Contents

[**CHAPTER 1: AN OVERVIEW OF COMPILER** 2](#_Toc29062739)

[**1.** **Definiton of compiler** 2](#_Toc29062740)

[**2.** **Components of a compiler** 2](#_Toc29062741)

[**3.** **Main phases of compiling process** 2](#_Toc29062742)

[**3.1** **. Lexical Analyzer - Scanner** 3](#_Toc29062743)

[**3.2** **. Syntax Analyzer - Parser** 3](#_Toc29062744)

[**3.3** **. Semantic Analyzer** 3](#_Toc29062745)

[**3.4** **. Intermediate Code Generation** 3](#_Toc29062746)

[**3.5** **. Code optimization** 3](#_Toc29062747)

[**CHAPTER 2: DESIGN A LEXICAL ANALYZER FOR KPL** 4](#_Toc29062748)

[**1.** **Tasks of Lexical Analyzer (Scanner)** 4](#_Toc29062749)

[**2.** **Main functions** 5](#_Toc29062750)

[**CHAPTER 3: DESIGN SYNTACTIC ANALYZER FOR KPL** 6](#_Toc29062751)

[**1.** **Tasks of Syntactic Analyzer (Parser)** 6](#_Toc29062752)

[**2.** **Design of Syntactic Analyzer** 6](#_Toc29062753)

[**3.** **Main functions** 7](#_Toc29062754)

[**CHAPTER 4: DESIGN SEMANTIC ANALYZER FOR KPL** 8](#_Toc29062755)

[**1.** **Tasks of Semantic Analyzer** 8](#_Toc29062756)

[**2.** **Data Structure of Semantic Analyzer** 8](#_Toc29062757)

[**3.** **Main functions** 9](#_Toc29062758)

[**REFERENCES** 10](#_Toc29062759)

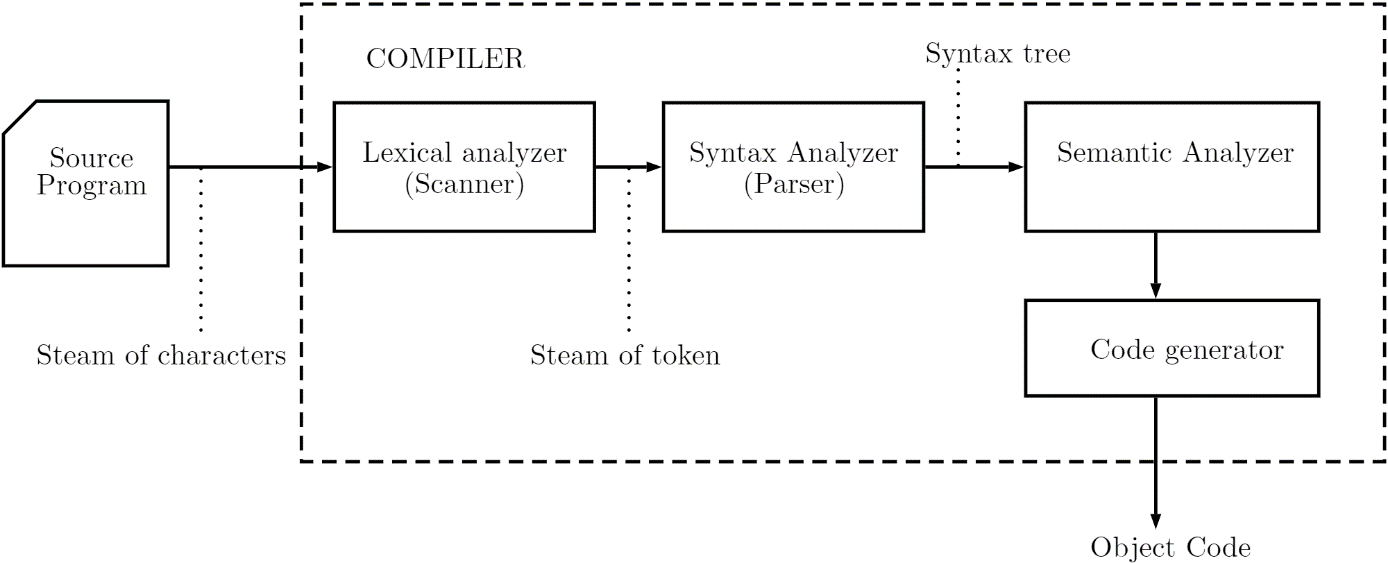
**CHAPTER 1: AN OVERVIEW OF COMPILER**

1. **Definiton of compiler**

Compiler is a computer program (or set of programs) that transforms source code written in a high level language in to the target language, often having a binary form known as object code. Typically, a programmer writes language statements in a language such as Pascal or C one line at a time using an editor.

1. **Components of a compiler**

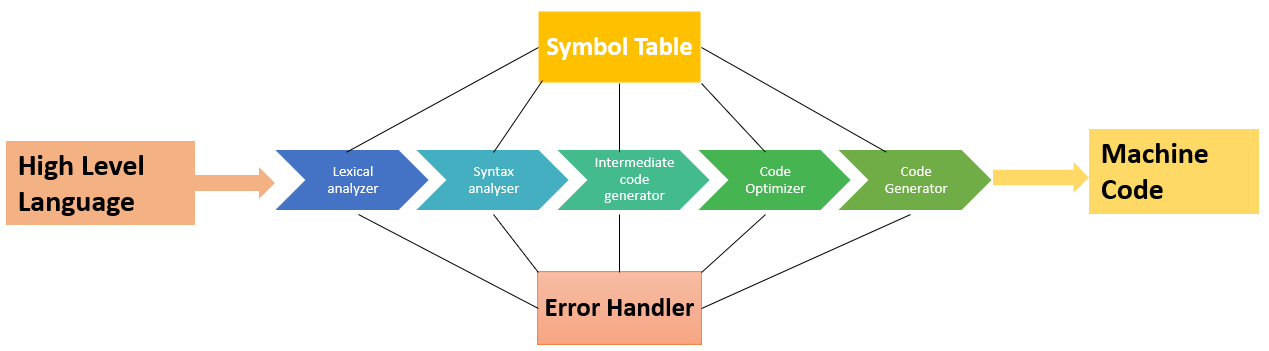
Simple Compiler has 4 main part as figure below:



1. **Main phases of compiling process**

To describe more precisely, a compiler is divided into several interrelated processes, in each process, source program is translated from a specific form to another form of representation.

A typical decompositions is illustrated in this figure:



* 1. **. Lexical Analyzer - Scanner**

Lexical Analyzer (Scanner) converts the stream of input characters into a stream of tokens, i.e: meaningful character strings. The process of lexical analyzer will occur as follows: the scanner will read character - by - character input stream to generate tokens.

* 1. **. Syntax Analyzer - Parser**

Syntax analyzer is the second phase of a compiler, it analyzes a string of tokens, conforming to the rules of a formal grammar or not. The output of this phase is a parse tree, or error. Parsing is based on grammar provided to build parse tree. The most important part of building a compiler is the task of building a grammar that generates structure of a program and cannot be ambiguous. An ambiguous grammar will produce more than one parse tree, therefore must be forbidden.

* 1. **. Semantic Analyzer**

Semantic analyzer is the third phase of a compiler, it is the phase in which the compiler adds semantic information to the parse tree and builds the symbol table. This phase performs semantic checks such as type checking (checking for type errors), or object binding (associating variable and function references with their definitions), or definite assignment (requiring all local variables to be initialized before use), rejecting incorrect programs or issuing warnings.

* 1. **. Intermediate Code Generation**

After the phase of semantic analysis, some compiler will generate an intermediate representation of source program, known as intermediate code. We can consider this representation as a program for an abstract virtual machine. They have two important properties: easy to generate and easy to translate into object code. Moreover, intermediate code is machine-independent. Usually, compiler use three-address codes.

* 1. **. Code optimization**

In this phase, code optimizer will try to optimize the intermediate code into equivalent one with faster execution then we got the final code.

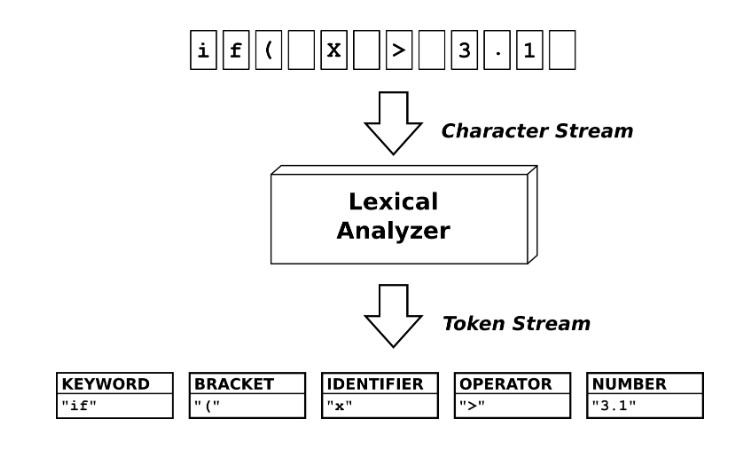
# **CHAPTER 2: DESIGN A LEXICAL ANALYZER FOR KPL**

1. **Tasks of Lexical Analyzer (Scanner)**

There are three main tasks which scanner is responsible for:

* Neglect meaningless character: space, tabulalor, EOF, CR, LF, comments.
* Detect invalid symbols: @, ! (stand-alone), etc. Whenever the lexical analyzer finds an invalid token, it generates an error.
* Recognize different types of token:
* Identifier
* Keyword
* Number
* Special character
* …
* Pass recognized tokens to the parser to perform job of syntactic analysis.

For example:

****

1. **Main functions**

* *void skipBlank()*: skip characters: blank character, tab, new line.
* *void skipComment()*: skip comment ( string that start by *(\** and end by *\*)* )
* *Token \*readIdentKeyword(void)*: if the first character of string is alphabet, it can be a identifier or keyword.
* *Token \*readNumber(void)*: if the first character of string is number, it is a number.
* *Token \*readConstChar(void)*: if the first character of string is singlequote, it is a character
* *Token \*getToken(void)*: read characters from source code and use these functions to create a new token.

In order to recognize different types of token, we define a structure:

typedef enum {

TK\_NONE, TK\_IDENT, TK\_NUMBER, TK\_CHAR, TK\_EOF,

KW\_PROGRAM, KW\_CONST, KW\_TYPE, KW\_VAR,

KW\_INTEGER, KW\_CHAR, KW\_ARRAY, KW\_OF,

KW\_FUNCTION, KW\_PROCEDURE,

KW\_BEGIN, KW\_END, KW\_CALL,

KW\_IF, KW\_THEN, KW\_ELSE,

KW\_WHILE, KW\_DO, KW\_FOR, KW\_TO,

SB\_SEMICOLON, SB\_COLON, SB\_PERIOD, SB\_COMMA,

SB\_ASSIGN, SB\_EQ, SB\_NEQ, SB\_LT, SB\_LE, SB\_GT, SB\_GE,

SB\_PLUS, SB\_MINUS, SB\_TIMES, SB\_SLASH,

SB\_LPAR, SB\_RPAR, SB\_LSEL, SB\_RSEL

} TokenType;

typedef struct {

char string[MAX\_IDENT\_LEN + 1];

int lineNo, colNo;

TokenType tokenType;

int value;

} Token;

One of the most difficult problems in this Scanner is skip comment:

void skipComment()

{

while (1)

{

//Read next character

readChar();

if (currentChar == -1)

{ // End of file

error(ERR\_ENDOFCOMMENT, lineNo, colNo); // comment end with EOF

}

else if (charCodes[currentChar] == CHAR\_TIMES)

{ //If next is asteric character

readChar(); //get next character

}

// Note: (\* then continue to get next character until meet next if

if (currentChar == -1)

{ // End of file

error(ERR\_ENDOFCOMMENT, lineNo, colNo); // comment end with EOF

}

else if (charCodes[currentChar] == CHAR\_RPAR)

{ // Next is right parathesis

// End of comment

readChar();

return;

}

}

}

# **CHAPTER 3: DESIGN SYNTACTIC ANALYZER FOR KPL**

1. **Tasks of Syntactic Analyzer (Parser)**

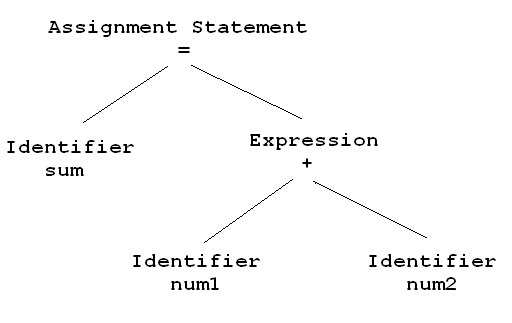
* Take the input from a lexical analyzer in form of token streams.
* Check the syntax of the program for errors.
* Produce parse tree for semantic analyzer.

1. **Design of Syntactic Analyzer**

**-** Use top-down parsing:

* Token LookAhead.
* Parsing terminal symbol.
* Parsing non-terminal symbol.

**-** It define some syntax diagram and BNF grammar.



1. **Main functions**

* Data Structure used in parser

Token \*currentToken;

Token \*lookAhead;

* Scan function use for changing *currentToken* to *lookAhead* and get new *lookAhead* token.

void scan(void) {

Token\* tmp = currentToken;

currentToken = lookAhead;

lookAhead = getValidToken();

free(tmp);

}

* Eat function use for check currentToken if currentToken is expected token type then invoke scan functio else it will has an error.

void eat(TokenType tokenType) {

if (lookAhead->tokenType == tokenType) {

printToken(lookAhead);

scan();

} else missingToken(tokenType, lookAhead->lineNo, lookAhead->colNo);

}

# **CHAPTER 4: DESIGN SEMANTIC ANALYZER FOR KPL**

1. **Tasks of Semantic Analyzer**

* Semantic helps interpret symbols, types and the relationship with each other. It used to check if the syntax structure constructed of a program derived meaning or not.
* It contains 3 tasks:

- Produces symbol table for future references (eg. scope & type checking).

- Scope checking.

- Type checking.

1. **Data Structure of Semantic Analyzer**

* Data Structure of Symbol Table

struct SymTab\_ {

Object\* program;

Scope\* currentScope;

ObjectNode \*globalObjectList;

};

We need a symbol table to:

- Store information about each object in program such as main program, procedure, function, variable, constant,…

- Store typical attributes for each type: a function must have a parameter list and return type.

* Data Structure of Object

struct Object\_ {

char name[MAX\_IDENT\_LEN];

enum ObjectKind kind;

union {

ConstantAttributes\* constAttrs;

VariableAttributes\* varAttrs;

TypeAttributes\* typeAttrs;

FunctionAttributes\* funcAttrs;

ProcedureAttributes\* procAttrs;

ProgramAttributes\* progAttrs;

ParameterAttributes\* paramAttrs;

};

};

1. **Main functions**

* In order to check type or check kind of Object, we need to find it first.

Object\* lookupObject(char \*name) {

Scope\* scope = symtab->currentScope;

Object\* obj;

while (scope != NULL) {

obj = findObject(scope->objList, name);

if (obj != NULL) return obj;

scope = scope->outer;

}

obj = findObject(symtab->globalObjectList, name);

if (obj != NULL) return obj;

return NULL;

}

* List functions for checking declaration:

void checkFreshIdent(char \*name);

Object\* checkDeclaredIdent(char \*name);

Object\* checkDeclaredConstant(char \*name);

Object\* checkDeclaredType(char \*name);

Object\* checkDeclaredVariable(char \*name);

Object\* checkDeclaredFunction(char \*name);

Object\* checkDeclaredProcedure(char \*name);

Object\* checkDeclaredLValueIdent(char \*name);

We determine if an identifier is not declared yet, by function *void checkFreshIdent (char \*name)*. If the identifier is already declared, *findObject* function will return an non-null value.

* List functions for checking type:

void checkIntType(Type\* type);

void checkCharType(Type\* type);

void checkArrayType(Type\* type);

void checkBasicType(Type\* type);

void checkTypeEquality(Type\* type1, Type\* type2);

Type checker verifies that the type of a construct (constant, variable, array, list, object) matches what is expected in its usage context.

**REFERENCES**

1. **Lecture Slides: Compiler Construction 2019-2020**

*Dr. Nguyen Thi Thu Huong (HUST)*

1. <https://www.wikipedia.org/>
2. <https://www.geeksforgeeks.org/semantic-analysis-in-compiler-design/>
3. <https://www.geeksforgeeks.org/introduction-of-lexical-analysis/>
4. <https://www.geeksforgeeks.org/introduction-to-syntax-analysis-in-compiler-design/>