RESEARCH PAPER ON PHASES OF COMPILER

Charu Arora, Chetna Arora, Monika Jaitwal

***Abstract*- This paper gives a short description about the different phases of the compiler. It describes the compilation process in the introduction part. It also describes the different phases of the compiler in detail. This paper explains about how the source program is compiled to produce the target code by passing through the various phases of thecompiler**

I. INTRODUCTION

A **compiler** is a computer program (or set of programs) that transforms source code written in a programming language (the source language) into another computer language (the target language, often having a binary form known as object code). The most common reason for wanting to transform source code is to create an executable program

II. LEXICAL ANALYSIS

Lexical analysis is the process of converting a sequence of characters into a sequence of tokens. A program or
function which performs lexical analysis is called a **lexical analyzer**, **lexer** or**scanner**. A lexer often exists as a
single function which is called by aparser or another function.It is roughly the equivalent of splitting ordinary text written in a natural language (e.g. English) into a sequence of words and punctuation symbols. Lexical analysis is often done with tools such as lex, flex and jflex.

2.1Tokens, Patterns, and Lexemes

• A token is a pair

– token name (e.g., VARIABLE)

– token value (e.g., "myCounter")

• A lexeme is a sequence of program characters that form a token

– (e.g., "myCounter")

• A pattern is a description of the form that the lexemes of a token may take

– e.g., character strings including A-Z, a-z, 0-9, and _

EXAMPLE:

Typically,

1. Each keyword is a token, e.g, then, begin, integer.

2. Each identier is a token, e.g., a, zap.

3. Each constant is a token, e.g., 123, 123.45, 1.2E3.

4. Each sign is a token, e.g., (, <, <=, +.

Example:

if foo = bar then

consists of tokens:

IF, <ID,1>, EQSIGN, <ID,2>, THEN

Note that we have to distinguish among different occurrences of the token ID.

LA is usually called by the parser as a subroutine, each time a new token is needed.

III. SYNTAX ANALYSIS

This is alternatively known as parsing. It is roughly the equivalent of checking that some ordinary text written in a natural language (e.g. English) is grammatically correct (without worrying about meaning)

Parsing or **syntactic analysis** is the process of analysing a string of symbols, either in natural language or
in computer languages, according to the rules of aformal grammar. The term *parsing* comes from
Latin *pars* (*orationis*), meaning part (of speech).
The term has slightly different meanings in different branches of linguistics and computer science. Traditional
sentence parsing is often performed as a method of understanding the exact meaning of a sentence, sometimes with the aid of devices such as sentence diagrams. It usually emphasizes the importance of grammatical divisions such as subject and predicate.

• SYNTAX consists of rules for constructing “acceptable” utterances.

To determine that a program is syntactically correct, you must determine how the grammatical rules were used to construct it.

– Powerful tools (“parser generators”) are available for generating analysis phases of your compiler, starting from a formal specification of the languages

3.1Syntax Errors

■ Many errors are syntactic or exposed by parsing

■ eg. Unbalanced ()

■ Error handling goals:

■ Report errors quickly & accurately

■ Recover quickly (continue parsing after error)

■ Little overhead on parse time

IV. SEMANTIC ANALYSIS

Parsing only verifies that the program consists of tokens arranged in a syntactically valid combination. Now we'll move forward to semantic analysis, where we delvelop even deeper to check whether they form a sensible set of instructions in the programmig language.

In linguistics, **semantic analysis** is the process of relating syntactic structures, from the levels of phrases,sentences and paragraphs to the level of the writing as a whole, to their language-independent meanings. It also involves removing features specific to particular linguistic and cultural contexts, to the extent that such a project is possible.

The elements of idiom and figurative speech, being cultural, are often also converted into relatively invariant

meanings in semantic analysis.

The purpose of semantic analysis is to check that we have a meaningful sequence of tokens. Note that a sequence can be meaningful without being correct; in most programming languages, the phrase "x + 1" would be considered to be a meaningful arithmetic expression. However, if the programmer really meant to write "x - 1", then it is not correct.

V. INTERMEDIATE CODE

The **Intermediate Code** is the basic structure on which both built-in and user-created translators operate. The names of the functions used in **Intermediate Code** expressions are members of the Code Generation ,Names sub package.

5.1What should the intermediate code look like?

- it should be simple, well-defined
- it should have well-defined *semantics*, i.e., there should be no doubt about what different constructs mean
- it should be independent of both source language and machine language
- but translation from source and to machine should be easy

VI. OPTIMISATION

Optimization is the process of transforming a piece of code to make more efficient (either in terms of time or space) without changing its output or side-effects. The only difference visible to the code's user should be that it runs faster and/or consumes less memory. It is really a misnomer that the name implies you are finding an "optimal" solution— in truth, optimization aims to improve, not perfect, the result.

In computing, an **optimizing compiler** is a compiler that tries to minimize or maximize some attributes of

an executable computer program. The most common requirement is to minimize the time taken to execute

a program; a less common one is to minimize the amount of memory occupied. The growth of portable

computers has created a market for minimizing the power consumed by a program. Compiler optimization is

generally implemented using a sequence of *optimizing transformations*, algorithms which take a program and

transform it to produce a semantically equivalent output program that uses fewer resources

VII. CODE GENERATION

In computing, **code generation** is the process by which a compiler's **code generator** converts some intermediate

representation of source code into a form (e.g., machine code) that can be readily executed by a machine.

The input to the code generator typically consists of a parse tree or an abstract syntax tree. The tree is converted

into a linear sequence of instructions, usually in an intermediate language such as three address code. Further

stages of compilation may or may not be referred to as "code generation", depending on whether they involve a

significant change in the representation of the program. (For example, a peephole optimization pass would not likely be called "code generation", although a code generator might incorporate a peephole optimization pass.

REFERENCES

[1] Muchnick . Advanced Compiler Design and Implementation.

[2] Robert Morgan. Building an Optimizing compiler.

[3] Y.N. Srikant P. Shankar. The Compiler Design Handbook: Optimizations