syntax rules of the programming language. This report section discusses various aspects of Syntax Analysis, including the KPL grammar, the function of a parser, two approaches to syntax analysis, important functions such as compileStatement, compileExpression, compileTerm2, and compileFactor, and the corresponding set of syntax rules.

* Role of a parser.  
    
  The role of a parser is to take the sequence of tokens produced by the scanner and construct a parse tree that represents the syntactic structure of the program. The parse tree is a hierarchical structure that shows how the program is composed of statements, expressions, and other language constructs. The parser uses a set of syntax rules to recognize the structure of the program and generate the parse tree.
* Syntax directive approach  
    
  The syntax directive approach uses special directives in the source code to specify a programming language's syntax rules. These directives, which use a notation similar to Backus-Naur Form (BNF) to describe the syntax rules, are typically placed in comments or other special sections of the source code. These directives are used by the parser to identify the program's structure and produce the parse tree.
* Recursive descent method  
    
  A parsing strategy known as the recursive descent method entails writing a set of recursive procedures, one for each grammar symbol that is not a terminator. In order to identify program substructures, each procedure calls other procedures recursively in accordance with a syntax rule. Simple compilers frequently employ the recursive descent technique, which is simple to implement.
* KPL Grammar  
    
  KPL (K Programming Language) is a simple programming language used to illustrate the concepts of compiler construction.
* Implementation (description of important funtions like compileStatement, compileExpression, CompileTerm2, compileFactor together with corresponding set of syntax rules)  
    
  Writing code that follows the programming language's syntax rules is necessary for putting a parser into action. In this section, we will go over a few key syntax analysis functions.
* compileStatement: This function compiles a statement and generates the corresponding code. It recognizes the different types of statements using the syntax rules and calls other functions such as compileExpression to generate the code for expressions.
* compileExpression: This function compiles an expression and generates the corresponding code. It uses the syntax rules to recognize the structure of the expression and generates code for each sub-expression using other functions such as CompileTerm2.
* CompileTerm2: This function compiles the second part of a term (after the first factor) and generates the corresponding code. It uses the syntax rules to recognize the structure of the term and generates code for each sub-expression using other functions such as compileFactor.
* compileFactor: This function compiles a factor and generates the corresponding code. It uses the syntax rules to recognize the structure of the factor and generates code for each sub-expression. The sub-expressions may include identifiers, literals, and expressions enclosed in parentheses.

The parser uses the syntax rules to identify the program's structure and produce the parse tree. The code generator then uses the parse tree to generate the appropriate machine code.

In conclusion, a compiler's syntax analysis phase involves generating a parse tree and examining the program's structure. The parser generates the parse tree by recognizing the program's structure using a set of syntax rules. This section discussed the KPL grammar and the implementation of crucial functions like compileStatement, compileExpression, compileTerm2, and compileFactor.

1. **Semantic analysis**The semantic analysis phase of a compiler is responsible for verifying that the program is semantically correct, meaning that the program makes sense and conforms to the language's rules and standards. This phase involves checking the types of expressions and statements, checking if identifiers are declared and used correctly, and building a symbol table that is used in subsequent phases of the compiler.

* ***Symbol Table***The symbol table is a data structure used by the compiler to keep track of the program's identifiers, their types, and other attributes. The symbol table includes the following components:

- Attributes of objects: Each object in the symbol table has a set of attributes associated with it. For example, a variable may have attributes such as its type, scope, and whether it is a constant or not.

- Scope: Each identifier has a scope that determines where it can be accessed. The scope of an identifier is usually defined by the block in which it is declared.

- Global Object List: The symbol table also contains a list of all global objects in the program, such as global variables and functions.

* ***Scope management***Scope management involves ensuring that identifiers are declared and used correctly. The following functions are used in scope management:

- checkFreshIdent: This function is used in a declaration to ensure that an identifier is not already declared in the current scope.

- checkDeclaredLValueIdent: This function is used to check if an identifier has been declared before it is used as a value (e.g., in an expression).

- checkDeclaredIdent: This function is used to check if an identifier has been declared before it is used as a reference (e.g., in an assignment statement).

* ***Type checking***Type checking involves ensuring that expressions and statements are of the correct type. The following functions are used in type checking:
* Type checking in statements: The expression types on the left and right sides of the equal sign must match for statements like assignments and function calls.
* Type checking in expression: The operands and operators of an expression determine its type. For instance, an arithmetic operation's operands must be of the numerical type.
* Check number of dimension in an array: When using arrays in expressions or assignments, the number of dimensions must match.
* Check list of arguments: The number and types of arguments passed to a function must match its signature when it is called.
* Check call-by-reference parameters: The types of the arguments must match the types of the call-by-reference parameters in a function.