BHCS15B: System Programming

Introduction to Compilers

Mahesh Kumar
(maheshkumar@andc.du.ac.in)

Course Web Page
(www.mkbhandari.com/mkwiki)

Outline

1. Overview of Compilation

2. Phases of Compilation

Introduction

- Programming languages are *notations for describing computations* to people and to machines.

- All the software running on all the computers are *written in some programming language*.

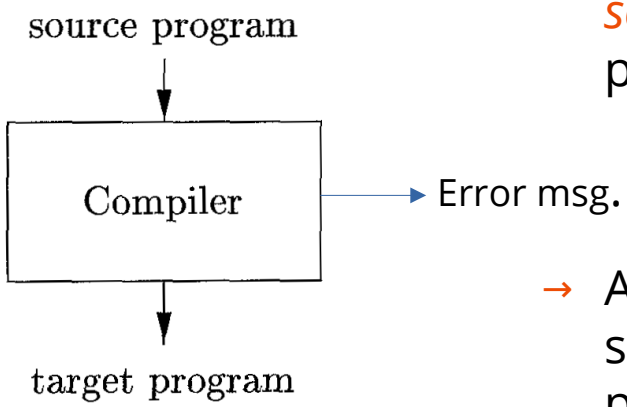
  → *The world depends on programming languages.*

- A program must be *translated into a form* in which it can be executed by a computer.

  → *The software systems that do this translation are called* ***compilers*****.**

Language Processors

- Compiler

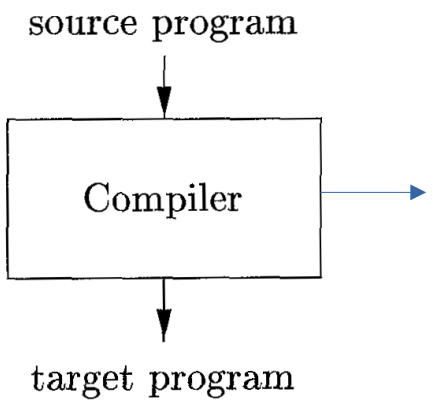


→ A program that can read a program in one language (*the source language*) and translates it into an equivalent program in another language (*the target language*).

→ An important role of a compiler is to *report errors* in the source program that it detects during the translation process.

Language Processors

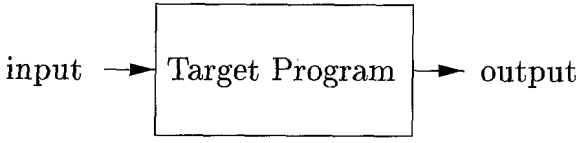
- Compiler



→ A program that can read a program in one language (*the source language*) and translates it into an equivalent program in another language (*the target language*).

Error msg.

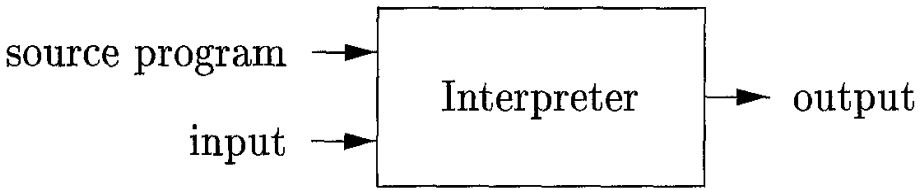
→ An important role of a compiler is to *report errors* in the source program that it detects during the translation process.



→ If the target program is an *executable machine-language program*, it can then be called by the user to process input and produce outputs.

Language Processors (2)

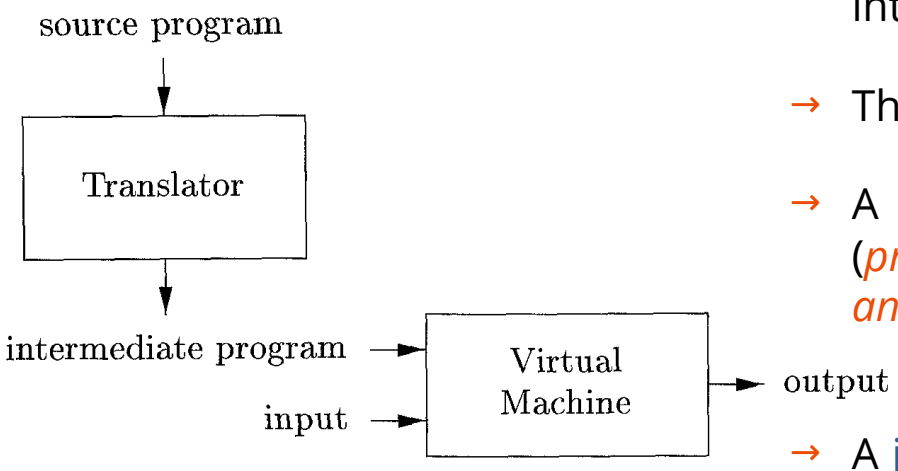
- Interpreter
  - → It is another common kind of language processor.
  - → It appears to *directly execute the operations* specified in the source program on inputs supplied by the user



  - → A compiler is usually *much faster* than an interpreter at mapping inputs to outputs.
  - → An interpreter gives *better error diagnostics* than a compiler, because it executes the source program statement by statement.

Language Processors (3)

- A Hybrid Compiler



→ Java language processors combine compilation and interpretation

→ A Java source program are first be compiled into an intermediate form called bytecodes.

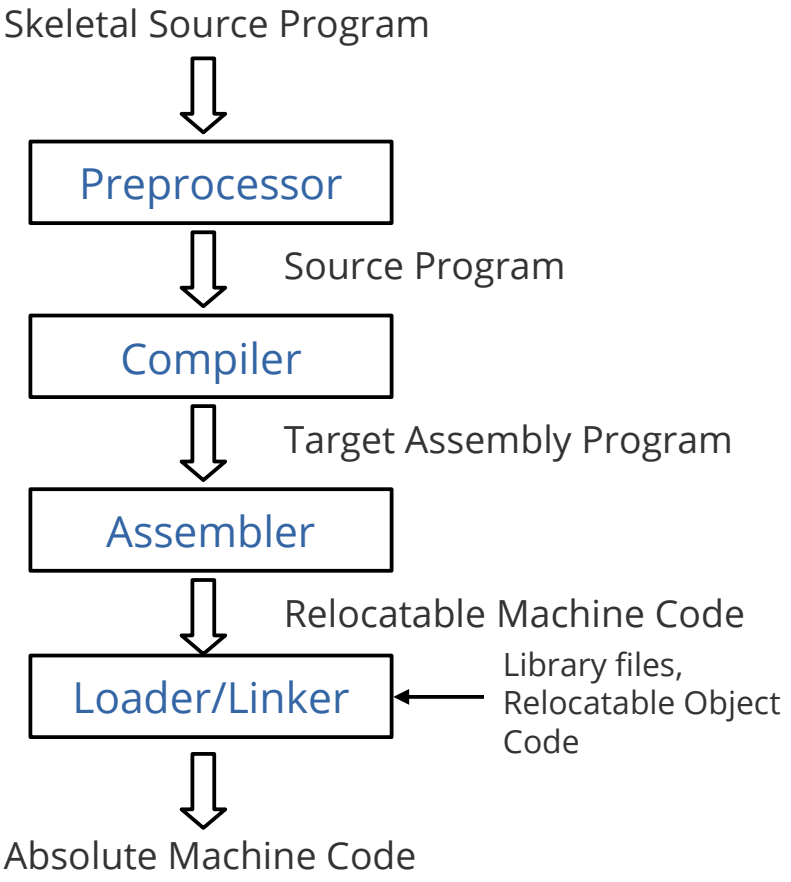
→ The *bytecodes* are then interpreted by a virtual machine.

→ A benefit of bytecodes is it is plateform independent (*program compiled on one machine can be interpreted on another machine, perhaps across a network*).

→ A just-in-time compilers in JVM helps to to achieve faster processing of inputs to outputs

→ *Just-in-time compilers*, translate the *bytecodes* into *machine language* immediately before they run the intermediate program to process the input.

Language Processors (4)



The structure of a Compiler

- So far, we have treated a compiler as a single box (unit) that maps a source program into semantically equivalent target program.

- It can be broadly divided into two parts:

  1. *Analysis (Front end of compiler)*
  2. *Synthesis (Back end of compiler)*

The structure of a Compiler (2)

- ***Analysis* part:** breaks up the source program into constituent pieces and imposes a grammatical structure on them.

- Grammatical structure is then used to create an intermediate representation of the source program.

- If it detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so the user can take corrective action.

- It also collects information about the source program and stores it in the symbol table, which is passed along with the intermediate representation to the ***synthesis part***.

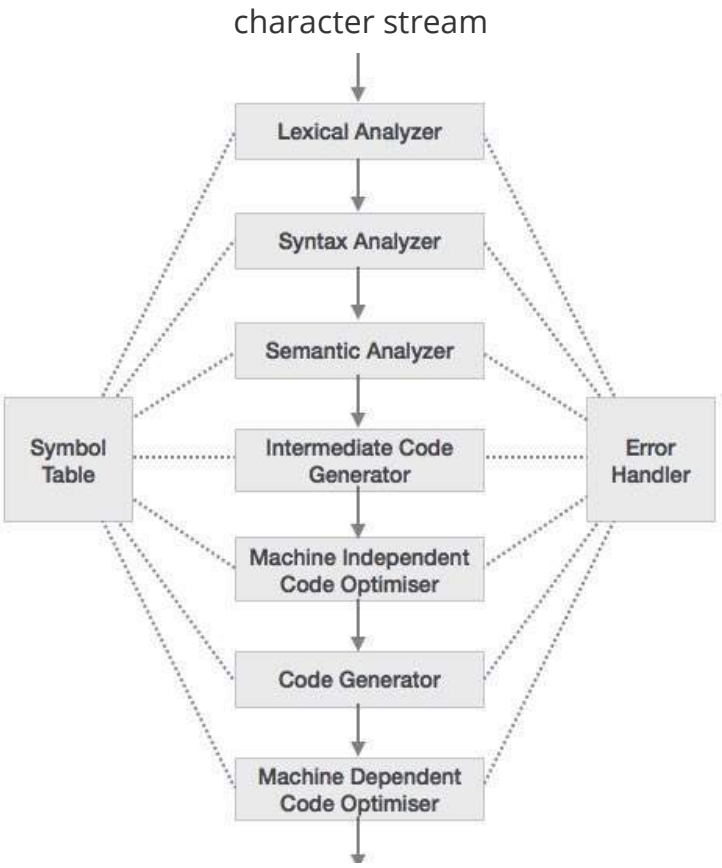
The structure of a Compiler (3)

- ***Synthesis part*:** constructs the desired target program from the intermediate representation and the information in the symbol table.

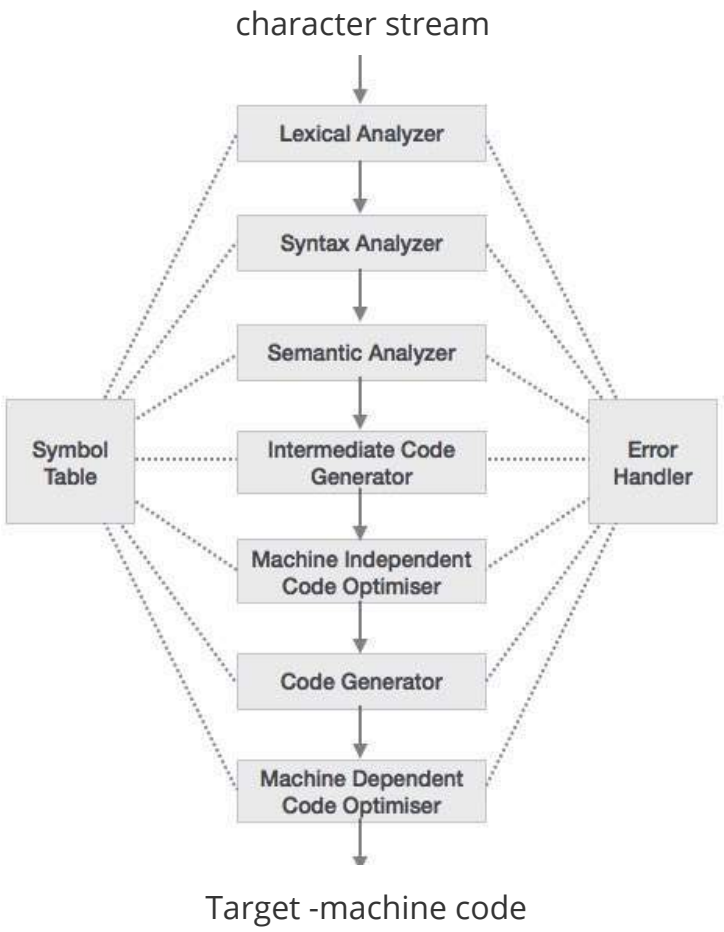
- ***Compilation process**:* operates as a sequence of phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler.

- A typical decomposition of a compiler into phases is shown in next slide ->>

Phases of a Compiler



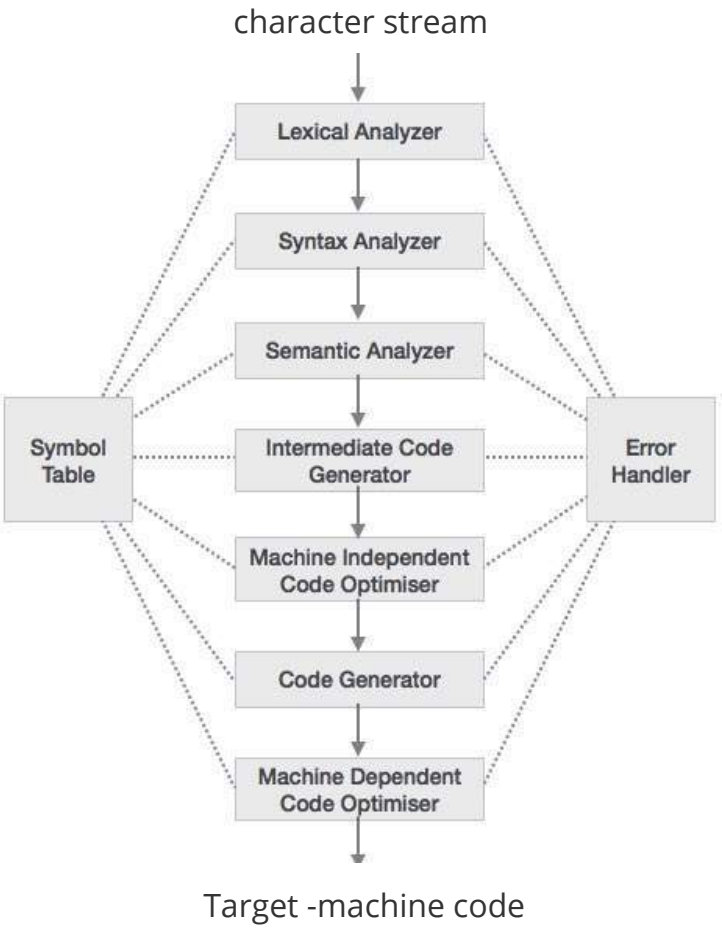
Phases of a Compiler



- *Lexical Analysis (Scanning)*:
  - → The *lexical analyzer* reads the stream of characters (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)
  - → *Tokens* are then passed on to the next phase, i.e., *syntax analysis*.
  - → ***Please see the example in Translation phase*.jpg.**

Phases of a Compiler



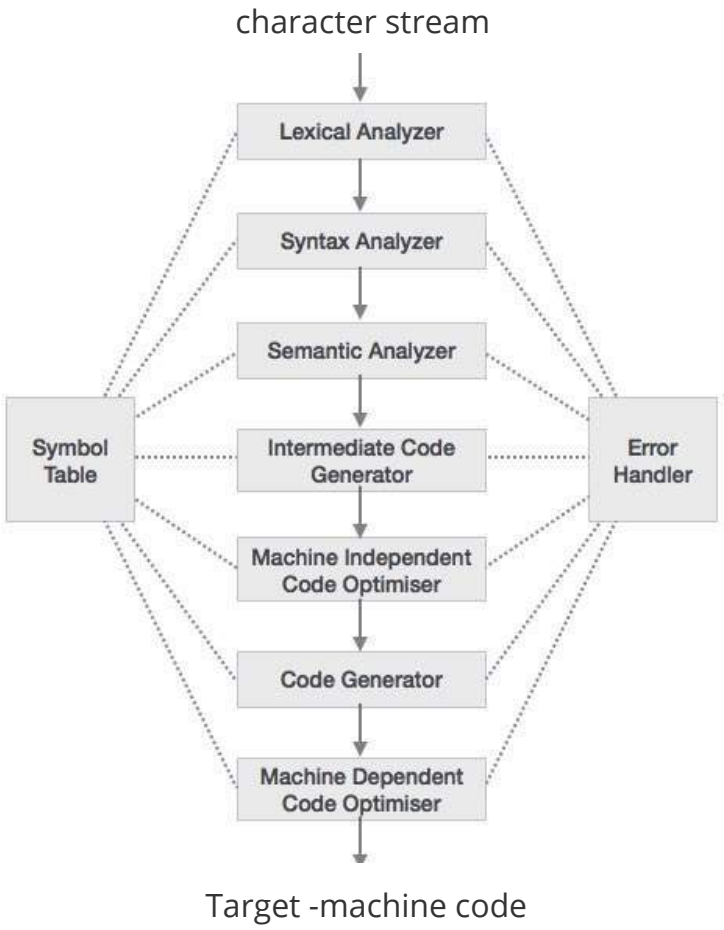
- *Syntax Analysis (Parsing)*:

  → The parser uses tokens produced by the *lexical analyzer* to create a *syntax tree* (an **IR** that depicts the grammatical structure of the token stream)

  → In *syntax tree* each interior node represents an operation and the children of the node represents the arguments of the operation.

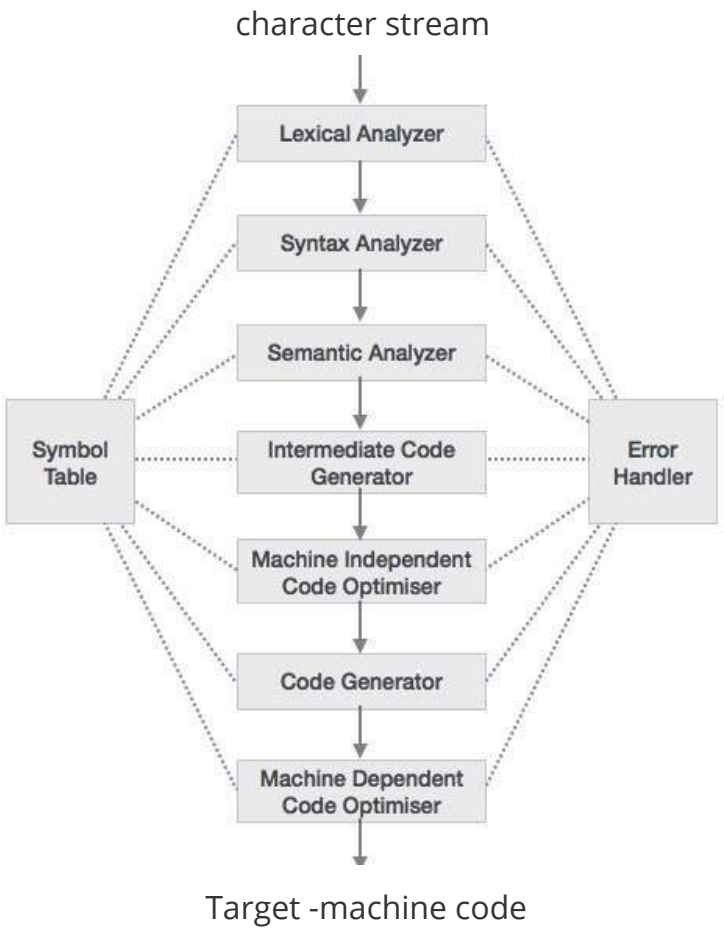
  → ***Please see the example in Translation phase*.jpg.**

Phases of a Compiler



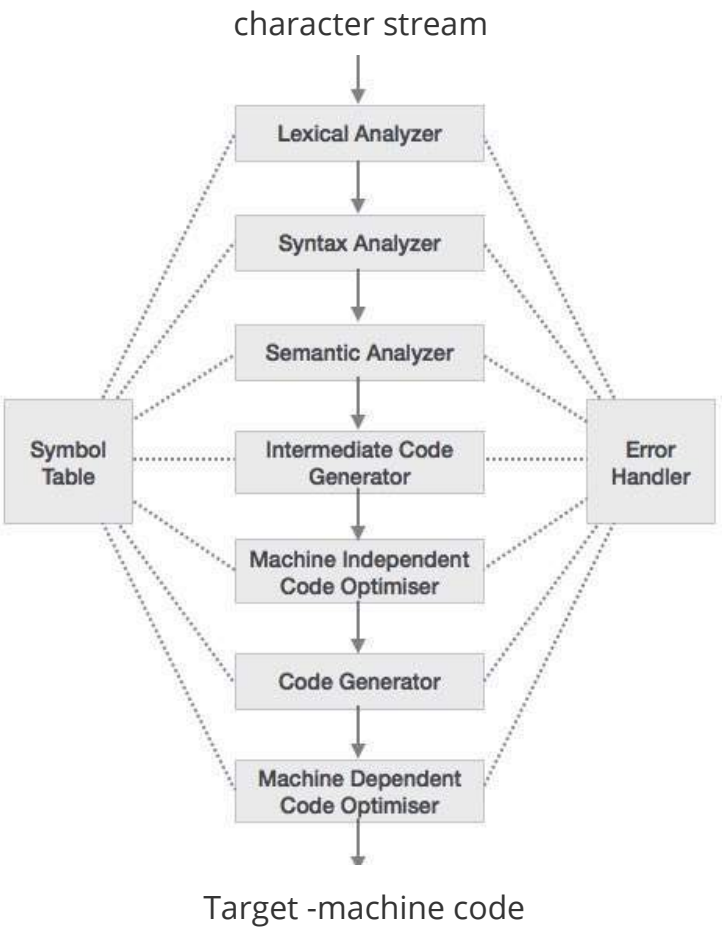
- *Semantic Analysis*:
  - → The semantic analyzer uses the *syntax tree* and the information in the symbol table to check the source program for semantic consistency with the language definition.
  - → It also gathers type information and saves it either in the syntax tree or the symbol table, for subsequent use during intermediate-code generation.
  - → *Type checking* is also done in this phase, where each operator is checked for matching operands.
  - → ***Please see the example in Translation phase*.jpg.**

Phases of a Compiler



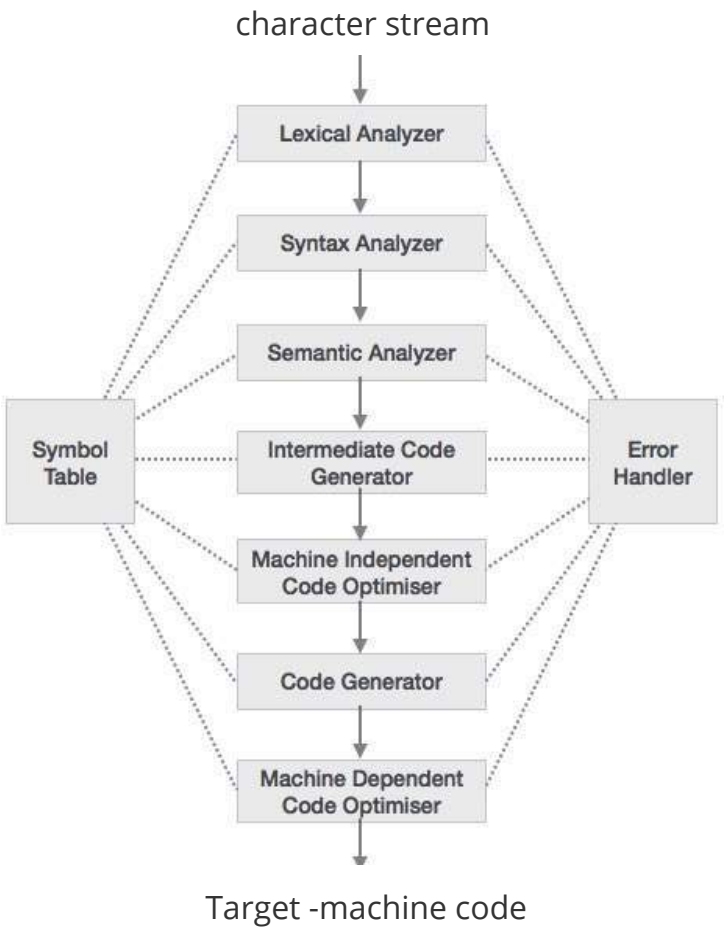
- *Intermediate Code Generation*:
  - → During compilation process, a compiler may construct one or more intermediate representations (IRs), which can have a variety of forms.
  - → Syntax trees are a form of **IR,** commonly used during syntax and semantic analysis.
  - → After syntax and semantic analysis, many compilers generate an explicit low-level or machine-like **IR** (program for an abstract machine).
  - → This **IR** should be easy to produce and easy to translate into the target machine**.**
  - → ***Please see the example in Translation phase*.jpg.**

Phases of a Compiler



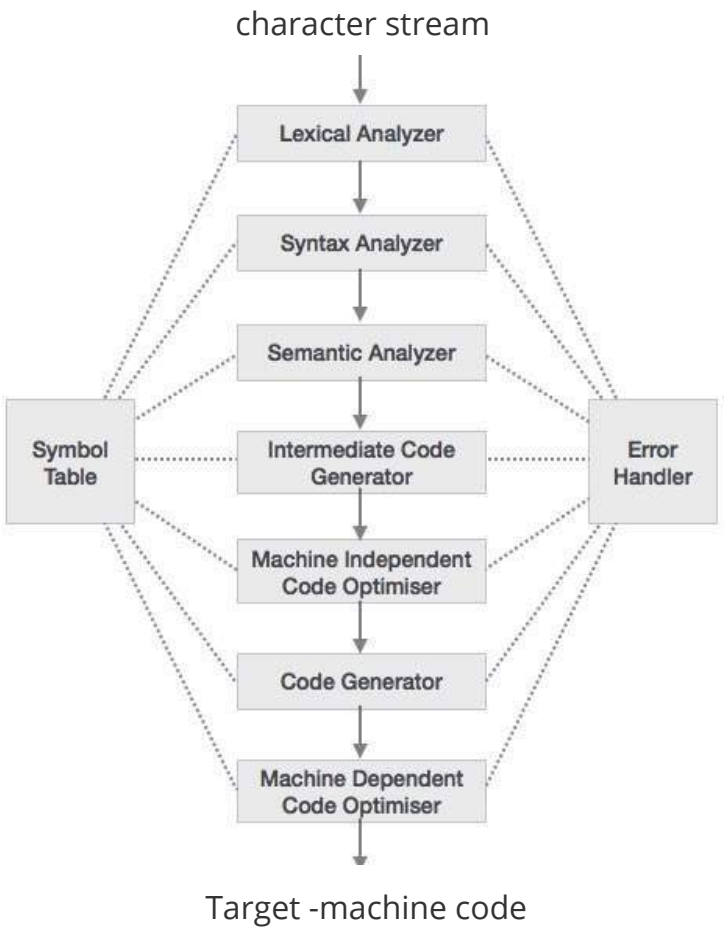
- *Code Optimization*:
  - → The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result.
    - faster
    - shorter
  - → A simple intermediate code generation algorithm followed by code optimization is a reasonable way to generate good target code.
  - → ***Please see the example in Translation phase*.jpg.**

Phases of a Compiler



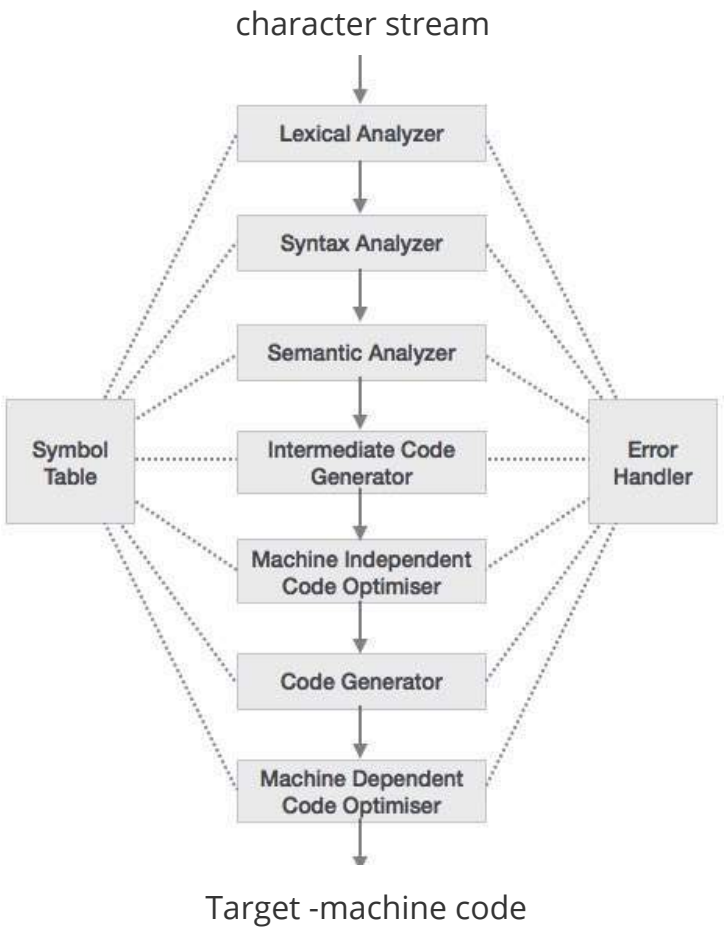
- *Code Generation*:
  - → The code generator takes as input an **IR** of the source program and maps it into the target language.
  - → If target language is machine code, then registers or memory locations are selected for each variables used by the program.
  - → Then, the intermediate instructions are translated into sequence of machine instruction
  - → Judicious assignment of the registers is the crucial aspect of code generation.
  - → ***Please see the example in Translation phase*.jpg.**

Phases of a Compiler



- Symbol Table Management
  - → Records the variable names used in source program and collect information about their attributes:
    - Storage allocation, type, scope
  - → In case of procedures names:
    - Number and type of arguments
    - Pass by value or pass by reference
    - Type returned
  - → It should be designed such that, the store and retrieve operations are quick.
  - → ***Please see the example in Translation phase*.jpg.**

Phases of a Compiler



- The grouping of Phases into Passes
  - → In the implementation of a compiler, activities from several **phases** may be grouped together into a **pass** that reads an input file and writes an output file.
  - → **Front-end phases** (lexical analysis, syntax analysis, semantic analysis, intermediate code generation) are grouped together into **one pass**:
  - → **Code optimization** might be an **optional pass**.
  - → **Back-end pass** consisting of code generation for a particular target machine.

References

Reference for this topic

- **Book:** Alfred V. Aho, Monica S. Lam, Ravi Sethi, J D Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd Edition, Prentice Hall, 2006.

- **Web:** *CS143 Compilers*, Lecture 1, Stanford University.
  [https://web.stanford.edu/class/cs143/]

- **Web**: Tutorialspoint.com, Compiler Design - Phases of Compiler
  [www.tutorialspoint.com/compiler_design/compiler_design_phases_of_compiler.htm]