Experiment No: 1

AIM: To study phases of compiler

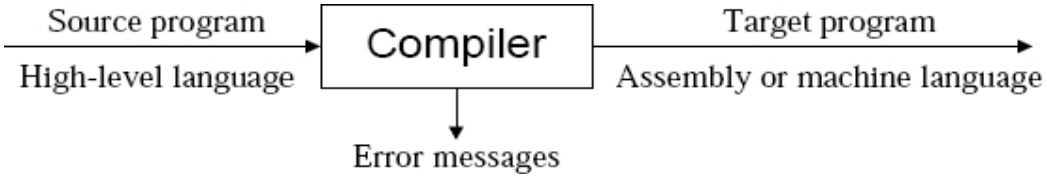
OBJECTIVE

-To study & understand the concept of compiler.

-To study different phases of compiler

THEORY:

In order to reduce the complexity of designing and building computers, nearly all of these are made to execute relatively simple commands. A program for a computer must be build by combining these very simple commands into a program in what is called machine language. Since this is a tedious and error-prone process most programming is, instead, done using a high-level programming language. This language can be very different from the machine language that the computer can execute, so some means of bridging the gap is required. This is where the compiler comes in. A compiler translates (or compiles) a program written in a high-level programming language that is suitable for human programmers into the low-level machine language that is required by computers. During this process, the compiler will also attempt to spot and report obvious programmer mistakes.



Using a high-level language for programming has a large impact on how fastprograms can be developed. The main reasons for this are:

- Compared to machine language, the notation used by programming languages is closer tothe way humans think about problems.
- The compiler can spot some obvious programming mistakes.
- Programs written in a high-level language tend to be shorter than equivalent programswritten in machine language.

Another advantage of using a high-level level language is that the same program can be compiled to many different machine languages and, hence, be brought to run on many different machines. On the other hand, programs that are written in a high-level language and automatically translated to machine language may run somewhat slower than programs that are hand-coded in machine language. Hence, some time-critical programs are still written partly in machine language. A good compiler will, however, be able to get very close to the speed of hand-written machine code when translating well-structured programs.

The Translation Process

A compiler performs two major tasks:

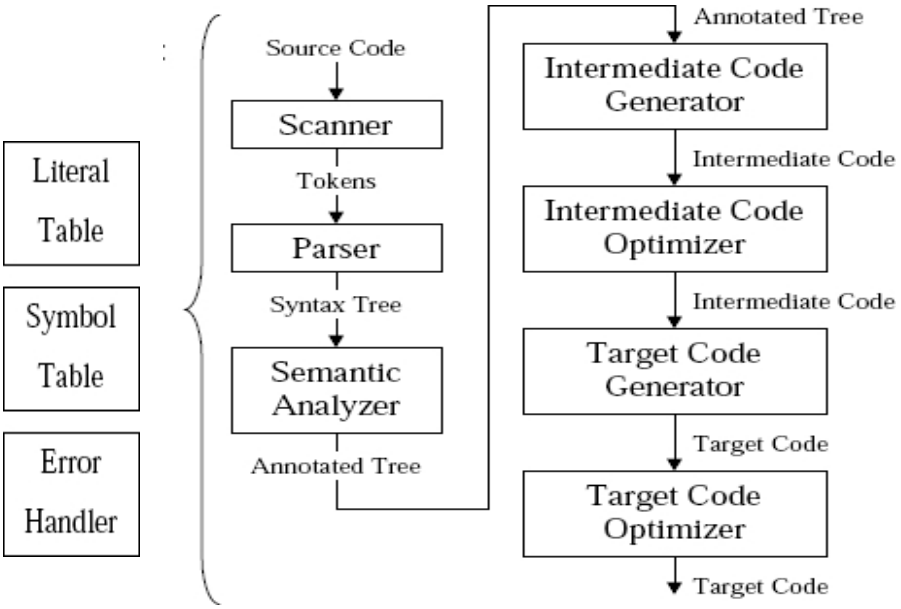
- Analysis of the source program
- Synthesis of the target-language instructions

Phases of a compiler:

Scanning
Parsing
Semantic Analysis
Intermediate Code Generation
Intermediate Code Optimizer
Target Code Generator
Target Code Optimizer

Three auxiliary components interact with some or all phases:

Literal Table
Symbol Table
Error Handler



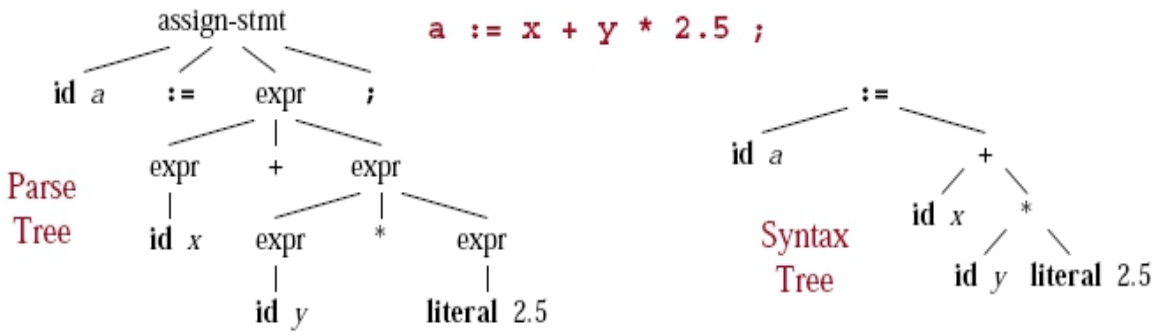
Scanner

The scanner begins the analysis of the source program by:

- Reading file character by character
- Grouping characters into tokens
- Eliminating unneeded information (comments and white space)
- Entering preliminary information into literal or symbol tables
- Processing compiler directives by setting flags
- Tokens represent basic program entities such as:
- Identifiers, Literals, Reserved Words, Operators, Delimiters, etc.
- Example: **a := x + y * 2.5 ;** is scanned as **a** identifier **y** identifier **:=** assignmentoperator ***** multiplication operator **x** identifier **2.5** real literal **+** plus operator **;** semicolon

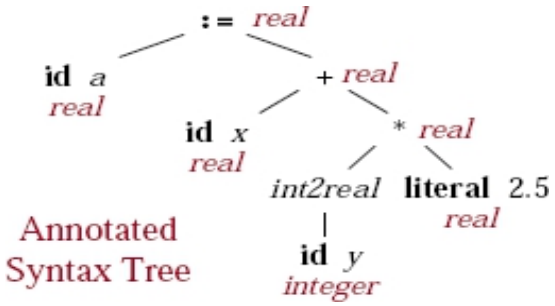
Parser

- Receives tokens from the scanner
- Recognizes the structure of the program as a **parse tree**
- Parse tree is recognized according to a context-free grammar
- Syntax errors are reported if the program is syntactically incorrect
- A parse tree is inefficient to represent the structure of a program
- A **syntax tree** is a more condensed version of the parse tree
- A syntax tree is usually generated as output by the parser



Semantic Analyzer

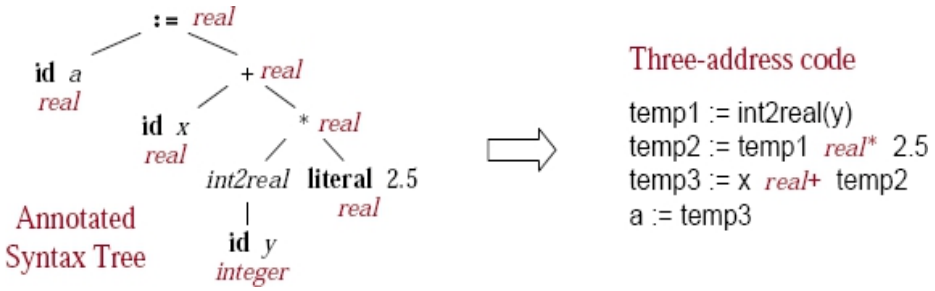
- The semantics of a program are its **meaning** as opposed to syntax or structure
- The semantics consist of:
- **Runtime semantics** – behavior of program at runtime
- **Static semantics** – checked by the compiler
- Static semantics include:
- Declarations of variables and constants before use
- Calling functions that exist (predefined in a library or defined by the user)
- Passing parameters properly
- Type checking.
- Static semantics are difficult to check by the parser
- The semantic analyzer does the following:
- Checks the static semantics of the language
- Annotates the syntax tree with type information



Intermediate Code Generator

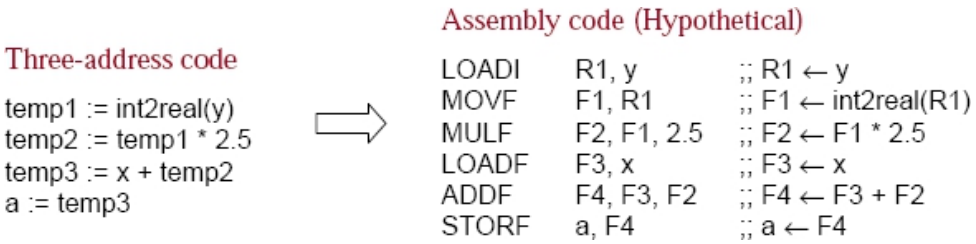
- Comes after syntax and semantic analysis
- Separates the compiler front end from its backend

- Intermediate representation should have 2 important properties:
- Should be easy to produce
- Should be easy to translate into the target program
- Intermediate representation can have a variety of forms:
- Three-address code, P-code for an abstract machine, Tree or DAG representation



Code Generator

- Generates code for the target machine, typically:
- Assembly code, or
- Relocatable machine code
- Properties of the target machine become a major factor
- Code generator selects appropriate machine instructions
- Allocates memory locations for variables
- Allocates registers for intermediate computations



Code Improvement

- Code improvement techniques can be applied to:
- Intermediate code – independent of the target machine
- Target code – dependent on the target machine
- Intermediate code improvement include:
- Constant folding
- Elimination of common sub-expressions
- Identification and elimination of unreachable code (called dead code)
- Improving loops
- Improving function calls
- Target code improvement include:
- Allocation and use of registers
- Selection of better (faster) instructions and addressing modes

Conclusion: Thus we have studied and understood the concept of compiler and phases ofcompiler.