**HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY**

**SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY**

\*\*\*\*\*



COMPILER CONSTRUCTION

REPORT

**Student’s name: Pham Hai Dang**

**Class: ICT-K64**

**Instructor: Dr. Nguyen Thi Thu Huong**

***Hà Nội, February 2023***

Contents

[1. CHAPTER 1: Phases of a compiler 3](#_Toc128603664)

[1. Task of a compiler 3](#_Toc128603665)

[2. Overview of phases of a compiler 3](#_Toc128603666)

[2. CHAPTER 2: Lexical analysis 3](#_Toc128603667)

[1. Task of a scanner. 4](#_Toc128603668)

[2. DATA STRUCTURE IN KPL 4](#_Toc128603669)

[3. Implementation 4](#_Toc128603670)

[3. CHAPTER 3: Syntax analysis 5](#_Toc128603671)

[1. Role of a parser 5](#_Toc128603672)

[2. Syntax directive approach 5](#_Toc128603673)

[3. Recursive descent parsing 7](#_Toc128603674)

[4. KPL Grammar 8](#_Toc128603675)

[5. Implementation 8](#_Toc128603676)

[*4.* *CHAPTER 4. Semantic analysis* 11](#_Toc128603677)

[1. Symbol table 11](#_Toc128603678)

[2. Scope management 11](#_Toc128603679)

[3. Type checking 12](#_Toc128603680)

# CHAPTER 1: Phases of a compiler

## Task of a compiler

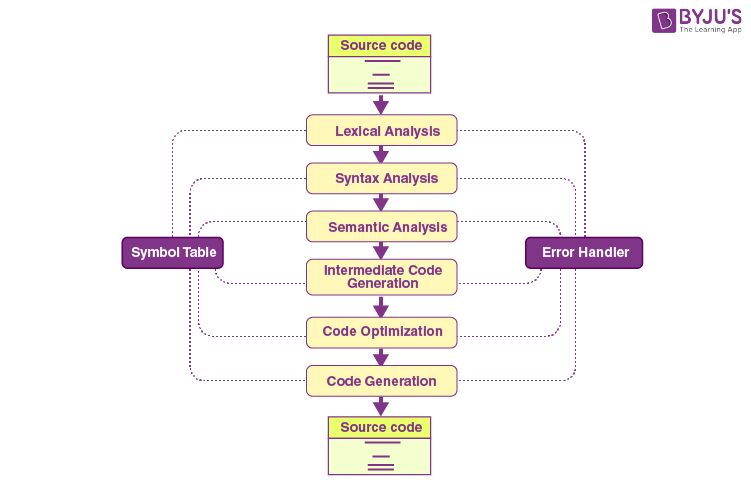
A compiler is a computer program which helps you transform source code written in a high-level language into low-level machine language. It translates the code written in one programming language to some other language without changing the meaning of the code. The compiler also makes the end code efficient, which is optimized for execution time and memory space.

The compiling process includes basic translation mechanisms and error detection. The compiler process goes through lexical, syntax, and semantic analysis at the front end and code generation and optimization at the back-end.

## Overview of phases of a compiler

The 6 phases of a compiler are:

1. Lexical Analysis
2. Syntactic Analysis or Parsing
3. Semantic Analysis
4. Intermediate Code Generation
5. Code Optimization
6. Code Generation



# CHAPTER 2: Lexical analysis

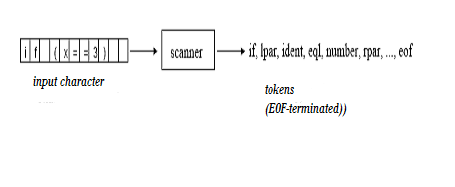
In a compiler, the program that perform lexical analysis is called the scanner.

## Task of a scanner.

The first phase of a compiler is called lexical analysis or scanning. The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes. For each lexeme, the lexical analyzer produces as output a token of the form:

<token-name, attribute-value>

that it passes on to the subsequent phase, syntax analysis. In the token, the first component token-name is an abstract symbol that is used during syntax analysis, and the second component attribute-value points to an entry in the symbol table for this token. Information from the symbol-table entry is needed for sematic analysis and code generation. For example:

****

## DATA STRUCTURE IN KPL

* Only use unsigned integer
* Identifier is made with a combination of lowercase or uppercase letters, digits. An identifier must start with a letter. The length <= 15
* Only allows character constants. A character constant is enclosed with pair of single quote marks. ‘”
* The language do not use string constant.
* “-“ is use for subtraction only. The language does not allow unary minus and negative numbers.
* The relational operator “not equal to” is represented by !=.

## Implementation

**void** *skipBlank*() : skip spaces.

**void** *skipComment*() : skip comments.

Token\* *readIdentKeyword*() : read identifiers or keywords, return a pointer of Token type.

Token\* *readNumber*() : read a integer number, return a pointer of Token type.

Token\* *readConstChar*() : read a constant character, return a pointer of Token type.

TokenType *checkKeyword*(char \*string) : check if the string is a keyword, return TOKEN\_NONE if keyword.

Token\* *makeToken*(TokenType tokenType, int lineNo, int colNo) : create a pointer to a token with predefined type and position.

Token\* *getToken*() : read and return a token (can be invalid token: TOKEN\_NONE).

Token\* *getValidToken*() : read and return a valid token.

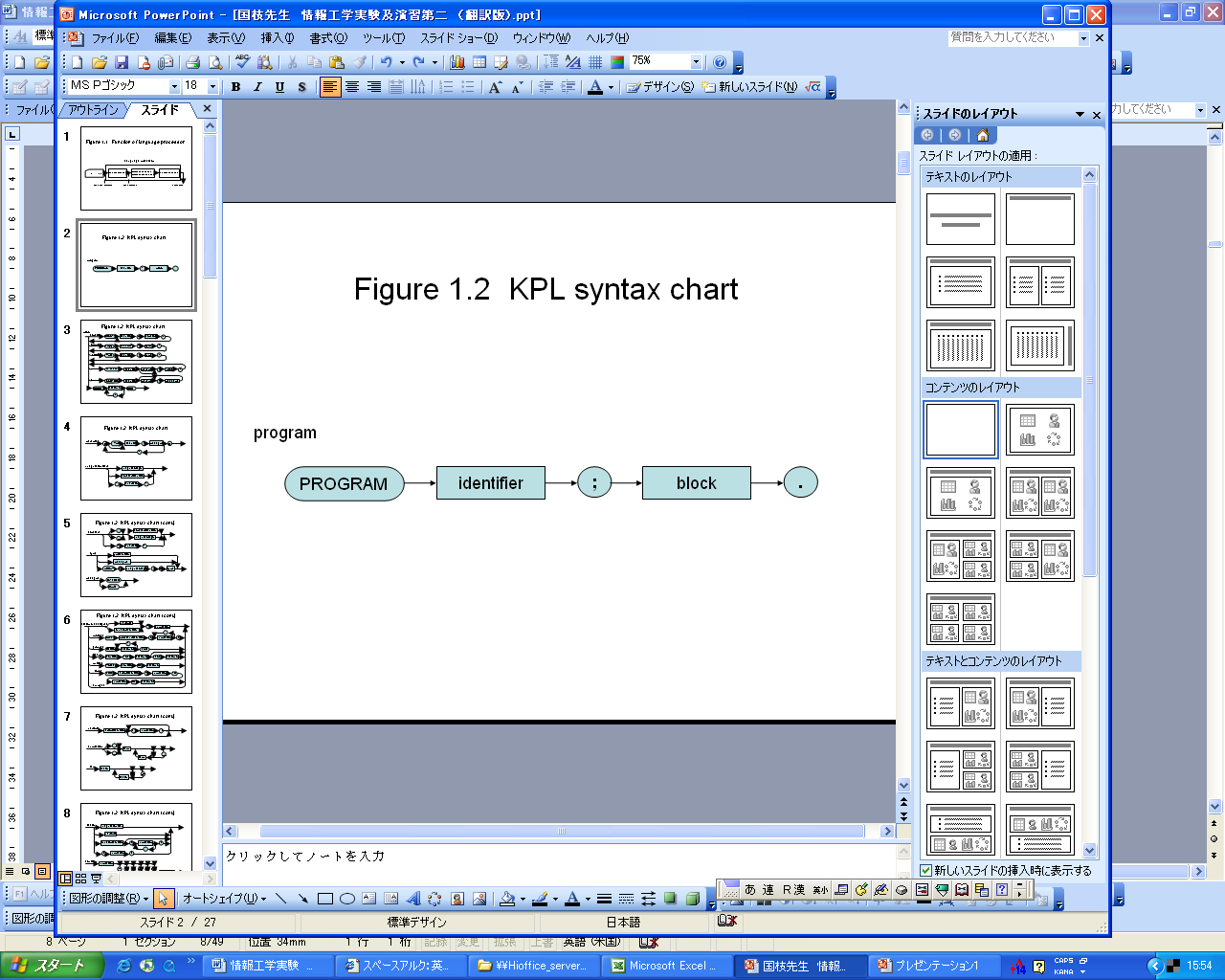
# CHAPTER 3: Syntax analysis

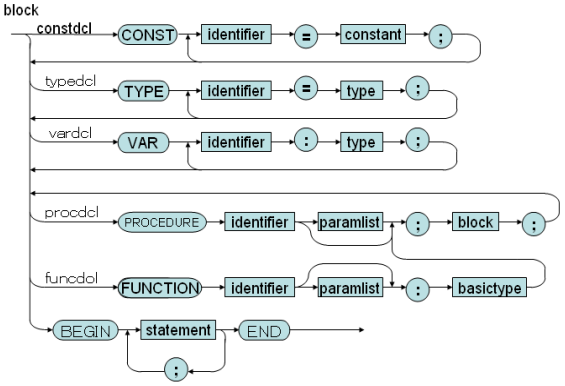
## Role of a parser

- Check the syntax of the program for errors.

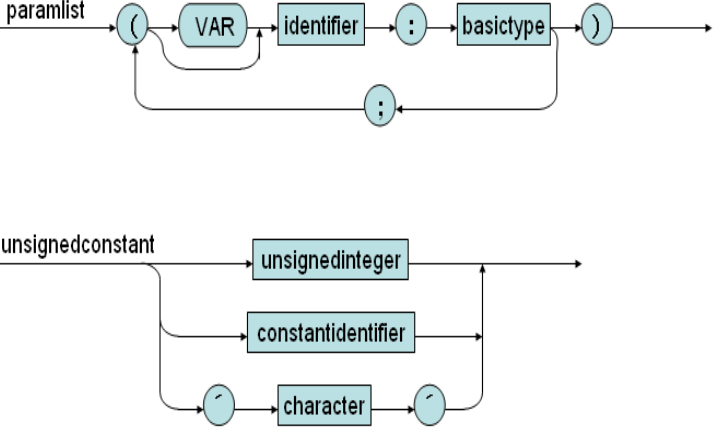
- Produce parse tree for semantic analyser otherwise.

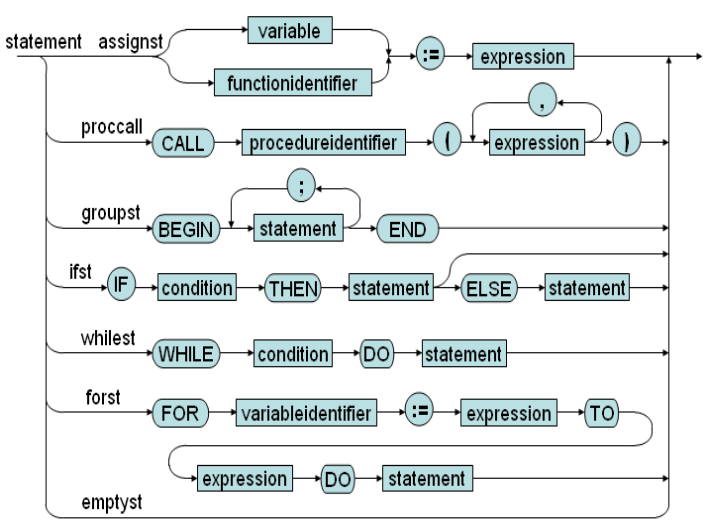
## Syntax directive approach

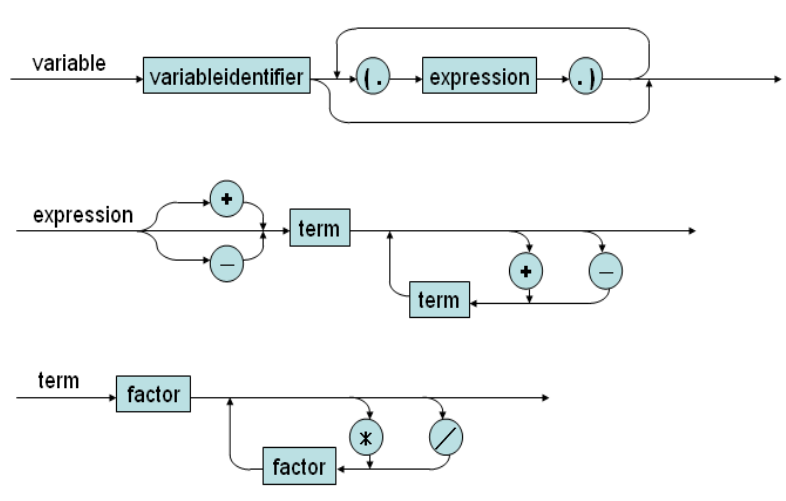


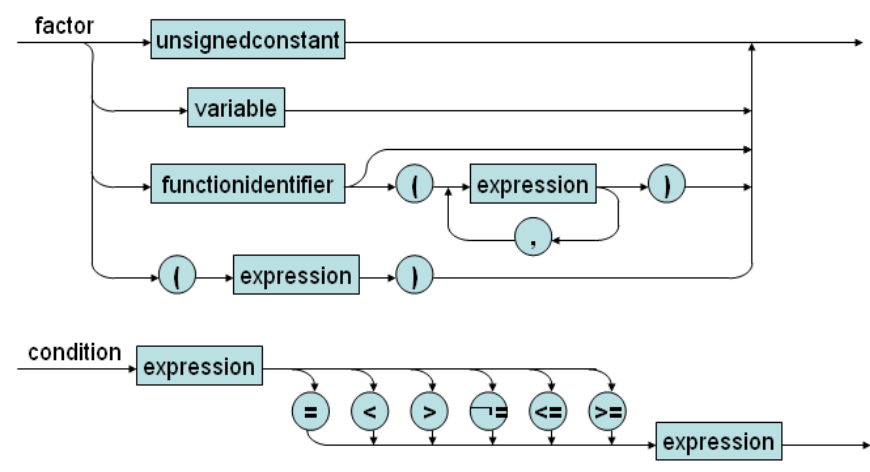


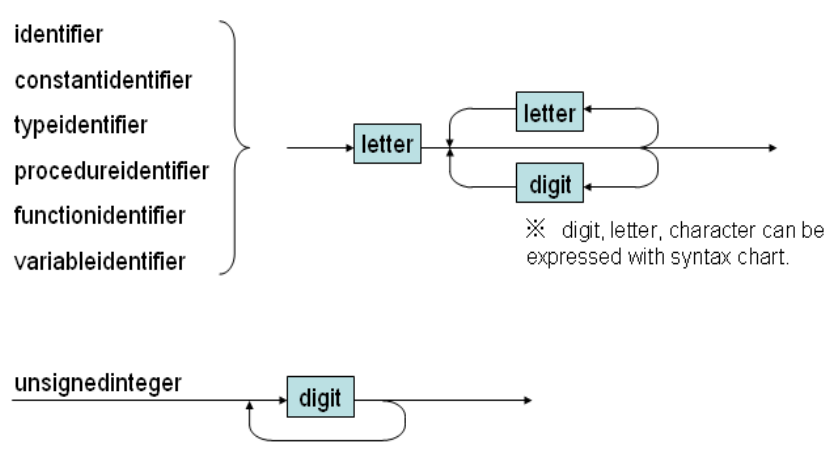
Diagram

Description automatically generated









## Recursive descent parsing

*-Properties:*

+ LL(k) is the language that needs looking ahead k character to produce a valid production.

+used to parse LL(1) language.

+ Can be extended for LL(k), but very complex.

+Used for other grammar can lead to infinite iteration.

-Recursive descent parsing:

- A top-down parsing method

- The term descent refers to the direction in which the parse tree is traversed (or built)

- Use a set of mutually recursive procedures (one procedure for each nonterminal symbol)

* Start the parsing process by calling the procedure that corresponding to the start symbol
* Each production becomes one branch in procedure for its LHS

- We consider a special type of recursive-descent parsing called predictive parsing

## KPL Grammar

**1. *Syntax of BNF***

- A sequence of non-terminal and terminal symbols, is recognized by parsing each symbol in turn

- For each non-terminal symbol, Y, call the corresponding parse function compile.

- For each terminal symbol, y, call a function eat(y) that will check if y is the next symbol in the source program

* The terminal symbols are the token types from the lexical analyzer
* If the variable currentsymbol always contain the next token

eat(y) :

if( currentsymbol == y) then getNextToken() else SyntaxError()

**2. *BNF grammar***

- Construct a grammar G based on syntax diagram

- Perform left recursive elimination (already)

- Perform left factoring

## Implementation

**void** *eat*(TokenType tokenType);

Function will compare the passed tokenType to token type read in scanner (currentToken).

If equals, print out the token.

Otherwise, report error: “missing token” at that position.

**void** *compileProgram*(): parse main program.

**void** *compileBlock*(void): parse constant declarations then call *compileBlock2*.

**void** *compileBlock2*(void): parse type declarations then call *compileBlock3*.

**void** *compileBlock3*(void): parse variable declarations then call *compileBlock4*.

**void** *compileBlock4*(void):parse subroutines declarations then call *compileBlock5*.

**void** *compileBlock5*(void): parse statements in main function.

**void** *compileConstDecls*(void): parse constant declarations.

**void** *compileConstDecl*(void): parse a single constant declaration.

**void** *compileTypeDecls*(void): parse type declarations.

**void** *compileTypeDecl*(void): parse a single type declaration.

**void** *compileVarDecls*(void): parse variable declarations.

**void** *compileVarDecl*(void): parse a variable declaration.

**void** *compileSubDecls*(void): parse subroutines declarations.

**void** *compileFuncDecl*(void): parse function declarations.

**void** *compileProcDecl*(void): parse procedures declarations.

**void** *compileUnsignedConstant*(void): parse unsigned constants.

**void** *compileConstant*(void): parse signed constants.

**void** *compileType*(void): parse a type.

**void** *compileBasicType*(void): parse a basic type.

**void** *compileParams*(void): parse list of parameters.

**void** *compileParam*(void): parse a single parameter.

**void** *compileStatements*(void): parse all statements.

**void** *compileStatement*(void): parse a single statement.

**void** *compileAssignSt*(void): parse an assignment statement.

**void** *compileCallSt*(void): parse a call statement.

**void** *compileIfSt*(void): parse an IF statement.

**void** *compileElseSt*(void): parse an ELSE statement.

**void** *compileWhileSt*(void): parse a WHILE statement.

**void** *compileForSt*(void): parse a FOR statement.

**void** *compileArguments*(void): parse list of arguments passed to a function or procedure.

**void** *compileArguments2*(void): parse list of arguments passed to a function or procedure.

**void** *compileCondition*(void): parse conditional expression.

**void** *compileExpression*(void): parse (+,-) of an expression then call *compileExpression2*

**void** *compileExpression2*(void): parse (+, - ) operators between terms then call *compileExpression3*

**void** *compileExpression3*(void): recursive of (+, -) operators between terms.

**void** *compileTerm*(void): compile a term, which can be composed of (\*, /) of compileFactor

**void** *compileTerm2*(void) recursive procedure of (\*, /) between factors.

**void** *compileFactor*(void): a factor can be a number, character, identifier .

**void** *compileIndexes*(void): parse indexes of an array.

# *CHAPTER 4. Semantic analysis*

## Symbol table

It maintains all declarations and their attributes

* Constants: {name, type, value}
* Types: {name, actual type}
* Variables: {name, type}
* Function: {name, parameters, return type, local declarations}
* Procedures: {name, parameters, local declarations}
* Parameters: {name, type, call by value/call by reference}

- In a KPL compiler, the symbol table is represented as a hierarchical structure

-Symbol table implementation

* Symbol table has currentScope tell current block.
* Update currentScope whenever beginning parsing a procedure/function void enterBlock(Scope\* scope);
* Return currentScope to outer block whenever parsing a procedure/function has been analysed void exitBlock(void);
* Declare a new object in current block

void declareObject(Object\* obj);

## Scope management

**Checking fresh identifier**

We determine if an identifier is not declared yet, by function void checkFreshIdent(char \*name)

void checkFreshIdent(char \*name) {

if (findObject(symtab->currentScope->objList, name) != NULL)

error(ERR\_DUPLICATE\_IDENT, currentToken->lineNo, currentToken->colNo);

}

If the identifier is already declared, findObject function will return an non-null value.

**Checking declared identifier**

Object\* checkDeclaredIdent(char \*name): check declared identifiers: (identifier is already declared or not. If declared return identifier object, else return null.)

Object\* checkDeclaredConstant(char \*name): check declared constants.

Object\* checkDeclaredType(char \*name): check declared identifiers.

Object\* checkDeclaredVariable(char \*name): check declared variables.

Object\* checkDeclaredFunction(char \*name): check declared functions.

Object\* checkDeclaredProcedure(char \*name): check declared procedure.

Object\* checkDeclaredLValueIdent(char \*name): check declared LValue.

## Type checking

void checkIntType(Type\* type): check if type is integer

void checkCharType(Type\* type): check if type is character

void checkArrayType(Type\* type): check if type is array type.

void checkBasicType(Type\* type): check if type is basic type.

void checkTypeEquality(Type\* type1, Type\* type2): check for equality of types, if not, report an error message.