**Module-7**

**System Programming**

**Concept of system programming-**

### What is a System?

Before we can understand what Systems Programming entails, we first need to understand what a System is. Software tends to fall into one of two camps, system software and application software.

**System software** is computer software designed to provide a platform to other software. Examples of system software include operating systems, computational science software, game engines, industrial automation, and software as service applications.Such software is not considered system software when it can be uninstalled usually without affecting the functioning of other software.
**—**[**wikipedia.org**](https://en.wikipedia.org/wiki/System_software?ref=hackernoon.com)

**System software** is a platform comprised of Operating System (OS) programs and services, including settings and preferences, file libraries and functions used for system applications. System software also includes device drivers that run basic computer hardware and peripherals.
**—**[**techopedia.com**](https://www.techopedia.com/definition/5476/system-software?ref=hackernoon.com)

**System software** refers to the files and programs that make up your computer’s [operating system](https://techterms.com/definition/operating_system?ref=hackernoon.com). System files include libraries of functions, system services, drivers for printers and other hardware, system preferences, and other configuration files. The programs that are part of the system software include assemblers, compilers, file management tools, system utilities, and [debuggers](https://techterms.com/definition/debugger?ref=hackernoon.com).
**—**[**techterms.com**](https://techterms.com/definition/systemsoftware?ref=hackernoon.com)

The Wikipedia definition is very vague on what is considered system software as long as it is providing services to other applications. However the other two definitions focus purely on the operating system — drivers, kernels, libraries and functions (think kernel/libc header files and shared objects). This implies a close relationship to hardware. If we look at another Wikipedia article on [Systems Programming](https://en.wikipedia.org/wiki/System_programming?ref=hackernoon.com) we see:

System programming requires a great degree of hardware awareness.

The article goes on to imply that a core part of system programming is the need for things to be very fast. This makes sense why we would need to know a lot about the hardware. It also makes sense that speed (performance) would be a core part of systems programming if it is a platform to other software.

If the most central part of your application (the system software “platform”) is slow, then the whole application is slow. For many applications, especially at scale, this would be a deal-breaker.

#### System Software in a Nutshell

The quotes above and other resources[1][2] has lead me to the following criteria to define system software:

* Provides a platform for other software to be built upon.
* Directly or closely interfaces with computer hardware in order to gain necessary performance and expose abstractions (as part of the platform).

#### What is and What isn’t System Software

Examples of what is system software:

* OS Kernels
* Drivers
* Bare Metal Hypervisors (e.g. [Hyper-V](https://en.wikipedia.org/wiki/Hyper-V?ref=hackernoon.com) and [VM Ware ESXi](https://en.wikipedia.org/wiki/VMware_ESXi?ref=hackernoon.com))
* Compilers (that produce native binaries) and Debuggers

Examples of what isn’t system software:

* GUI Chat application (Slack, Discord, etc)
* Web-based JavaScript Application
* Web Service API

You’ll notice that while Web Service API’s provide a service to other software, they don’t (typically) interact with hardware in order to expose abstractions over it. However there are applications that fall within a middle grey area. The ones that come to mind are high performance computing applications and embedded software.

High performance computing (HPC) applications, such as real-time trading on stock exchanges, don’t typically expose a platform, but it is common for them to write code that interfaces directly with hardware. An example would be bypassing the networking stack offered by the kernel and implementing their own networking stack talking directly to the NIC(s). In this way we can see how HPC software shares many similarities with systems software, by interacting directly with hardware in order to provide the needed performance gains.

Embedded software development also shares many similarities with systems software in that code is written to directly interface with hardware. However, any abstractions provided are typically consumed by the same software and could not be considered a platform.

It’s important to note applications that share similarities with our definition of system software since you’ll likely see those applications/jobs described in these terms (systems software, systems engineers, etc.)

### Systems Programming (+ Languages)

Having defined **Systems**, we can now define **Systems Programming** as the act of building **Systems Software**using **System Programming Languages**. Simple enough, right?

Well there one thing we skipped over, languages. People often talk about Systems Programming Languages in ways such as “X is great, it’s fast, compiled, and a systems programming language.” But is everyone on the same page as to what a systems programming language is?

Given our definitions of **Systems** I would define the criteria for a **Systems Programming Language**to be:

* Compiled to native binary
* Can be built without dependencies on other software (including a kernel)
* Performance characteristics similar to other systems programming languages

#### Compile to Native Binary

If a language cannot compile to an executable that is directly interpretable by the CPU then it, by definition, is running on a platform (e.g. JVM, Ruby VM, Python VM, etc). There may be some arguments to be made here, but for simplicity I think this is a suitable criteria.

#### No Dependencies

The argument is similar to compiling to a native binary. If the language always requires some other software be present to execute, then it is running on a platform. An example of this is Go and it’s included standard library. It requires support from the OS to perform basic actions such as allocating memory, spawning threads (for goroutines to run on), for its built-in network poller, and other actions. While [it is possible to re-implement these core functions](https://github.com/achilleasa/gopher-os?ref=hackernoon.com), it does create a barrier to use in this context and it is easy to imagine why not all languages, even those that compile to static binaries, are intended as system programming languages.

#### Similar Performance Characteristics

This one is a bit of a cop-out. However, it is to say that within the system of languages typically classified as systems programming languages, there should not be large (order of magnitudes) differences in performance characteristics. By characteristics I am explicitly referring to execution speed and memory efficiency.

The golden standard for comparison is C and/or C++ as is often represented in comparative benchmarks, which measure execution speed in how many orders of magnitudes slower languages are than C/C++.

#### Naming a Few

The languages that come to mind immediately, given the above definition are C and C++. But there are also newer languages such as [Rust](https://github.com/achilleasa/gopher-os?ref=hackernoon.com) and [Nim](https://nim-lang.org/?ref=hackernoon.com) which also fill this niche. In fact, there is already an OS written entirely in Rust [(RedoxOS](https://www.redox-os.org/?ref=hackernoon.com)) and a kernel in Nim ([nimkernel](https://github.com/dom96/nimkernel?ref=hackernoon.com)).

**What does System Programming mean?**

* System programming involves designing and writing computer programs that allow the computer hardware to interface with the programmer and the user, leading to the effective execution of application software on the computer system.
* The system programs are used to program the operating system software. While application programs provide software that is used directly by the user, system programs provide software that are used by other systems such as SaaS applications, computational science applications etc.
* System programming is an essential and important foundation in any computer’s application development, and always evolving to accommodate changes in the computer hardware. This kind of programming requires some level of hardware knowledge and is machine dependent; the system programmer must therefore know the intended hardware on which the software is required to operate.
* Additionally, the programmer may make some assumptions on the hardware and other system components. The software is usually written in a low-level programming language which is able to operate efficiently in a resource-constrained environment, and with little runtime overhead using a small library, or none at all. The low-level language enables direct control over memory access and allows the program to be written directly in an assembly language. The majority of programs are written using assembly languages such as C, C++ and C#.
* System programming leads to the development of computer system software that manages and controls the computer operations. The low-level codes are very close to the hardware level and deal with things such as registers and memory allocations. The system programs or system software coordinates data transfer across the various components and deals with the compiling, linking, starting and stopping of programs, reading from files as well as writing to files.
* The system programming enhances or extends the functions of an operating system and may comprise components such as drivers, utilities and updates. They enable efficient management of hardware resources such as memory, file access, I/O operations, device management and process management such as process administration and multi-tasking. An example is an operating system, which usually acts as the interface between the user, the application software, and computer hardware. The OS provides an environment that enables users to execute other programs efficiently. Comprising of a set of system programs, the operating system functions include storage management, file handling, memory management, CPU and device scheduling and management, error handling, process control and more.

**What is the purpose of System Programs?**

* System programs provide an environment where programs can be developed and executed. In the simplest sense, system programs also provide a bridge between the user interface and system calls. In reality, they are much more complex. For example, a compiler is a complex system program.
* **System Programs Purpose**
* The system program serves as a part of the operating system. It traditionally lies between the user interface and the system calls. The user view of the system is actually defined by system programs and not system calls because that is what they interact with and system programs are closer to the user interface.
* An image that describes system programs in the operating system hierarchy is as follows:



* In the above image, system programs as well as application programs form a bridge between the user interface and the system calls. So, from the user view the operating system observed is actually the system programs and not the system calls.
* **Types of System Programs**
* System programs can be divided into seven parts. These are given as follows:
* **Status Information**
* The status information system programs provide required data on the current or past status of the system. This may include the system date, system time, available memory in system, disk space, logged in users etc.
* **Communications**
* These system programs are needed for system communications such as web browsers. Web browsers allow systems to communicate and access information from the network as required.
* **File Manipulation**
* These system programs are used to manipulate system files. This can be done using various commands like create, delete, copy, rename, print etc. These commands can create files, delete files, copy the contents of one file into another, rename files, print them etc.
* **Program Loading and Execution**
* The system programs that deal with program loading and execution make sure that programs can be loaded into memory and executed correctly. Loaders and Linkers are a prime example of this type of system programs.
* **File Modification**
* System programs that are used for file modification basically change the data in the file or modify it in some other way. Text editors are a big example of file modification system programs.
* **Application Programs**
* Application programs can perform a wide range of services as per the needs of the users. These include programs for database systems, word processors, plotting tools, spreadsheets, games, scientific applications etc.
* **Programming Language Support**
* These system programs provide additional support features for different programming languages. Some examples of these are compilers, debuggers etc. These compile a program and make sure it is error free respectively.

**\*I/O System Management –**
The module that keeps track of the status of devices is called the I/O traffic controller. Each I/O device has a device handler that resides in a separate process associated with that device.
The I/O subsystem consists of

* A memory Management component that includes buffering caching and spooling.
* A general device driver interface.

Drivers for specific hardware devices.

**Assembler**
The input to an assembler is an assembly language program. The output is an object program plus information that enables the loader to prepare the object program for execution. At one time, the computer programmer had at his disposal a basic machine that interpreted, through hardware, certain fundamental instructions. He would program this computer by writing a series of ones and Zeros (Machine language), place them into the memory of the machine.

**Compiler**
The High-level languages- examples are FORTRAN, COBOL, ALGOL and PL/I are processed by compilers and interpreters. A compiler is a program that accepts a source program in a “high-level language “and produces a corresponding object program. An interpreter is a program that appears to execute a source program as if it was machine language. The same name (FORTRAN, COBOL, etc.) is often used to designate both a compiler and its associated language.

**Loader**
A Loader is a routine that loads an object program and prepares it for execution. There are various loading schemes: absolute, relocating and direct-linking. In general, the loader must load, relocate and link the object program. The loader is a program that places programs into memory and prepares them for execution. In a simple loading scheme, the assembler outputs the machine language translation of a program on a secondary device and a loader places it in the core. The loader places into memory the machine language version of the user’s program and transfers control to it. Since the loader program is much smaller than the assembler, those make more core available to the user’s program.

**Differentiate between application program and system program.**

**System Program**

* It is the collection of components and art or designing of a given program.
* System program aims to produce software which provides services to the computer hardware.
* System software that executes the application software. It helps in executing the application software.
* These programs allow the application programmer to focus on application to be develop without concerning about the internal details of the System.
* **Examples:** Assembler Microprocessor, Loader, Linker, Compiler

**Application Program:**

* These are the set of programs that view computer as a tool for solving a particular problem.
* Application software is a software that are been used by the end user.
* Application programming is used to build application software which includes software like notepad, Word Pad, calculator, web browser, Microsoft excel and many more.
* It interacts with system software which in turn interacts and makes physical hardware functional.
* **Examples:** Word Processing, Hotel Management System, Accounting Package etc.

| **System Program** | **Application Program** |
| --- | --- |
| 1) It is the collection of components and art or designing of a given program. | 1) It is the set of programs that view computer as a tool for solving a particular problem. |
| 2) Programming is done using assembly language which interacts with hardware. | 2) Application programming is used to build application software which includes software. |
| 3) System software that executes the application software. | 3) Application software is a software that are been used by the end user. |
| 4) System programming is used to write low level instructions | 4)Application Programming is used to write High level instructions |
|  |  |
| 5)Examples:- Loader, Linker, Compiler | 5) Examples:- Library management System, calculator, web browser |

**Compiler**

A compiler is a special program that processes statements written in a particular programming language and turns them into machine language or "code" that a computer's [processor](https://whatis.techtarget.com/definition/processor) uses. Typically, a programmer writes language statements in a language such as [Pascal](https://whatis.techtarget.com/definition/Pascal) or [C](https://searchwindowsserver.techtarget.com/definition/C) one line at a time using an editor. The file that is created contains what are called the source statements. The programmer then runs the appropriate language compiler, specifying the name of the file that contains the source statements.

In general a compiler is a computer program which helps you transform source code written in a high-level language into low-level machine language. It translates the code written in one programming language to some other language without changing the meaning of the code. The compiler also makes the end code efficient which is optimized for execution time and memory space.

The compiling process includes basic translation mechanisms and error detection. Compiler process goes through lexical, syntax, and semantic analysis at the front end, and code generation and optimization at a back-end.



When executing (running), the compiler first parses (or analyzes) all of the language statements syntactically one after the other and then, in one or more successive stages or "passes", builds the output code, making sure that statements that refer to other statements are referred to correctly in the final code. Traditionally, the output of the compilation has been called object code or sometimes an object module . (Note that the term "object" here is not related to [object-oriented programming](https://searchapparchitecture.techtarget.com/definition/object-oriented-programming-OOP).) The object code is machine code that the processor can execute one instruction at a time.

The [Java](https://www.theserverside.com/definition/Java) programming language, a language used in [object-oriented programming](https://searchapparchitecture.techtarget.com/definition/object-oriented-programming-OOP), introduced the possibility of compiling output (called [bytecode](https://whatis.techtarget.com/definition/bytecode) ) that can run on any computer system platform for which a Java [virtual machine](https://searchservervirtualization.techtarget.com/definition/virtual-machine) or bytecode interpreter is provided to convert the bytecode into instructions that can be executed by the actual hardware processor. Using this virtual machine, the bytecode can optionally be recompiled at the execution platform by a [just-in-time compiler](https://www.theserverside.com/definition/just-in-time-compiler-JIT). (See also: [Java compiler](https://www.theserverside.com/definition/Java-compiler))

Traditionally in some operating systems, an additional step was required after compilation - that of resolving the relative location of instructions and data when more than one object module was to be run at the same time and they cross-referred to each other's instruction sequences or data. This process was sometimes called linkage editing and the output known as a load module.

A compiler works with what are sometimes called [3GL](https://whatis.techtarget.com/definition/programming-language-generations) and higher-level languages. An [assembler](https://searchdatacenter.techtarget.com/definition/assembler) works on programs written using a processor's assembler language.

**Types of Compiler**

* Single Pass Compilers
* Two Pass Compilers
* Multipass Compilers

**Single Pass Compiler**



In single pass Compiler source code directly transforms into machine code. For example, Pascal language.

**Two Pass Compiler**



Two pass Compiler is divided into two sections, viz.

1. **Front end:**It maps legal code into Intermediate Representation (IR).
2. **Back end:**It maps IR onto the target machine

The Two pass compiler method also simplifies the retargeting process. It also allows multiple front ends.

**Multipass Compilers**



The multipass compiler processes the source code or syntax tree of a program several times. It divided a large program into multiple small programs and process them. It develops multiple intermediate codes. All of these multipass take the output of the previous phase as an input. So it requires less memory. It is also known as 'Wide Compiler'.

**Steps for Language processing systems**

Before knowing about the concept of compilers, you first need to understand a few other tools which work with compilers.



* **Preprocessor**: The preprocessor is considered as a part of the Compiler. It is a tool which produces input for Compiler. It deals with macro processing, augmentation, language extension, etc.
* **Interpreter**: An interpreter is like Compiler which translates high-level language into low-level machine language. The main difference between both is that interpreter reads and transforms code line by line. Compiler reads the entire code at once and creates the machine code.
* **Assembler**: It translates assembly language code into machine understandable language. The output result of assembler is known as an object file which is a combination of machine instruction as well as the data required to store these instructions in memory.
* **Linker**: The linker helps you to link and merge various object files to create an executable file. All these files might have been compiled with separate assemblers. The main task of a linker is to search for called modules in a program and to find out the memory location where all modules are stored.
* **Loader**: The loader is a part of the OS, which performs the tasks of loading executable files into memory and run them. It also calculates the size of a program which creates additional memory space.
* **Cross-compiler**: A cross compiler is a platform which helps you to generate executable code.
* **Source-to-source Compiler**: Source to source compiler is a term used when the source code of one programming language is translated into the source of another language.

**Application of Compilers**

* Compiler design helps full implementation Of High-Level Programming Languages
* Support optimization for Computer Architecture Parallelism
* Design of New Memory Hierarchies of Machines
* Widely used for Translating Programs
* Used with other Software Productivity Tools

**Compare compiler and interpreter**

**Role of Compiler**

* Compliers reads the source code, outputs executable code
* Translates software written in a higher-level language into instructions that computer can understand. It converts the text that a programmer writes into a format the CPU can understand.
* The process of compilation is relatively complicated. It spends a lot of time analyzing and processing the program.
* The executable result is some form of machine-specific binary code.

**Role of Interpreter**

* The interpreter converts the source code line-by-line during RUN Time.
* Interpret completely translates a program written in a high-level language into machine level language.
* Interpreter allows evaluation and modification of the program while it is executing.
* Relatively less time spent for analyzing and processing the program
* Program execution is relatively slow compared to compiler



**Similarities between compiled and interpreted languages are given below:**



| **BASIS FOR COMPARISON** | **COMPILER** | **INTERPRETER** |
| --- | --- | --- |
| Input | It takes an entire program at a time. | It takes a single line of code or instruction at a time. |
| Output | It generates intermediate object code. | It does not produce any intermediate object code. |
| Working mechanism | The compilation is done before execution. | Compilation and execution take place simultaneously. |
| Speed | Comparatively faster | Slower |
| Memory | Memory requirement is more due to the creation of object code. | It requires less memory as it does not create intermediate object code. |
| Errors | Display all errors after compilation, all at the same time. | Displays error of each line one by one. |
| Error detection | Difficult | Easier comparatively |
| Pertaining Programming languages | C, C++, C#, Scala, typescript uses compiler. | PHP, Perl, Python, Ruby uses an interpreter. |

**Phases of a Compiler**

[What are the Phases of Compiler Design?](https://www.guru99.com/compiler-design-phases-of-compiler.html#1)

There are two major phases of compilation, which in turn have many parts. Each of them take input from the output of the previous level and work in a coordinated way.



**Analysis Phase –** An intermediate representation is created from the give source code :

1. Lexical Analysis
2. Syntax Analysis
3. Semantic Analysis
4. Intermediate Code Generation

Lexical analyzer divides the program into “tokens”, Syntax analyzer recognizes “sentences” in the program using syntax of language and Semantic analyzer checks static semantics of each construct. Intermediate Code Generator generates “abstract” code.
**Synthesis Phase –** Equivalent target program is created from the intermediate representation. It has two parts:

1. Code Optimization
2. Code Generation

Code Optimizer optimizes the abstract code, and final Code Generator translates abstract intermediate code into specific machine instructions.

 **Phase 1: Lexical Analysis**

Lexical Analysis is the first phase when compiler scans the source code. This process can be left to right, character by character, and group these characters into tokens.

Here, the character stream from the source program is grouped in meaningful sequences by identifying the tokens. It makes the entry of the corresponding tickets into the symbol table and passes that token to next phase.

The primary functions of this phase are:

* Identify the lexical units in a source code
* Classify lexical units into classes like constants, reserved words, and enter them in different tables. It will Ignore comments in the source program
* Identify token which is not a part of the language

**Example**:

x = y + 10

Tokens

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

**Phase 2: Syntax Analysis**

Syntax analysis is all about discovering structure in code. It determines whether or not a text follows the expected format. The main aim of this phase is to make sure that the source code was written by the programmer is correct or not.

Syntax analysis is based on the rules based on the specific programing language by constructing the parse tree with the help of tokens. It also determines the structure of source language and grammar or syntax of the language.

Here, is a list of tasks performed in this phase:

* Obtain tokens from the lexical analyzer
* Checks if the expression is syntactically correct or not
* Report all syntax errors
* Construct a hierarchical structure which is known as a parse tree

**Example**

Any identifier/number is an expression

If x is an identifier and y+10 is an expression, then x= y+10 is a statement.

Consider parse tree for the following example

(a+b)\*c



In Parse Tree

* Interior node: record with an operator filed and two files for children
* Leaf: records with 2/more fields; one for token and other information about the token
* Ensure that the components of the program fit together meaningfully
* Gathers type information and checks for type compatibility
* Checks operands are permitted by the source language

**Phase 3: Semantic Analysis**

Semantic analysis checks the semantic consistency of the code. It uses the syntax tree of the previous phase along with the symbol table to verify that the given source code is semantically consistent. It also checks whether the code is conveying an appropriate meaning.

Semantic Analyzer will check for Type mismatches, incompatible operands, a function called with improper arguments, an undeclared variable, etc.

Functions of Semantic analyses phase are:

* Helps you to store type information gathered and save it in symbol table or syntax tree
* Allows you to perform type checking
* In the case of type mismatch, where there are no exact type correction rules which satisfy the desired operation a semantic error is shown
* Collects type information and checks for type compatibility
* Checks if the source language permits the operands or not

**Example**

float x = 20.2;

float y = x\*30;

In the above code, the semantic analyzer will typecast the integer 30 to float 30.0 before multiplication

**Phase 4: Intermediate Code Generation**

Once the semantic analysis phase is over the compiler, generates intermediate code for the target machine. It represents a program for some abstract machine.

Intermediate code is between the high-level and machine level language. This intermediate code needs to be generated in such a manner that makes it easy to translate it into the target machine code.

**Functions on Intermediate Code generation:**

* It should be generated from the semantic representation of the source program
* Holds the values computed during the process of translation
* Helps you to translate the intermediate code into target language
* Allows you to maintain precedence ordering of the source language
* It holds the correct number of operands of the instruction

**Example**

For example,

total = count + rate \* 5

Intermediate code with the help of address code method is:

t1 := int\_to\_float(5)

t2 := rate \* t1

t3 := count + t2

total := t3

**Phase 5: Code Optimization**

The next phase of is code optimization or Intermediate code. This phase removes unnecessary code line and arranges the sequence of statements to speed up the execution of the program without wasting resources. The main goal of this phase is to improve on the intermediate code to generate a code that runs faster and occupies less space.

**The primary functions of this phase are:**

* It helps you to establish a trade-off between execution and compilation speed
* Improves the running time of the target program
* Generates streamlined code still in intermediate representation
* Removing unreachable code and getting rid of unused variables
* Removing statements which are not altered from the loop

**Example:**

Consider the following code

a = intofloat(10)

b = c \* a

d = e + b

f = d

Can become

b =c \* 10.0

f = e+b

**Phase 6: Code Generation**

Code generation is the last and final phase of a compiler. It gets inputs from code optimization phases and produces the page code or object code as a result. The objective of this phase is to allocate storage and generate relocatable machine code.

It also allocates memory locations for the variable. The instructions in the intermediate code are converted into machine instructions. This phase coverts the optimize or intermediate code into the target language.

The target language is the machine code. Therefore, all the memory locations and registers are also selected and allotted during this phase. The code generated by this phase is executed to take inputs and generate expected outputs.

**Example:**

a = b + 60.0

Would be possibly translated to registers.

MOVF a, R1

MULF #60.0, R2

ADDF R1, R2

All in one example:

**Symbol Table Management**

A symbol table contains a record for each identifier with fields for the attributes of the identifier. This component makes it easier for the compiler to search the identifier record and retrieve it quickly. The symbol table also helps you for the scope management. The symbol table and error handler interact with all the phases and symbol table update correspondingly.

**Error Handling Routine:**

In the compiler design process error may occur in all the below-given phases:

* Lexical analyzer: Wrongly spelled tokens
* Syntax analyzer: Missing parenthesis
* Intermediate code generator: Mismatched operands for an operator
* Code Optimizer: When the statement is not reachable
* Code Generator: Unreachable statements
* Symbol tables: Error of multiple declared identifiers

Most common errors are invalid character sequence in scanning, invalid token sequences in type, scope error, and parsing in semantic analysis.

The error may be encountered in any of the above phases. After finding errors, the phase needs to deal with the errors to continue with the compilation process. These errors need to be reported to the error handler which handles the error to perform the compilation process. Generally, the errors are reported in the form of message.

**Summary**

* Compiler operates in various phases each phase transforms the source program from one representation to another
* Six phases of compiler design are 1) Lexical analysis 2) Syntax analysis 3) Semantic analysis 4) Intermediate code generator 5) Code optimizer 6) Code Generator
* Lexical Analysis is the first phase when compiler scans the source code
* Syntax analysis is all about discovering structure in text
* Semantic analysis checks the semantic consistency of the code
* Once the semantic analysis phase is over the compiler, generate intermediate code for the target machine
* Code optimization phase removes unnecessary code line and arranges the sequence of statements
* Code generation phase gets inputs from code optimization phase and produces the page code or object code as a result
* A symbol table contains a record for each identifier with fields for the attributes of the identifier
* Error handling routine handles error and reports during many phases