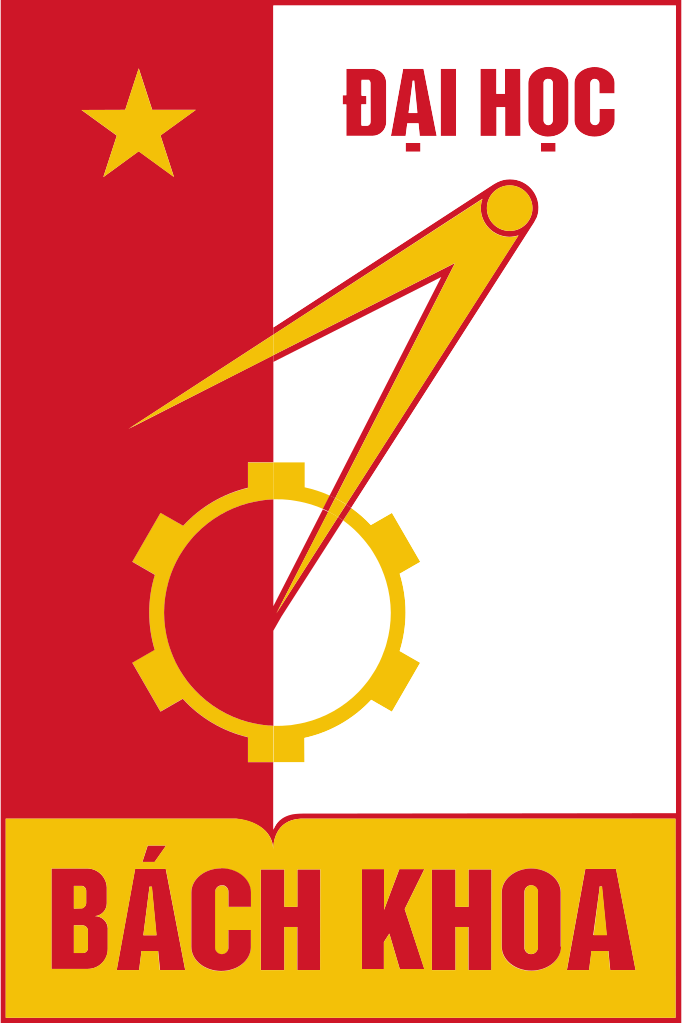
# TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI

--🙢🕮🙠--



# Final Report

**Name : Hoàng Thanh Lâm**

**Student ID : 20194786**

**Class : ICT02-K64**

**Course : Compiler Construction**

**TABLE OF CONTENTS**

**I. PHASE OF A COMPILER**

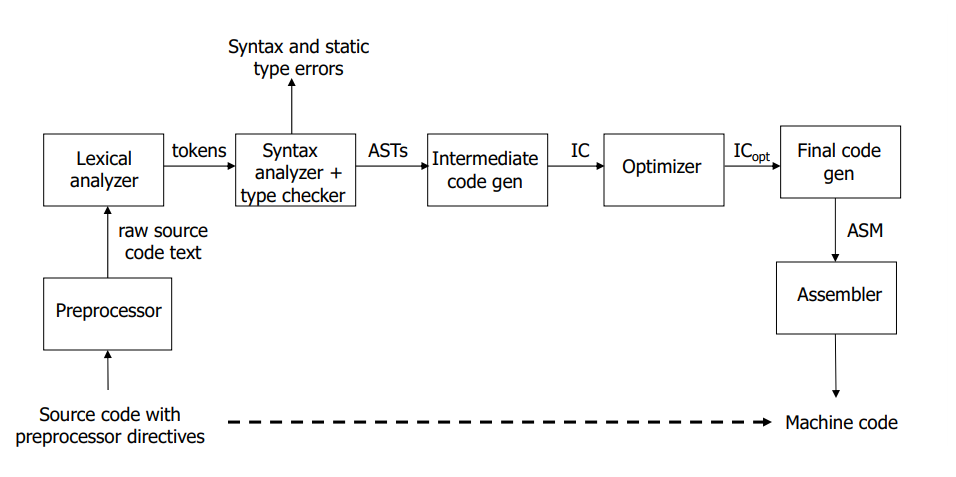
**II. LEXICAL ANALYSIS**

**III. SYNTAX ANALYSIS**

**IV. SEMANTIC ANALYSIS**

1. **Phases of a compiler** A compiler is a computer program that can transform high-level programming language source code into machine code. This process is divided into various stages or phases, each of which serves a particular purpose. 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:
* Scans the entire program and translates it as a whole into machine  
  code.
* Takes large amount of time to analyze the source code but the  
  overall execution time is comparatively faster.
* Generates the error message after scanning the whole program.
* Overview of phases of a compiler  
    
  The following is a brief overview of the phases of a compiler:
* Preprocessing: Conditional macro text substitution.
* Lexical Analysis: The source code is scanned and converted keywords, identifiers, constants into a sequence of tokens.
* Syntax Analysis: The tokens are parsed and a parse tree is constructed, also check that is syntactically correct.
* Semantic Analysis: The Parse Tree is checked for semantic errors and a symbol table is generated.
* Intermediate code generation: “walk” the Parse Trees and generate intermediate code. Applying optimizations to produce efficient code.
* 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 complex software tool known as a compiler. In order to ensure correct and effective translation, compiler processes must be completed in specific phases. 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, also known as a lexical analyzer, completes a number of critical tasks to turn the source code into a series of tokens that the compiler can process.

The following are the primary duties of a scanner:  
- Reading the source code: The scanner reads the source code character by character.  
- Skipping white space: Spaces, tabs, and line breaks that are insignificant to the program are skipped by the scanner.

* Recognizing tokens: The scanner recognizes tokens such as keywords, identifiers, operators, and literals.
* Handling errors: The scanner finds and corrects lexical mistakes like misspelled words or symbols that aren't defined.
* Data structure  
    
  The data structure used in lexical analysis is often a finite automaton, commonly referred to as a state machine. A finite automaton is a mathematical model comprising a transition function, an output function, a set of states, and a set of input symbols. The scanner uses the finite automaton to recognize tokens by flipping between states in response to input signals.
* Implementation (description of important functions 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: After reading a string of letters, this method determines if it is an identifier or a keyword. The symbol table is used to keep track of previously observed keywords and identifiers.

- skipComment: This function ignores the comments in the source code. Comments are not large in the software and are frequently enclosed in remarkable graphics, like /\* \*/or \*\*. The skipComment feature recognizes these images and skips over the remaining characters till the end of the comment.

- getNextToken: This function reads the following token from the source code. After removing comments and white space, it uses the finite automaton to determine the token based on the input symbols.

- handleError: Lexical errors are recognized and handled using this feature. After warning the user of the problem, it makes an effort to recover by skipping to the following token.

Overall, the lexical inspection phase of a compiler entails reading the source code and converting it into a collection of tokens. Among other things, the scanner handles mistakes, ignores comments and white space, and detects tokens. A scanner is often implemented using a finite automaton and the creation of functions like readIdentKeyword, skipComment, getNextToken, and handleError.

1. **Syntax analysis**The second step of a compiler is the Syntax Analysis stage, also known as parsing. Its main duty is to make sure that the tokens generated by the scanner follow the syntax conventions of the computer language. The KPL grammar, the role of a parser, two different approaches to syntax analysis, significant functions like compileStatement, compileExpression, compileTerm2, and compileFactor, and the related set of syntax rules are all covered in this report section..

* 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 program's statement, expression, and other linguistic elements are organized hierarchically in the parse tree. The parser creates the parse tree by recognizing the program's structure using a set of syntax rules.
* Syntax directive approach  
    
  The syntax directive approach specifies a programming language's syntax rules using special directives in the source code. These directives, which specify the syntax rules using a notation example Backus-Naur Form (BNF), are generally inserted in comments or other specialized areas of the source code. The parser employs these directives to determine the structure of the program and generate the parse tree.

* Recursive descent method  
    
  The recursive descent method, a parsing technique, calls for creating a sequence of recursive procedures, one for each grammar symbol that isn't a terminator. Each procedure makes recursive calls to other procedures in accordance with a syntactic rule to identify program substructures. The straightforward to build recursive descent technique is commonly used in simple compilers.
* 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)  
    
  To use a parser, you must write code that adheres to the syntactic requirements of the programming language. We will discuss a few important syntax analysis functions in this section.
* compileStatement: This function compiles a statement and generates the corresponding code. Using the syntax rules, it distinguishes between various statement kinds and executes methods like compileExpression to create expression code.
* compileExpression: This function compiles an expression and generates the corresponding code. It recognizes the expression's structure using the syntax rules, and then utilizes methods like CompileTerm2 to create code for each subexpression.
* CompileTerm2: This function compiles the second part of a term (after the first factor) and generates the corresponding code. It recognizes the term's structure using the syntax rules, and then creates code for each sub-expression using additional methods like compileFactor.
* compileFactor: This function compiles a factor and generates the corresponding code. The factor's structure is recognized using the syntax rules, and code is generated for each sub-expression. Identifiers, literals, and parenthesized expressions can all be used as sub-expressions.

In summary, a compiler's syntactic analysis stage creates a parse tree and analyzes the structure of the program. By identifying the program's structure using a set of grammar rules, the parser creates the parse tree. The KPL language and the implementation of significant functions like compileStatement, compileExpression, compileTerm2, and compileFactor were covered in this part.

1. **Semantic analysis**A compiler's semantic analysis phase is in charge of determining if the program is semantically valid, or whether it makes sense and complies with the language's norms and standards. The types of expressions and statements are examined, identifiers are appropriately declared and utilized, and a symbol table is constructed that will be used in later stages of the compiler.

* ***Symbol Table***The compiler uses a data structure called the symbol table to keep track of the identifiers, types, and other characteristics of the program. The following elements are included in the symbol table:

- Attributes of objects: There are a number of characteristics that may be applied to each item in the symbol table. For instance, a variable's type, scope, and whether it is a constant or not are all attributes.

- Scope: The scope of each identifier controls how and where it can be used. The block in which an identifier is declared often establishes its scope.

- 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: For statements like assignments and function calls, the expression types on the left and right of the equal sign must be the same.
* Type checking in expression: An expression's type is determined by its operands and operators. For instance, an arithmetic operation's operands must be of the numerical type.
* Check number of dimensions in an array: The number of dimensions must match when utilizing arrays in expressions or assignments.
* 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.