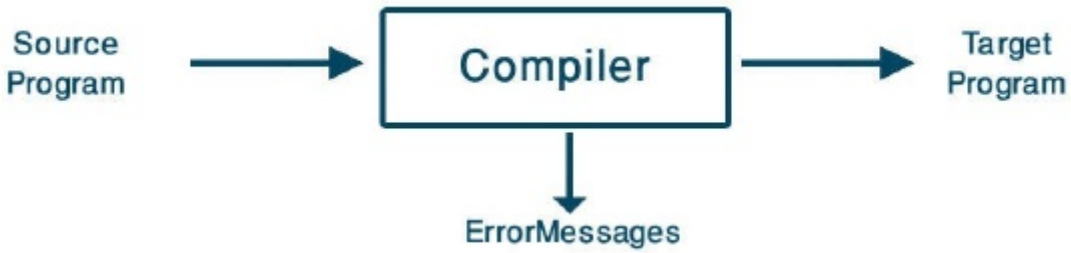
# Compiler Design

Module - 1

**Dr. Mousumi Dutt**

# Compiler Design

- 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.




- Compiler is a program
- 1000 to 1000000 lines of code
- Writing the whole is a big task
- Must have basic prerequisite knowledge

# Conceptual Roadmap

- To translate text from one language to another, the tool must understand both the **form, or syntax, and content, or meaning**, of the input language
- It needs to understand the **rules that govern syntax and meaning** in the output language
- It needs a **scheme for mapping content** from the source language to the target language

- The compiler has a front end to deal with the source language
- It has a back end to deal with the target language
- Connecting the front end and the back end, it has a formal structure for representing the program in an **intermediate form** whose meaning is largely independent of either language
- To improve the translation, a compiler often includes an **optimizer** that analyses and rewrites that intermediate form

# Why High-Level Programming Language?

- Large impact on how fast programs can be developed
- Compared to machine language, the notation used by programming languages is closer to the 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 programs written in machine language
- Same program can be compiled to many different machine languages
- Some time-critical programs are still written partly in machine language as programs written in machine language is faster

**A good compiler will, however, be able to get very close to the speed of hand-written machine code when translating well-structured programs**

# History of Compiler

- The "compiler" word was first used in the early 1950s by Grace Murray Hopper
- The first compiler was build by John Backum and his group between 1954 and 1957 at IBM
- COBOL was the first programming language which was compiled on multiple platforms in 1960
- The study of the scanning and parsing issues were pursued in the 1960s and 1970s to provide a complete solution

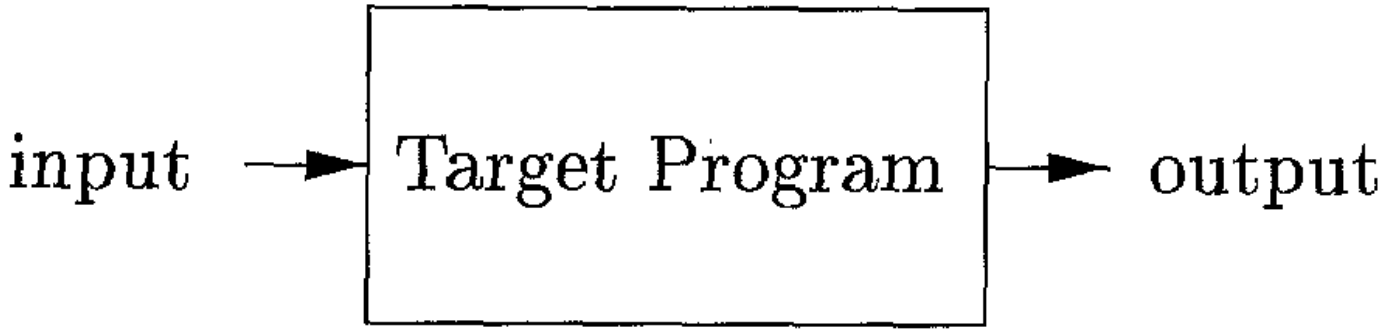
# Source-to-source Translation

- Some compilers produce a target program written in a human-oriented programming language rather than the assembly language of some computer
- The programs that these compilers produce require further translation before they can execute directly on a computer
- Example: Many research compilers
  - C programs produced as their output as compilers for C are available on most computers, this makes the target program executable on all those systems, at the cost of an extra compilation for the final target
- Compilers that produce target programming languages rather than the instruction set of a computer are often called ***source-to-source translators***.

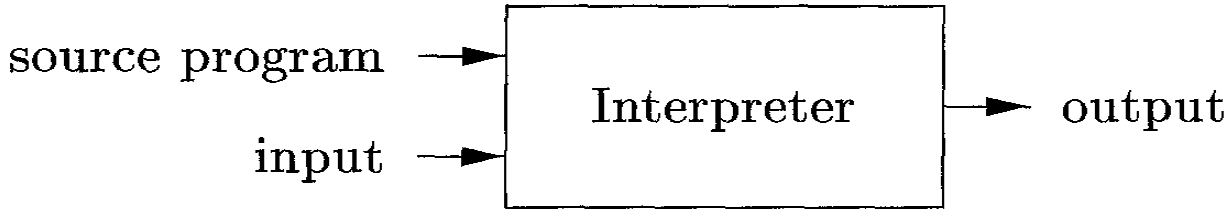
# Compiler for type setting programs

- A typesetting program that produces PostScript can be considered a compiler
- It takes as input a specification for how the document should look on the printed page and it produces as output a PostScript file
- PostScript is simply a language for describing images
- Because the typesetting program takes an executable specification and produces another executable specification, it is a compiler
- Example: LaTeX

# Running Target Program




# Interpreter

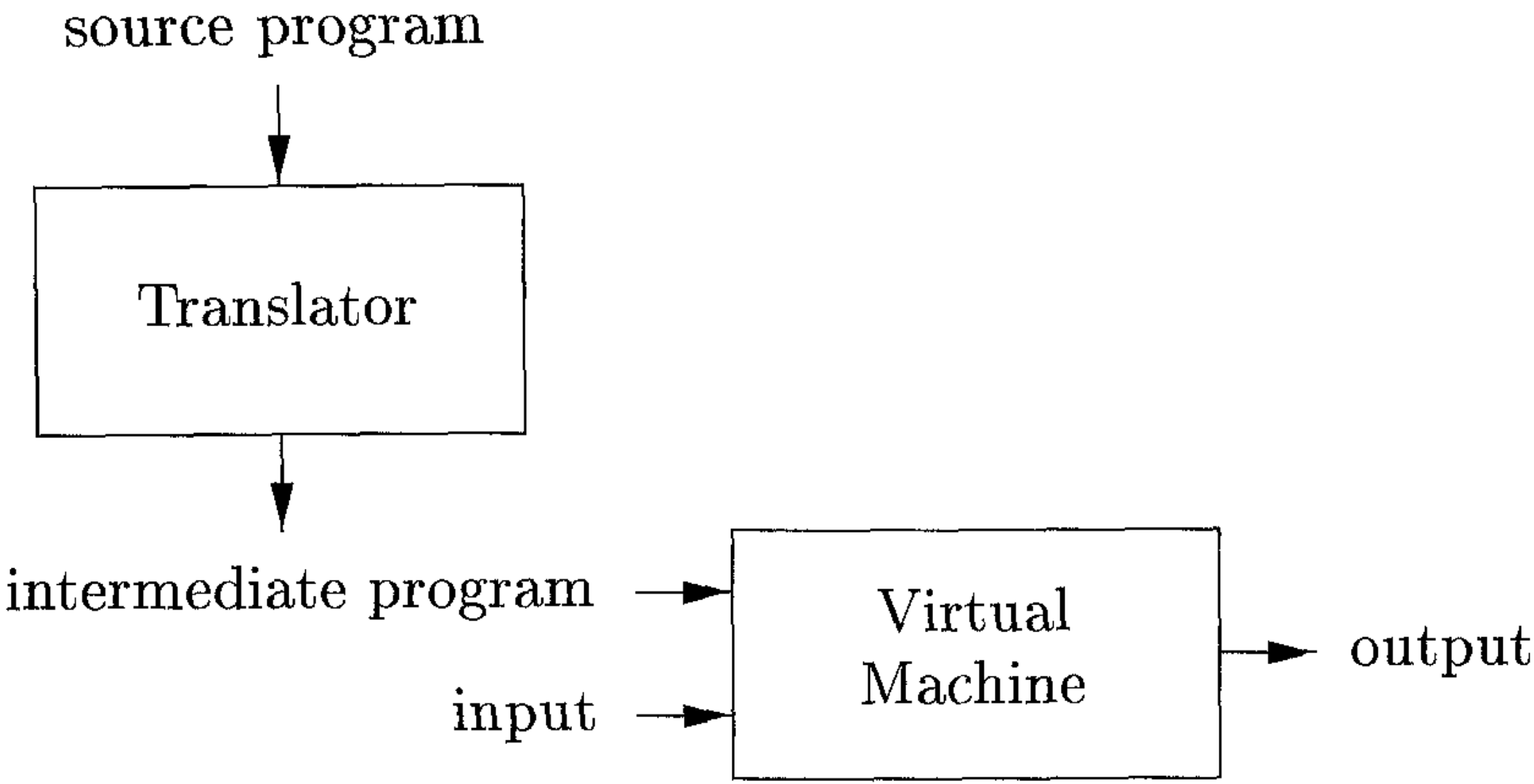



- **The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs**
- **An interpreter, however, can usually give better error diagnostics than a compiler, because it executes the source program statement by statement.**

# Compiler and Interpreter

- Both analyse the input program and determine whether or not it is a valid program
- Both build an internal model of the structure and meaning of the program
- Both determine where to store values during execution
- However, interpreting the code to produce a result is quite different from emitting a translated program that can be executed to produce the result

# Hybrid Compiler



# For JAVA

- Includes both compilation and interpretation
- Java is compiled from source code into a form called *bytecode*, a compact representation intended to decrease download times for Java applications
- **Virtual machine:** A virtual machine is a simulator for some processor. It is an *interpreter* for that machine's instruction set
- Java applications execute by running the bytecode on the corresponding Java Virtual Machine (jvm), an interpreter for bytecode
- Many implementations of the jvm include a compiler that executes at runtime, sometimes called a *just-in-time compiler*, or jit, that translates heavily used bytecode sequences into native code for the underlying computer

# Software Tools: Analysis before manipulating source program

Structure Editors:

- Input  a sequence of commands to build the source program
- Analyses the program text and putting it into hierarchical structure
- Performs additional tasks useful for preparation of the program
- Ex: Matching do for a while

Pretty Printers:

- Analyses a program and prints it in such a way that the structure of the program is clearly visible
- Ex: different font or color for comments, if-else

# Software Tools: Analysis before manipulating source program

Static Checkers:

- Reads a program and analyses it
- Attempts to discover potential bugs

Interpreters

# Conventional Compilers

Text Formatters:

- Input stream of characters
- Text to be typeset
- Commands to indicate paragraph, figures, etc.

Silicon Compilers:

- Source language similar to programming language as input
- Variables does not locate a position in memory rather signals 0 or 1 or group of signals
- Output is circuit design in an appropriate language

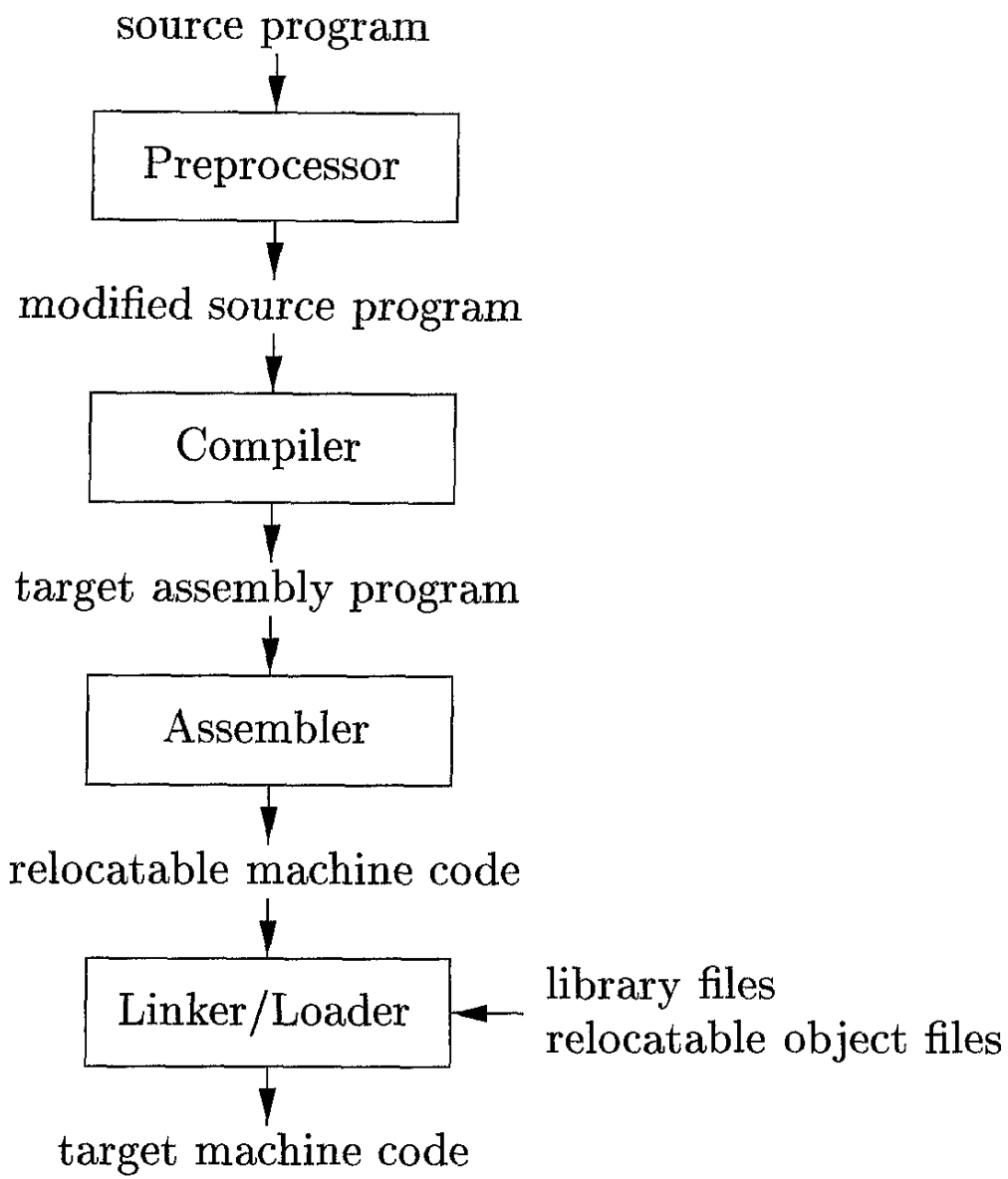
Query Interpreters:

- Translates a predicate containing relational and Boolean operator into commands to search a database for records satisfying that predicate

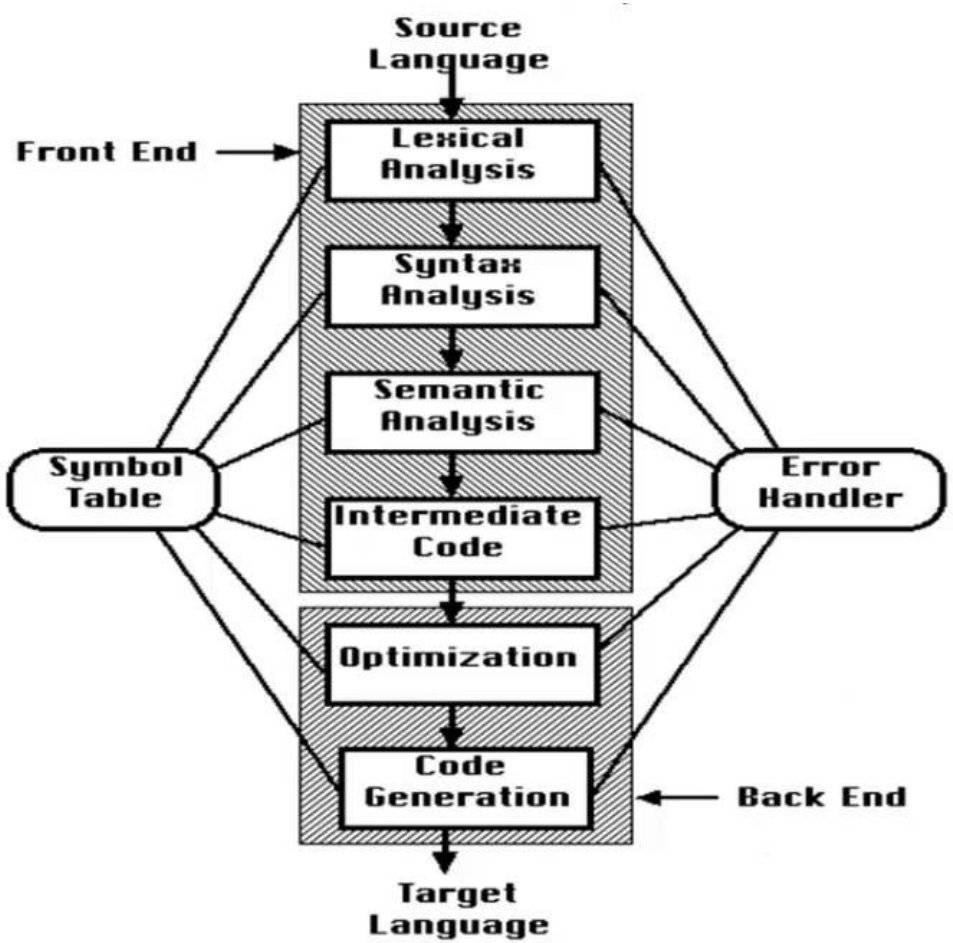
# A Language Processing System

- Preprocessor – expand shorthands
- Compiler -> assembly language program
- Easier to produce as output
- Easier to debug
- Assembly language is processed by assembler
- Relocatable machine code is produced
- Large programs are often compiled in pieces, so the relocatable machine code may have to be linked together with other relocatable object files and library files into the code that actually runs on the machine
- *linker* resolves external memory addresses, where the code in one file may refer to a location in another file
- The *loader* then puts together all of the executable object files into memory for execution

# A Language Processing System



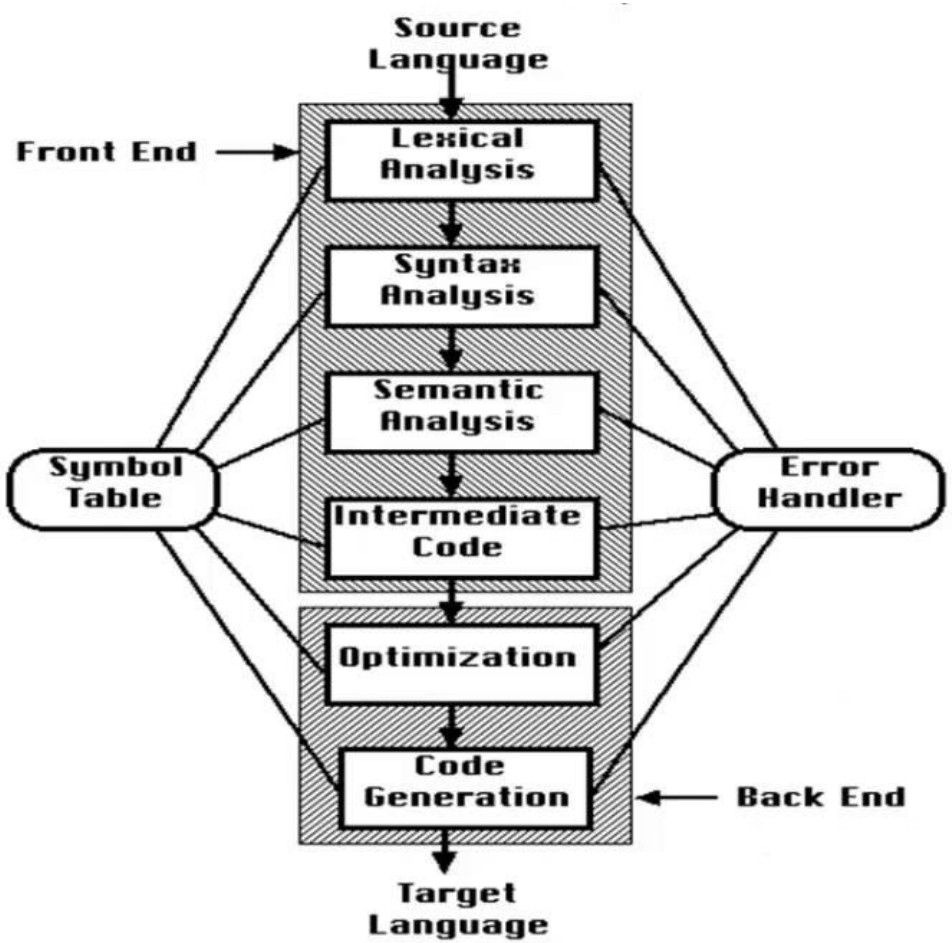
# Phases of Compiler




**ANALYSIS PART (front end)**

- The analysis part breaks up the source program into constituent pieces and imposes a grammatical structure on them
- Intermediate representation of the source program is created
- If syntactic or semantic mistakes, it must provide informative messages, so the user can take corrective action
- The analysis part also collects information about the source program and stores it in a data structure called a **symbol table**, which is passed along with the intermediate representation to the synthesis part

# Phases of Compiler

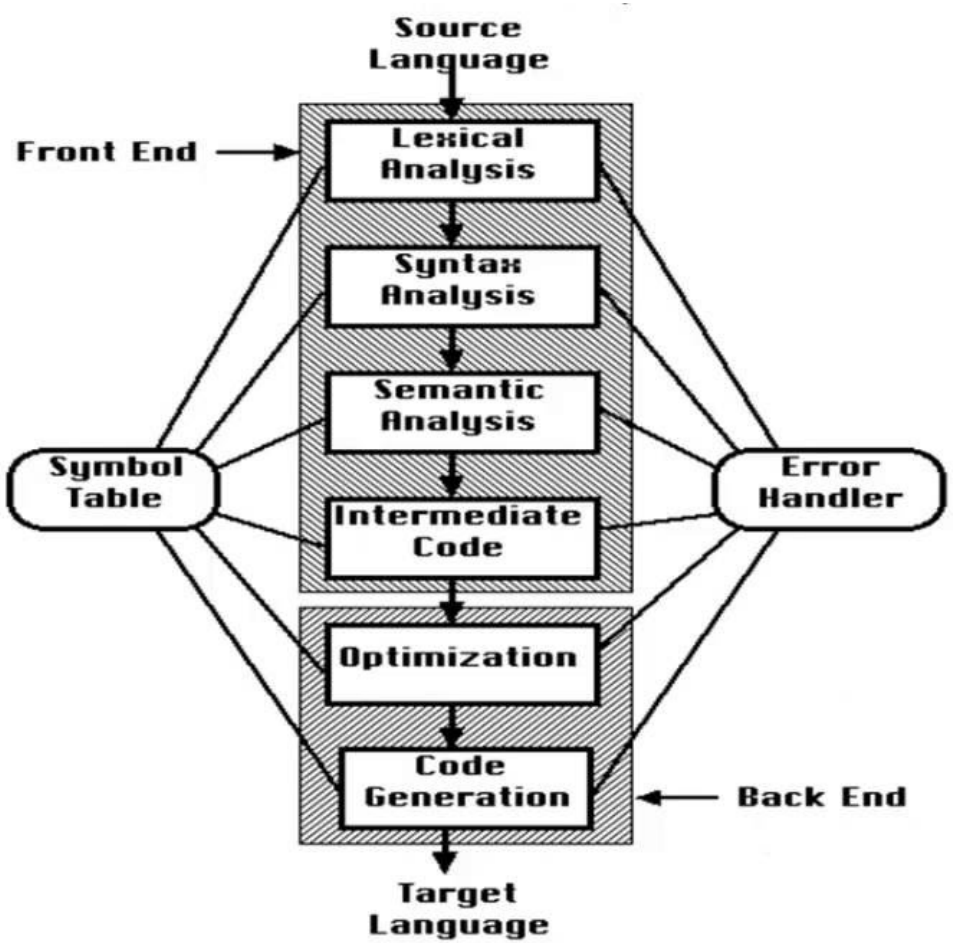



**SYNTHESIS PART (back end)**

Constructs the desired target program from the intermediate representation and the information in the symbol table

- Some compilers have a machine-independent optimization phase between the front end and the back end
- Optimization is optional

# Phases of Compiler

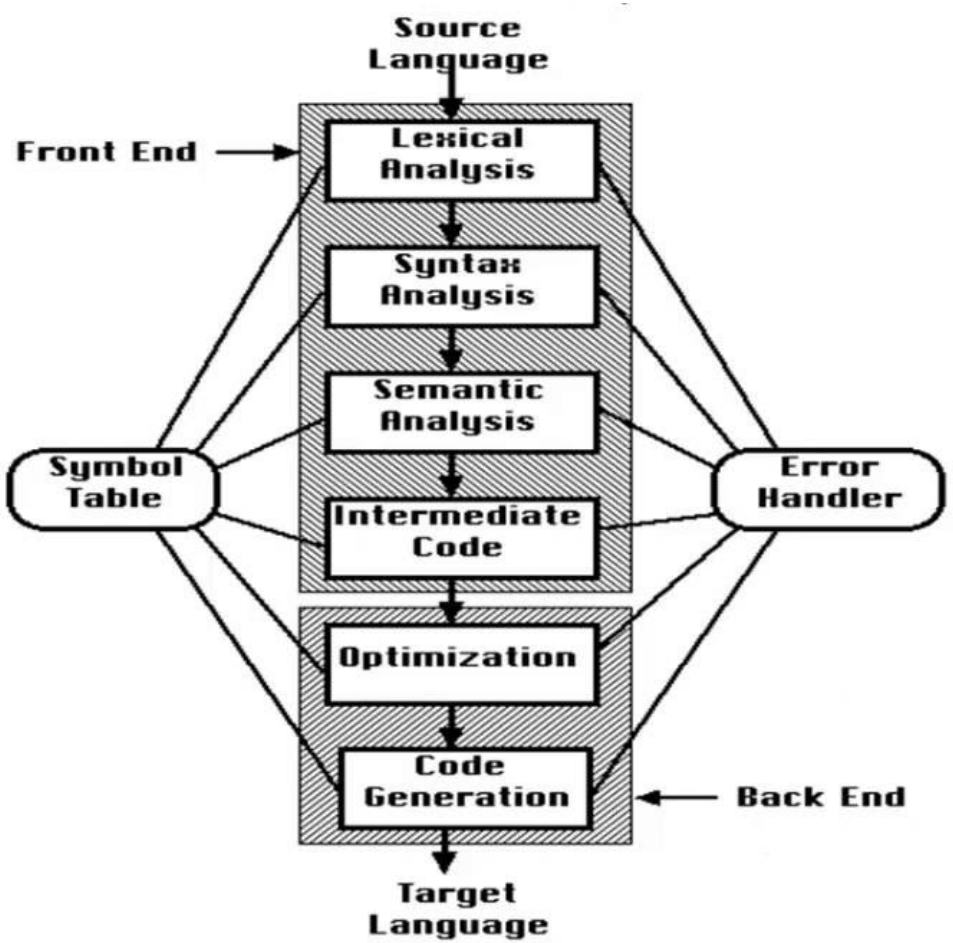



## Lexical Analysis

- reads the stream of characters making up the source program
- groups the characters into meaningful sequences called lexeme
- For each lexeme, the lexical analyser produces as output a token of the form

  <token-name, attribute-value>
- token-name is an abstract symbol that is used during syntax analysis
- 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 semantic analysis and code generation

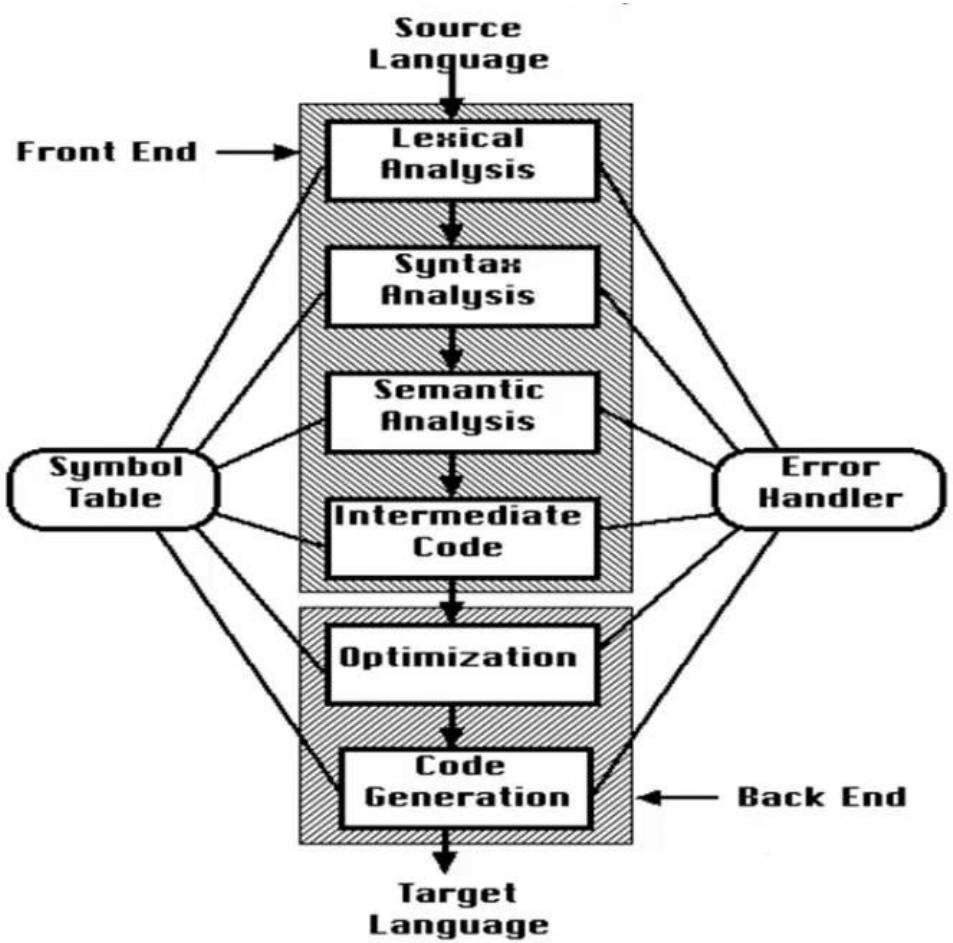
# Phases of Compiler




## Syntax Analysis

- Syntax tree: create a tree-like intermediate representation that depicts the grammatical structure of the token stream
- Interior node represents an operation and the children of the node represent the arguments of the operation

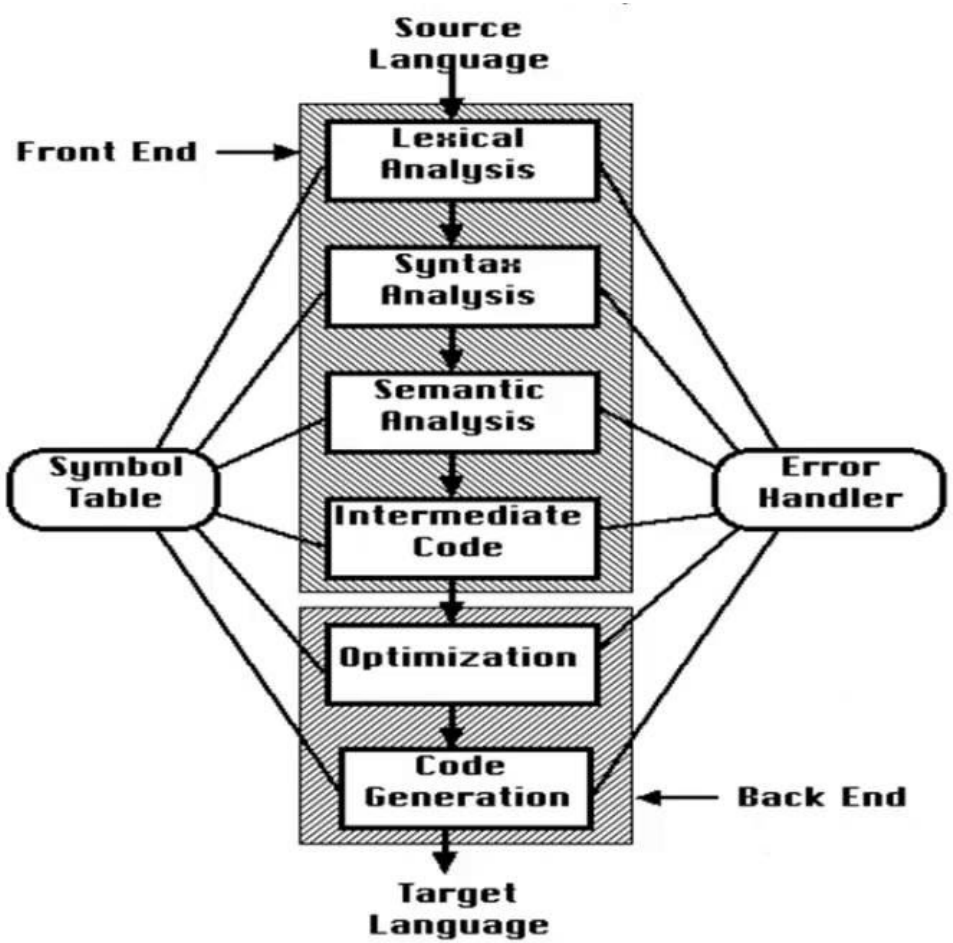
# Phases of Compiler




## Semantic Analysis

- The semantic analyser uses the syntax tree and the information in the symbol table to check the source program for **semantic consistency** with the language definition
- Type checking is an important part of semantic analysis
- The language specification may permit some type conversions called **coercions**

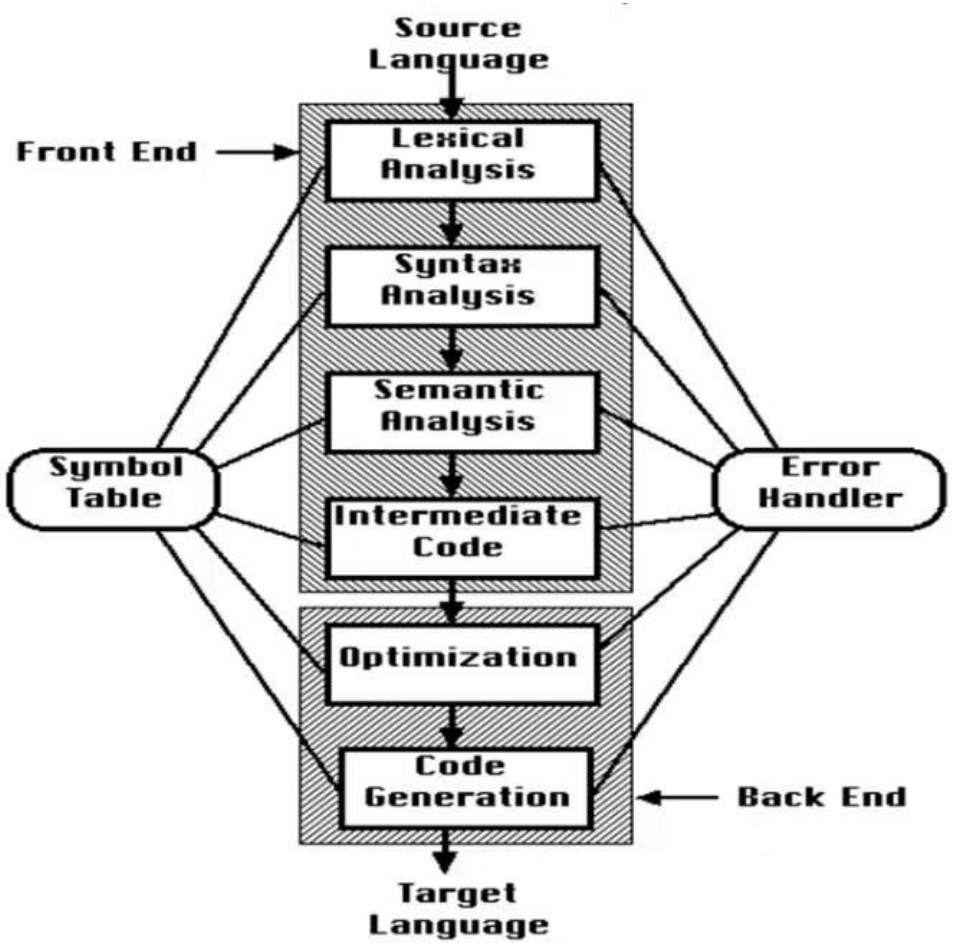
# Phases of Compiler




## Intermediate Code Generation

- A compiler may construct one or more intermediate representations, which can have a variety of forms
- After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation, which we can think of as a program for an **abstract machine**
- It should be easy to produce
- It should be easy to translate into the target machine

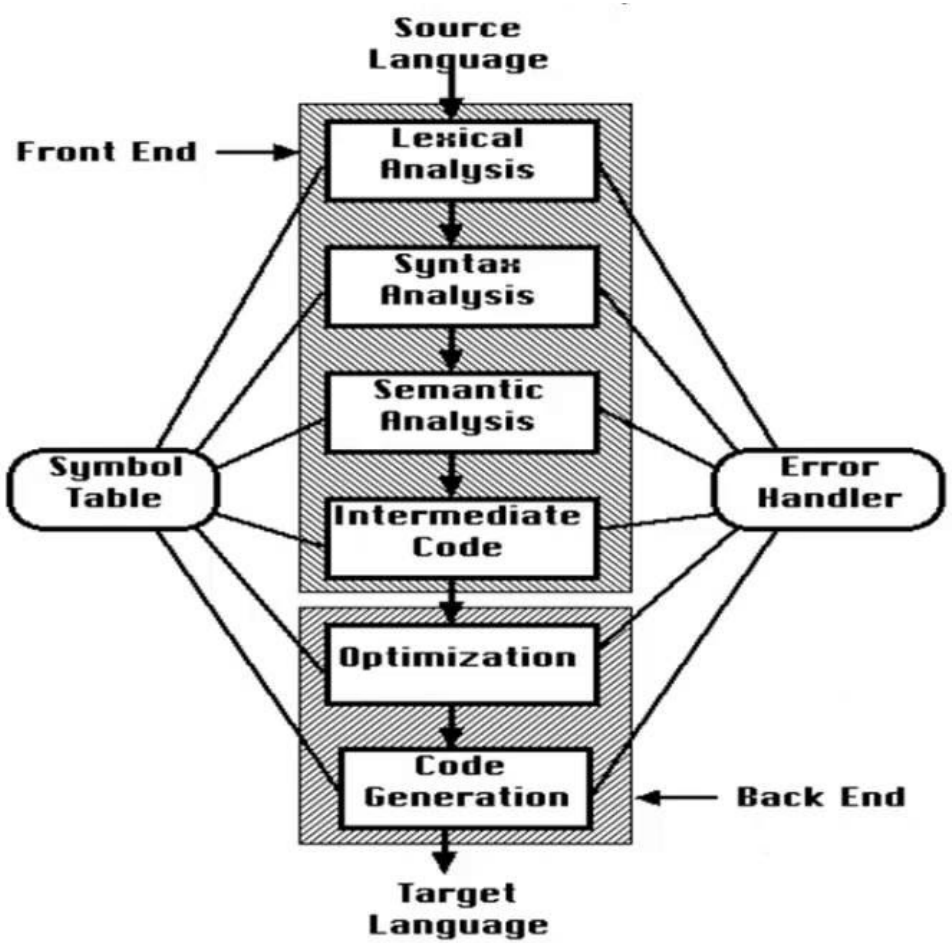
# Phases of Compiler




## Code Optimization

- To improve the intermediate code so that better target code will result
- Significant amount of time is spent on this phase
- There are simple optimizations that significantly improve the running time of the target program without slowing down compilation too much

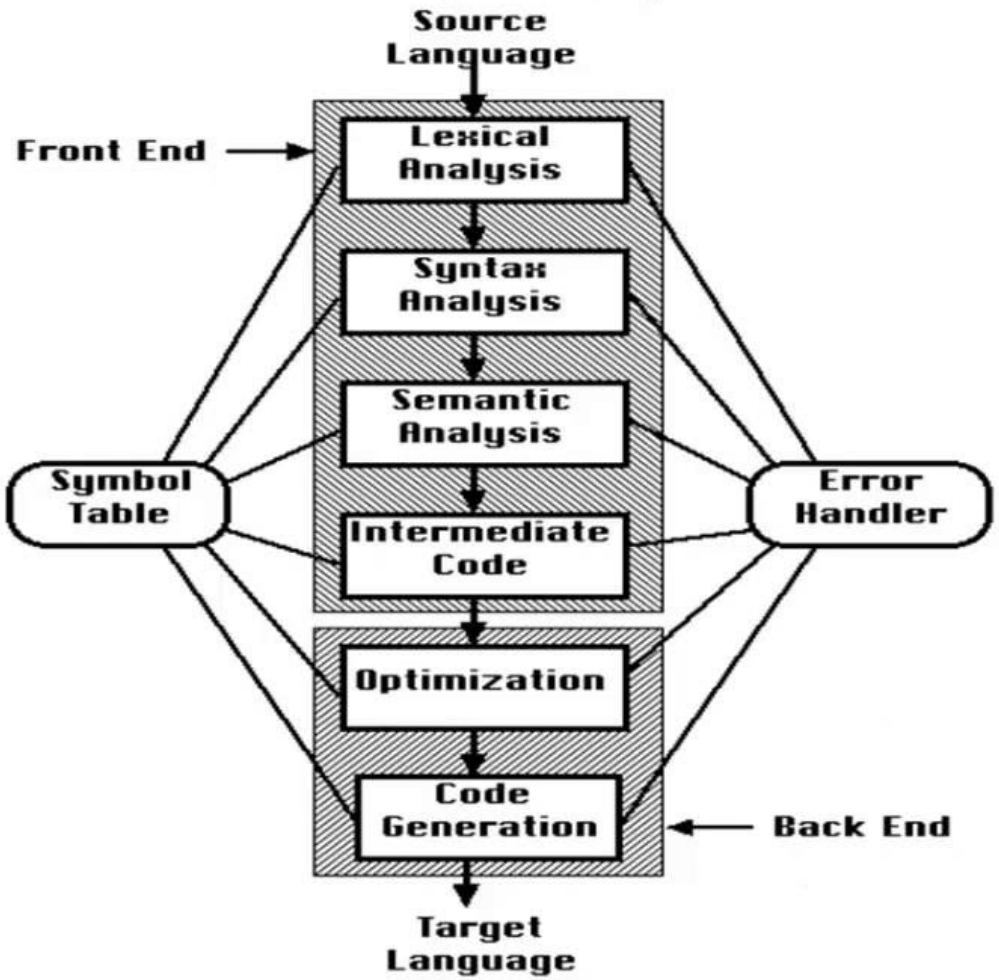
# Phases of Compiler




## Code Generation

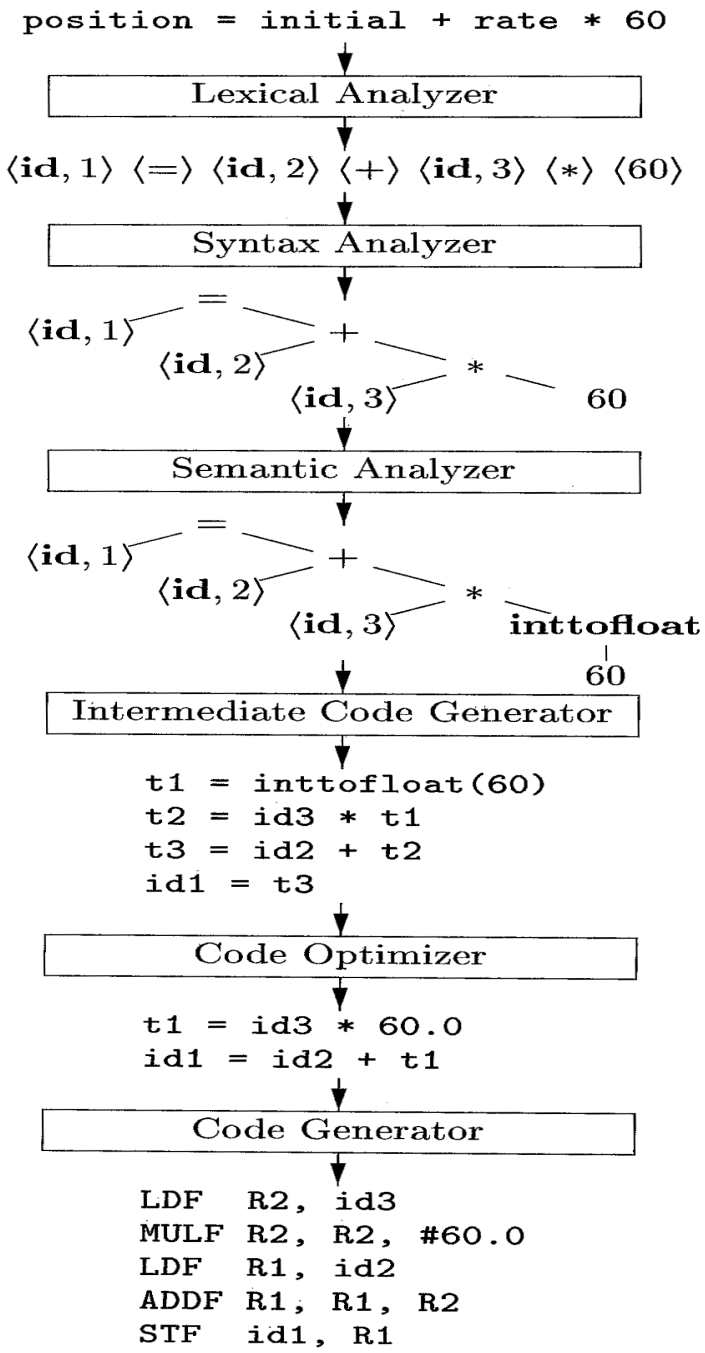
- If the target language is machine code, registers or memory locations are selected for each of the variables used by the program
- The intermediate instructions are translated into sequences of machine instructions that perform the same task
- Storage-allocation decisions are made either during intermediate code generation or during code generation

# Phases of Compiler

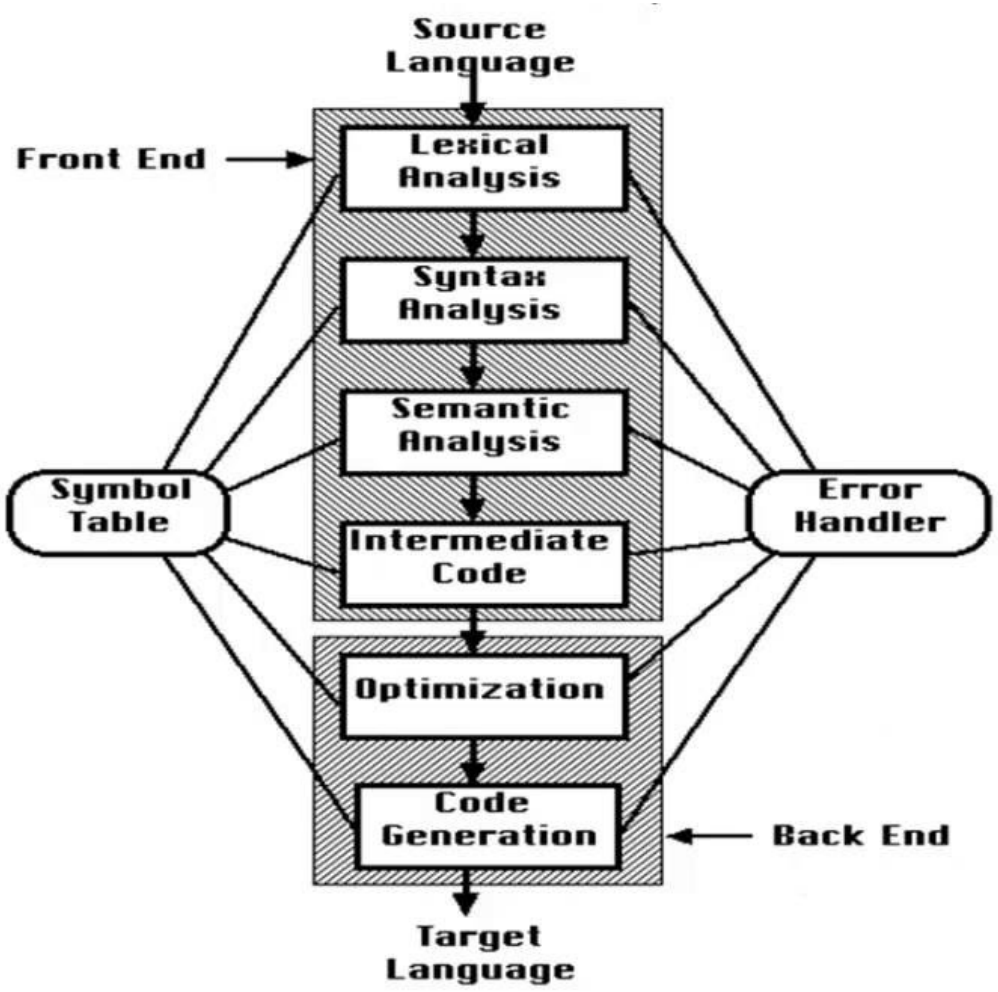



| | | |
|---|---|---|
| 1 | position | ··· |
| 2 | initial | ··· |
| 3 | rate | ··· |
| | | |

SYMBOL TABLE



# Phases of Compiler



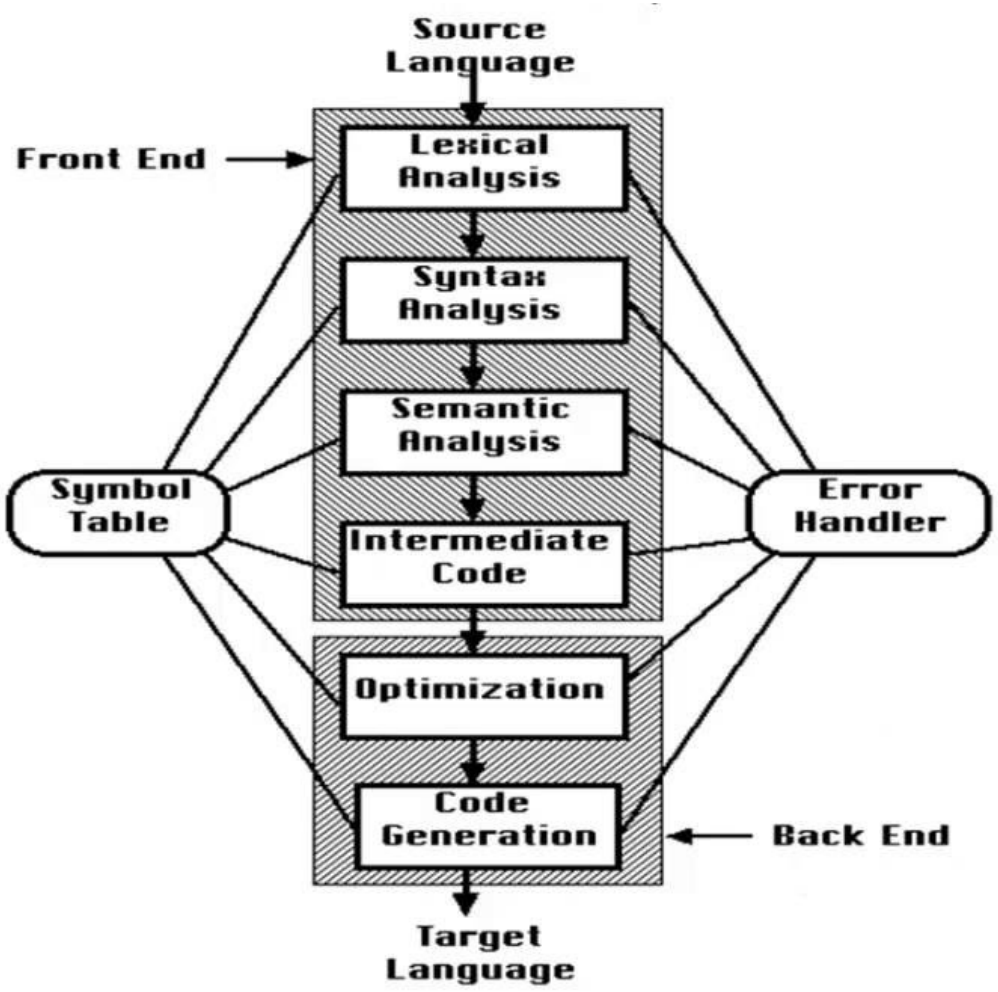

**Example**:

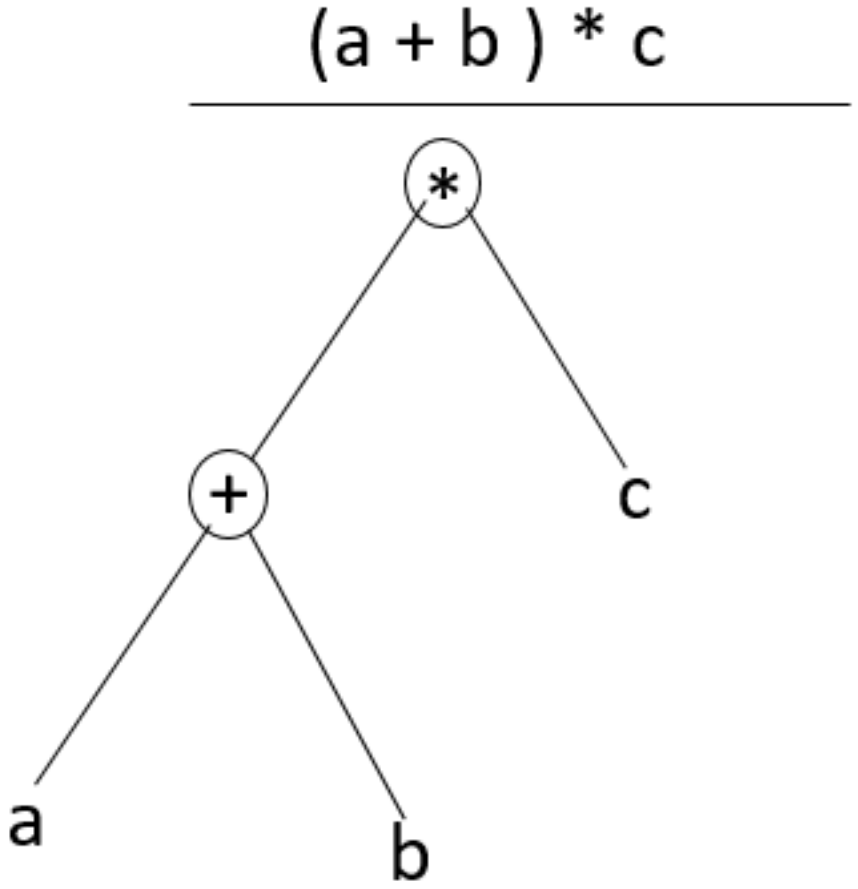
x = y + 10

Tokens

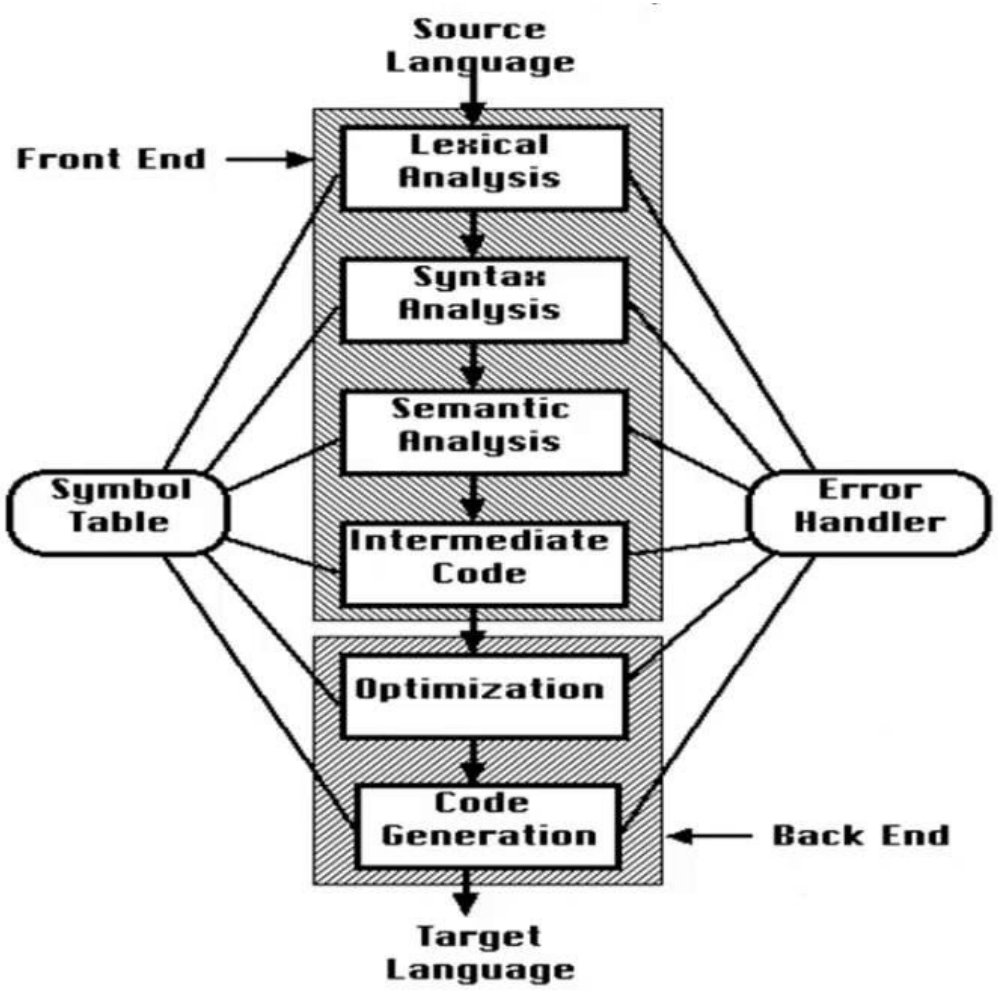
| | |
|---|---|
| X | identifier |
| = | Assignment operator |
| Y | identifier |
| + | Addition operator |
| 10 | Number |

# Phases of Compiler



# Phases of Compiler




$a = b + 60.0$

LDF R2, b

ADDF R1, R2, #60

STF a, R1

# Symbol Table

- A symbol table is a data structure containing a record for each identifier, with fields for the attributes of the identifier

- The data structure allows us to find the record for each identifier quickly and to store or retrieve data from that record quickly

- Symbol table is a Data Structure in a Compiler used for managing information about variables & their attributes

# Error Recovery

- Each phase can encounter errors
- After detecting an error, a phase must somehow deal with that error, so that compilation can proceed, allowing further errors in the source program to be detected
- A compiler that stops when it finds the first error is not as helpful
- The syntax and semantic analysis phases usually handle a large fraction of the errors detectable by the compiler
- The lexical phase can detect errors where the characters remaining in the input do not form any token of the language
- Errors where the token stream violates the structure rules (syntax) of the language are determined by the syntax analysis phase

# Cross Compiler

- The computer the compiler runs on is called the **host**, and the computer the new programs run on is called the **target**. When the host and target are the same type of machine, the compiler is a **native compiler**. When the host and target are different, the compiler is a **cross compiler**

# Why Cross Compile?

- In theory, a PC user who wanted to build programs for some device could get the appropriate target hardware (or emulator), boot a Linux distro on that, and compile natively within that environment. While this is a valid approach (and possibly even a good idea when dealing with something like a Mac Mini), it has a few prominent downsides for things like a linksys router or iPod:

- **Speed** - Target platforms are usually much slower than hosts, by an order of magnitude or more. Most special-purpose embedded hardware is designed for low cost and low power consumption, not high performance. Modern emulators (like qemu) are actually faster than a lot of the real world hardware they emulate, by virtue of running on high-powered desktop hardware

# Why Cross Compile?

- **Capability** - Compiling is very resource-intensive. The target platform usually doesn't have gigabytes of memory and hundreds of gigabytes of disk space the way a desktop does; it may not even have the resources to build "hello world", let alone large and complicated packages.
- **Availability** - Bringing Linux up on a hardware platform it's never run on before requires a cross-compiler. Even on long-established platforms like Arm or Mips, finding an up-to-date full-featured prebuilt native environment for a given target can be hard. If the platform in question isn't normally used as a development workstation, there may not be a recent prebuilt distro readily available for it, and if there is it's probably out of date. If you have to build your own distro for the target before you can build on the target, you're back to cross-compiling anyway.

# Why Cross Compile?

**Flexibility** - A fully capable Linux distribution consists of hundreds of packages, but a cross-compile environment can depend on the host's existing distro from most things. Cross compiling focuses on building the target packages to be deployed, not spending time getting build-only prerequisites working on the target system.

**Convenience** - The user interface of headless boxes tends to be a bit crampled. Diagnosing build breaks is frustrating enough as it is. Installing from CD onto a machine that hasn't got a CD-ROM drive is a pain. Rebooting back and forth between your test environment and your development environment gets old fast, and it's nice to be able to recover from accidentally lobotomizing your test system.

# Cousins of Compilers

Preprocessor

A preprocessor is a program that processes its input data to produce output that is used as input to another program. The output is said to be a preprocessed form of the input data, which is often used by some subsequent programs like compilers.

They may perform the following function

1. Macro Processing
2. File Inclusion
3. Rational Processing
4. Language Extension

# Cousins of Compilers

**Macro processing:**

A macro is a rule or pattern that specifies how a certain input sequence should be mapped to an output sequence according to a defined procedure. The mapping procedure that instantiates a macro into a specific output sequence is known as macro expansion.

**File Inclusion:**

Preprocessor includes header files into the program text. When the preprocessor finds an #include directive it replaces it by the entire content of the specified file.

# Cousins of Compilers

**Rational Processing:**

These processors change older languages with more modern flow-of-control and data-structuring facilities.

**Language extension:**

These processors attempt to add capabilities to the language by what amounts to built-inmacros. For example, the language Equel is a database query language embedded in C

# Cousins of Compilers

**Assembler**

Assembler creates object code by translating assembly instruction mnemonics into machine code. There are two types of assemblers:

· One-pass assemblers go through the source code once and assume that all symbols will be defined before any instruction that references them.

· Two-pass assemblers create a table with all symbols and their values in the first pass, and then use the table in a second pass to generate code

# Cousins of Compilers

**Linker and Loader**

A **linker** or **link editor** is a program that takes one or more objects generated by a compiler and combines them into a single executable program. Three tasks of the linker are

1.Searches the program to find library routines used by program, e.g. printf(), math routines.

2.Determines the memory locations that code from each module will occupy and relocates its instructions by adjusting absolute references 3. Resolves references among files.

A **loader** is the part of an operating system that is responsible for loading programs in memory, one of the essential stages in the process of starting a program.

# Compiler Passes

A collection of phases is done only once (single pass) or multiple times (multi pass)

· Single pass: usually requires everything to be defined before being used in source program.

· Multi pass: compiler may have to keep entire program representation in memory.

Several phases can be grouped into one single pass and the activities of these phases are interleaved during the pass. For example, lexical analysis, syntax analysis, semantic analysis and intermediate code generation might be grouped into one pass.

# More on next class