Introduction to Compiler

Muhammad Hamza
Lecturer, CS
GDC, KTS, Haripur

Objective of Compiler Design Course

- The primary objective is that at the end of the course the students must be quite comfortable with the concepts related to compilers
- We can use language processing technology for various software developments

- We can design, develop, understand, modify/enhance, and maintain compilers for (even complex!) programming languages

- To introduce students to the concepts underlying the design and implementation of language processors.

Objectives cont..

- A fair idea about modular programming.
- What language processor are and what functionality do they provide to the user.

Introduction to Compiler

- Compiler is a program which translates a program written in one language (the source language) to an equivalent program in other language (the target language). Usually the source language is a high level language like Java, C, Fortran etc. whereas the target language is machine code or "code" that a computer's processor understands.

OR

- It is a program which translates a high level language program into a machine language program

Continue

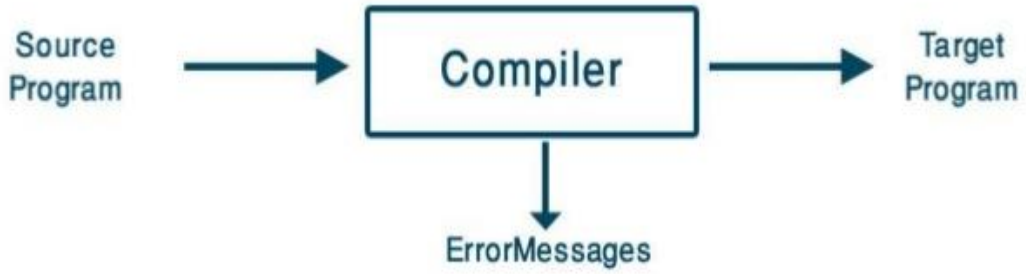
- The most common reason for transforming source code is to create an executable program.

- Any program written in a high level programming language must be translated to object code before it can be executed

Compiler Cont..

- The compiler reports to its user the presence of errors in the source also.
- There are thousands of source languages e.g. C, C++, Pascal , Fortran etc.
- A target language may be another programming language or a machine language.

Compiler Design



Compiler Cont..

- Advantages of using a compiler

- Source code is not included, therefore compiled code is more secure .
- Tends to produce faster code than interpreting source code
- Produces an executable file, and therefore the program can be run without need of the source code

Compiler cont..

Disadvantages of using a compiler

- Object code needs to be produced before a final executable file, this can be a slow process
- The source code must be 100% correct for the executable file to be produced

Interpreter

- An interpreter program executes other programs directly, running through program code and executing it line-by-line.
- As it analyses every line, an interpreter is slower than running compiled code but it can take less time to interpret program code than to compile and then run it — this is very useful when prototyping and testing code..

Interpreter and Compiler

- An interpreter is faster than a compiler because it has fewer stages e.g. no optimization stage.
- An interpreter works on one instruction as a time
- it produces the results of the computations as it translates the source instructions.

Cont..

- A compiler is more complicated than an interpreter, simply because it has more stages
- for doing analysis and optimizations, and because it focuses on the translation of the
- entire program.

However, the resulting target program translation is more efficient when it is executed.

Assembler

Assembler

- An assembler translates assembly language into machine code. Assembly language consists of mnemonics for machine opcodes so assemblers perform a 1:1 translation from mnemonic to a direct instruction.

- Conversely, one instruction in a high level language will translate to one or more instructions at machine level.

Advantages of Assembler:

- Very fast in translating assembly language to machine code as 1 to 1 relationship
- Assembly code is often very efficient (and therefore fast) because it is a low level language
- Assembly code is fairly easy to understand due to the use of English-like mnemonics

Disadvantages of Assembler:

- Assembly language is written for a certain instruction set and/or processor
- Assembly tends to be optimized for the hardware it's designed for, meaning it is often incompatible with different hardware
- Lots of assembly code is needed to do relatively simple tasks, and complex programs require lots of programming time

History of compiler

- The concept of a compiler was developed by Grace Hopper, an American computer scientist, in the 1950s.
- She created the first compiler, called the A-0 System, which translated mathematical notation into machine code.
- Hopper's invention revolutionized programming by allowing programmers to write code in human-readable languages rather than machine code, making software development faster and more accessible.
- Her pioneering work laid the foundation for modern compiler technology, which continues to be essential in computer programming today.

History of compiler

- The first compiler was written by Grace Hopper, in 1952, for the A-0 System language. The term compiler was used by Hopper.
- The A-0 functioned more as a loader or linker than the modern notion of a compiler.
- The first compiler were developed by Alick Glennie in 1952 for the Mark 1 computer at the University of Manchester and is considered by some to be the first compiled programming language.

History of Compiler

- The FORTRAN team led by John W. Backus at IBM is generally credited as having introduced the first complete compiler, in 1957.

How does a Compiler Work

- A compiler translates high-level programming code written by humans into machine-readable instructions that computers can understand and execute.
- It first analyzes the structure of the code and syntax to ensure correctness, then optimizes it for efficiency.
- Afterward, the compiler generates machine code, consisting of binary instructions modified to the computer's architecture.
- This process automates the translation of complex code, making programming more accessible and efficient for developers while enabling computers to execute tasks accurately.

Types of Compiler

Single-Pass Compiler – A single-pass compiler processes the source code in a single pass, from start to finish, generating machine code as it goes. It is efficient but may not catch all errors or perform extensive optimization.

Multi-Pass Compiler – A multi-pass compiler makes multiple passes over the source code, analyzing it in different stages. This allows for more thorough error checking and optimization but can be slower than a single-pass compiler.

Types of Compiler

Just-In-Time (JIT) Compiler – A JIT compiler translates code into machine language while the program is running, on-the-fly. It is used in languages like Java and JavaScript to improve performance by converting code as needed during execution.

Ahead-of-Time (AOT) Compiler – An AOT compiler translates code into machine language before the program is run, producing an executable file. This approach is common in languages like C and C++, providing fast execution but requiring compilation before running the program.

How long does it take to learn to write a basic compiler?

- Learning to write a basic compiler can vary in time depending on factors like prior programming experience and the complexity of the compiler.
- It might take several months to a year or more to understand the concepts and develop the skills needed to create a basic compiler.
- This process involves learning about lexical analysis, parsing, code generation, and optimization techniques, as well as gaining proficiency in a programming language and understanding computer architecture.
- Practice, experimentation, and learning from resources like books, tutorials, and online courses can help speed up the learning process.

What is syntax tree in compiler design?

- A syntax tree in compiler design is a hierarchical representation of the structure of source code written in a programming language.
- It visually organizes the elements of code, such as expressions, statements, and declarations, into a tree-like structure based on the grammar rules of the language.
- Each node in the tree represents a specific syntactic construct, while the edges between nodes indicate relationships between them, such as parent-child or sibling relationships.

Is compiler a system software or application software?

- A compiler is considered system software. System software is a type of software that provides essential functions for a computer system to operate, manage resources, and support the execution of other software applications.
- A compiler falls into this category because it is responsible for translating high-level programming code into machine-readable instructions that computers can execute.
- Without a compiler, programmers would not be able to create software applications.

What is compiler architecture?

- Compiler architecture refers to the overall design and structure of a compiler. It consists of various components and stages involved in the compilation process, from analyzing the source code to generating machine-readable output.
- Compiler architecture generally includes modules for lexical analysis (breaking down code into tokens), syntax analysis (parsing the structure of the code), semantic analysis (checking for meaning and correctness), optimization (improving the efficiency of the code), and code generation (producing machine code).
- Each of these components interacts with one another in a coordinated manner to translate high-level programming languages into machine-executable instructions efficiently and accurately.

Compiler Design - Overview

- Computers are a balanced mix of software and hardware. Hardware is just a piece of mechanical device and its functions are being controlled by a compatible software.
- Hardware understands instructions in the form of electronic charge, which is the counterpart of binary language in software programming.
- Binary language has only two alphabets, 0 and 1. To instruct, the hardware codes must be written in binary format, which is simply a series of 1s and 0s.
- It would be a difficult and cumbersome task for computer programmers to write such codes, which is why we have compilers to write such codes.

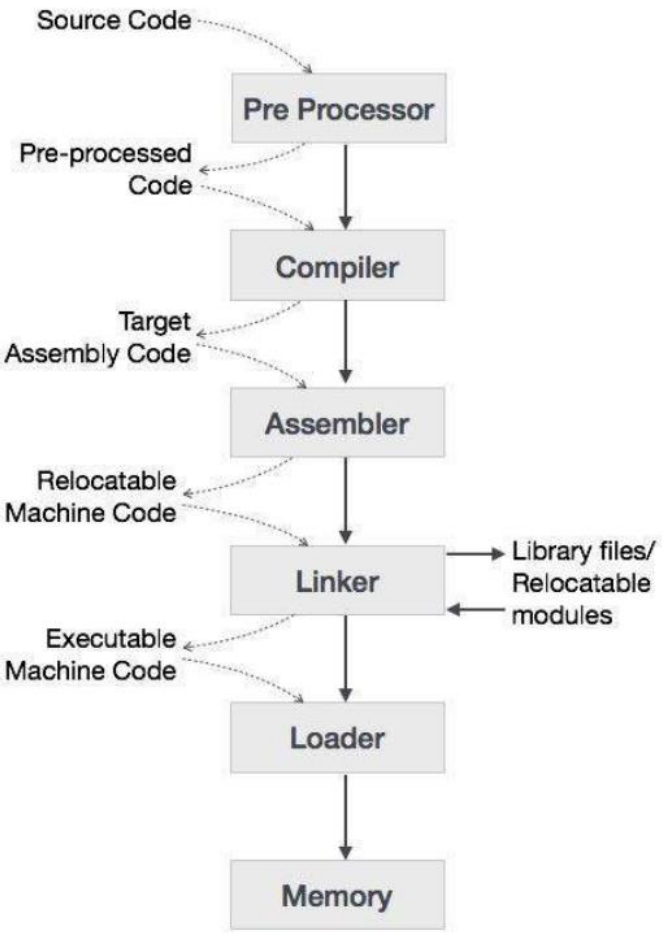
How a program in C Language is executed using Compiler

- User writes a program in C language (high-level language).
- The C compiler, compiles the program and translates it to assembly program (low-level language).
- An assembler then translates the assembly program into machine code (object).
- A linker tool is used to link all the parts of the program together for execution (executable machine code).
- A loader loads all of them into memory and then the program is executed.

Language Processing System

- We have learnt that any computer system is made of hardware and software.
- The hardware understands a language, which humans cannot understand.
- So we write programs in high-level language, which is easier for us to understand and remember.
- These programs are then fed into a series of tools and OS components to get the desired code that can be used by the machine. This is known as Language Processing System.

Language Processing System



Preprocessor

A preprocessor, generally considered as a part of compiler, is a tool that produces input for compilers. It deals with macro-processing, augmentation, file inclusion, language extension, etc.

Interpreter

An interpreter, like a compiler, translates high-level language into low-level machine language. The difference lies in the way they read the source code or input. A compiler reads the whole source code at once, creates tokens, checks semantics, generates intermediate code, executes the whole program and may involve many passes. In contrast, an interpreter reads a statement from the input, converts it to an intermediate code, executes it, then takes the next statement in sequence. If an error occurs, an interpreter stops execution and reports it. whereas a compiler reads the whole program even if it encounters several errors.

Assembler

An assembler translates assembly language programs into machine code. The output of an assembler is called an object file, which contains a combination of machine instructions as well as the data required to place these instructions in memory.

Linker

- Linker is a computer program that links and merges various object files together in order to make an executable file.
- All these files might have been compiled by separate assemblers. The major task of a linker is to search and locate referenced module/routines in a program and to determine the memory location where these codes will be loaded, making the program instruction to have absolute references.

Loader

- Loader is a part of operating system and is responsible for loading executable files into memory and execute them. It calculates the size of a program (instructions and data) and creates memory space for it. It initializes various registers to initiate execution.

Cross-compiler

A compiler that runs on platform (A) and is capable of generating executable code for platform (B) is called a cross-compiler.

Source-to-source Compiler

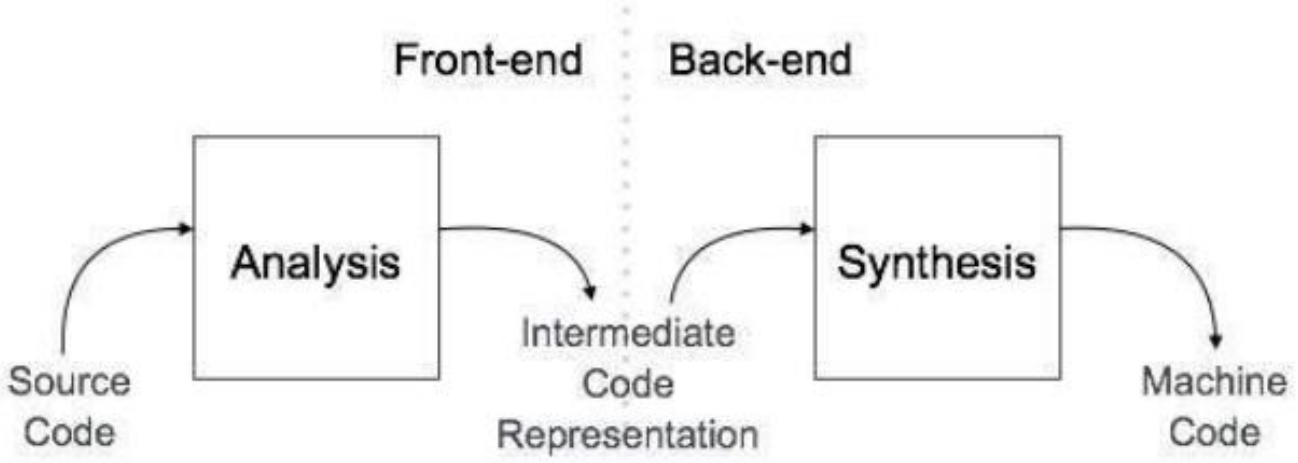
A compiler that takes the source code of one programming language and translates it into the source code of another programming language is called a source-to-source compiler.

Parts of Compiler

A compiler can broadly be divided into two parts / phases based on the way they compile.

1. Analysis Phase
2. Synthesis Phase

Parts of Compiler



Analysis Phase

- Known as the front-end of the compiler, the **analysis** phase of the compiler reads the source program, divides it into core parts and then checks for lexical, grammar and syntax errors.
- The analysis phase generates an intermediate representation of the source program and symbol table, which should be fed to the Synthesis phase as input.

Synthesis Phase

- Known as the back-end of the compiler, the **synthesis** phase generates the target program with the help of intermediate source code representation and symbol table.
- A compiler can have many phases and passes.
- **Pass** : A pass refers to the traversal of a compiler through the entire program.
- **Phase** : A phase of a compiler is a distinguishable stage, which takes input from the previous stage, processes and yields output that can be used as input for the next stage. A pass can have more than one phase.

The Analysis and Synthesis Model of Compilation

- There are two parts of compilation
- Analysis
- Synthesis

The analysis part breakup the source program into constituent pieces and creates an intermediate representation of the source program.

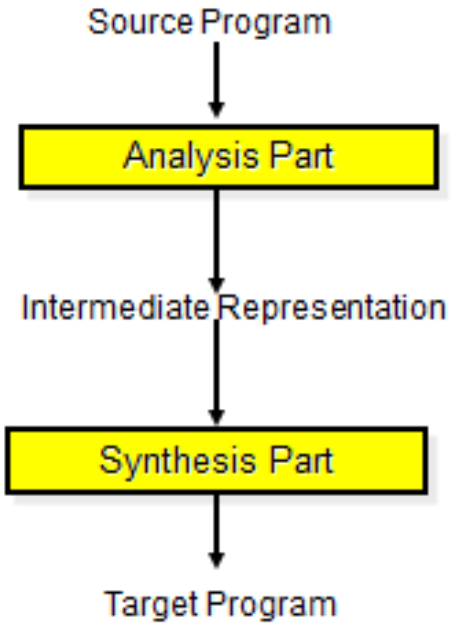
Analysis Phase

In compiling analysis consists of three phases.

- Linear
- Hierarchical
- Semantic

Analysis-Synthesis model

- The Analysis and Synthesis Model of Compilation.



Analysis Cont…

Linear Analysis

- In linear analysis the stream of characters making up the source program is read from left to right and grouped into tokens that are sequences of characters having collective meaning.
- linear analysis is also called scanning or lexical analysis.
- For example the linear analysis of the following assignment statement will be as follow.

Analysis Cont..

Position = initial + rate * 60

The characters would be grouped into the following tokens.

The identifier , position.

The assignment statement, =

The identifier, initial

The plus sign, +

The identifier, rate.

The multiplication sign

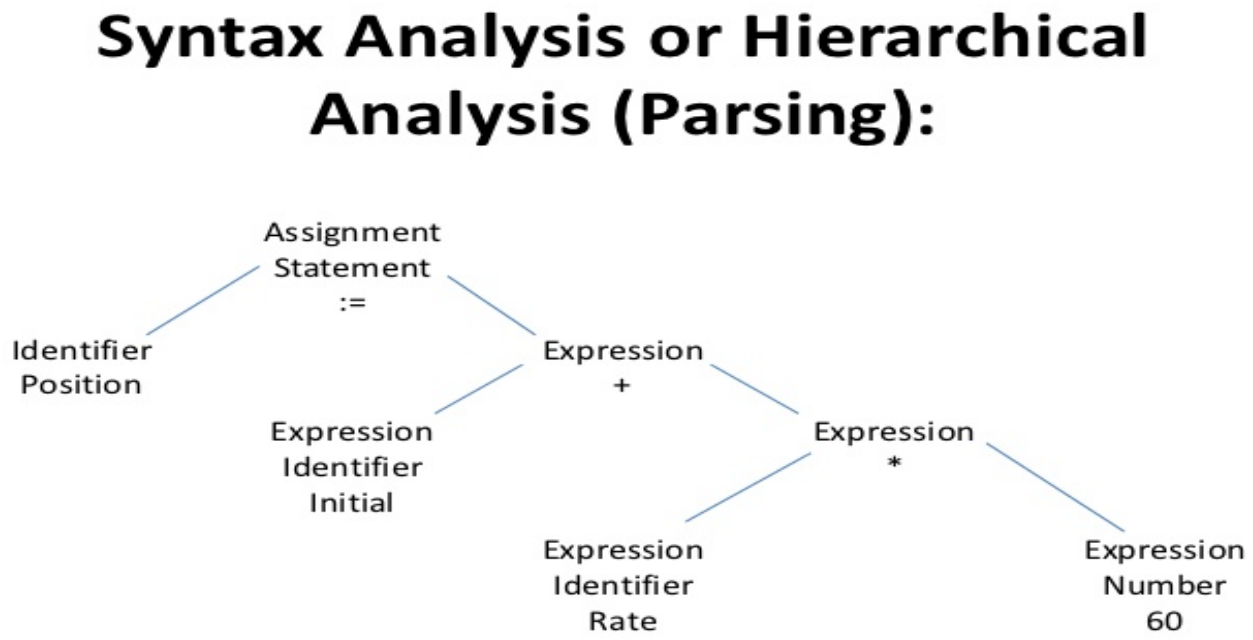
The number, 60.

Hierarchical Analysis

- During this phase tokens of the source program grouped into grammatical phrases. These phrases are then used by the compiler to synthesize the output. The grammatical phrases of the source program are represented by a parse tree.

Hierarchical Analysis

- A syntax tree or parse tree



Semantic Analysis

- During this analysis phase certain checks are performed to ensure that the components of a program fit together meaningfully. It check the source program for semantic errors

Cont…

- An important component of semantic analysis is type checking.
- Here are the compiler checks that each operator has operands that are permitted by the source language specification.

Synthesis part

- The synthesis part constructs the desired target program from the intermediate representation.

The Grouping of Phases

Front and Back Ends:

- The phases are collected into a front end and a back end.
- The front end consists of those phases that depend primarily on the source language and are largely independent of the target machine.

Front and Back Ends:

- These normally include lexical and syntactic analysis, the creating of the symbol table, semantic analysis, and the generation of intermediate code.

- A certain among of code optimization can be done by the front end as well.

Front and Back Ends:

- The front end also includes the error handling that goes along with each of these phases.

Front and Back Ends:

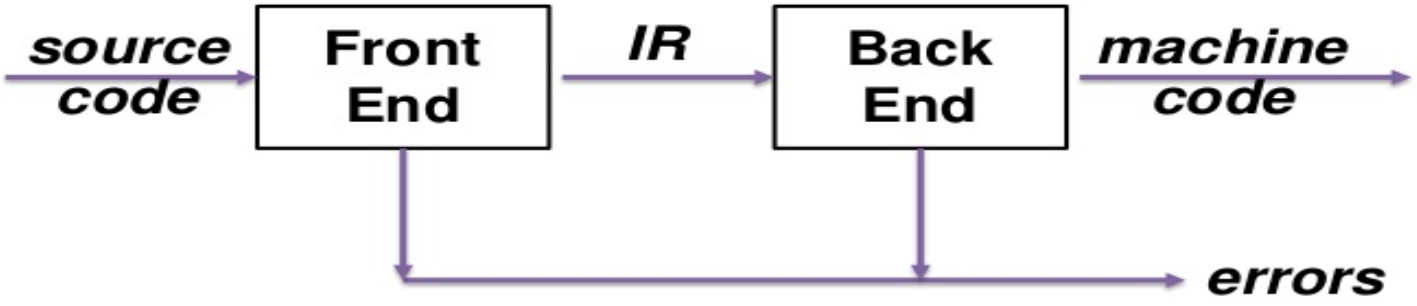
- The back end includes those portions of the compiler that depend on the target machine.

- And generally, these portions do not depend on the source language, depend on just the intermediate language.

Front and Back Ends:

- In the back end, we find aspects of the code optimization phase, and we find code generation, along with the necessary error handling and symbol table operations.

The Analysis-Synthesis Model of Compilation:



Linker

Linker:

- In high level languages, some built in header files or libraries are stored.
- These libraries are predefined and these contain basic functions which are essential for executing the program.
- These functions are linked to the libraries by a program called Linker.
- If linker does not find a library of a function then it informs to compiler and then compiler generates an error.

Linker Cont..

- The compiler automatically invokes the linker as the last step in compiling a program.
- Not built in libraries, it also links the user defined functions to the user defined libraries.
- Usually a longer program is divided into smaller subprograms called modules. And these modules must be combined to execute the program.
- The process of combining the modules is done by the linker.

Loader

Loader:

- Loader is a program that loads machine codes of a program into the system memory.
- A loader is the part of an Operating System that is responsible for loading programs.

Loader Cont..

- Loading a program involves reading the contents of executable file into memory.
- After completion of the loading process the operating system passes control the loaded program for its execution.

Compiler Construction Tools

Compiler-Construction Tools:

- Shortly after the first compilers were written, systems to help with the compiler-writing process appeared.
- These systems have often been referred to as
  - Compiler-compilers,
  - Compiler-generators,
  - Or Translator-writing systems.

Compiler-Construction Tools:

- Some general tools have been created for the automatic design of specific compiler components.
- These tools use specialized languages for specifying and implementing the component, and many use algorithms that are quite sophisticated.

Tools Cont..

The two main types of tools used in compiler production are:

- A lexical Analyser Generator.
- A syntax Analyser Generator(Parser Generator)

Compiler-Construction Tools:

- **Parser generators**
  - These produce syntax analyzers, normally from input that is based on a context-free grammar.
  - In early compilers, syntax analysis consumed not only a large fraction of the running time of a compiler, but a large fraction of the intellectual effort of writing a compiler.
  - This phase is considered one of the easiest to implement.

- Yacc is a parser generator Yacc was developed by Stephen C. Johnson at AT&T for the Unix operating system.The name is an acronym for "Yet Another Compiler Compiler."

- Johnson worked on Yacc in the early 1970s at Bell Labs. Because Yacc was the default compiler generator on most Unix systems, it was widely distributed and used.

Tools used for compiler design

- The compiler generated by YACC requires a lexical analyzer. Lexical analyzer generators, such as lex or flex are widely available.
- YACC supports a general type of bottom up parsing.

YACC Cont..

- An object oriented version of YACC (YACC++) is also available and have actions written in C++

Lex(Lexical Generator)

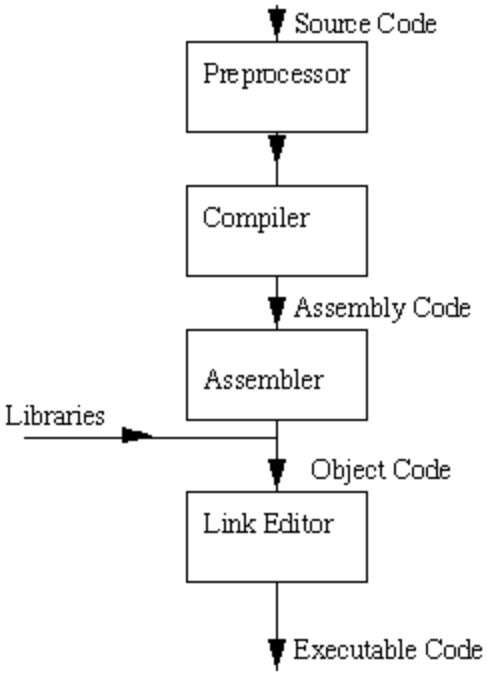
Lex is a program generator designed for lexical processing of character input streams.

- It accepts as its input source program language and produces as output a lexical analyser.

Lex Cont..

- An object oriented version of Lex is also available(LEX++) that have actions written in C++.

The C Compilation Model



The Preprocessor

- The Preprocessor accepts source code as input and is responsible for

- removing comments
- interpreting special preprocessor directives denoted by #.

For example

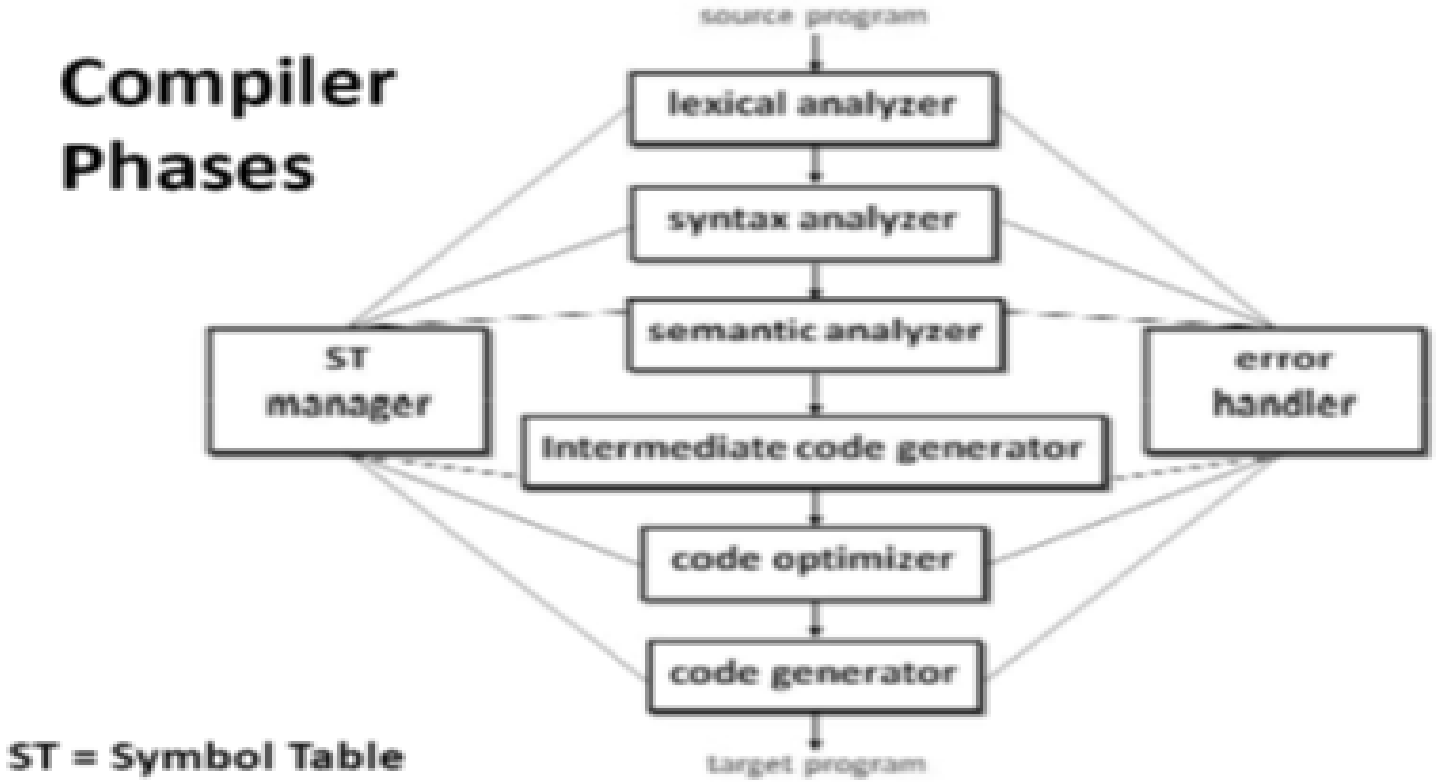
- #include -- includes contents of a named file. Files usually called header files. e.g
- #include <math.h> -- standard library maths file.
- #include <stdio.h> -- standard library I/O file
- #define -- defines a symbolic name or constant. Macro substitution.
- #define MAX_ARRAY_SIZE 100

The C Compilation Model

- C Compiler
- The C compiler translates source to assembly code. The source code is received from the preprocessor.
- Assembler
- The assembler creates object code.

Phases of Compiler

- The Phases of a Compiler.



Phases of Compiler

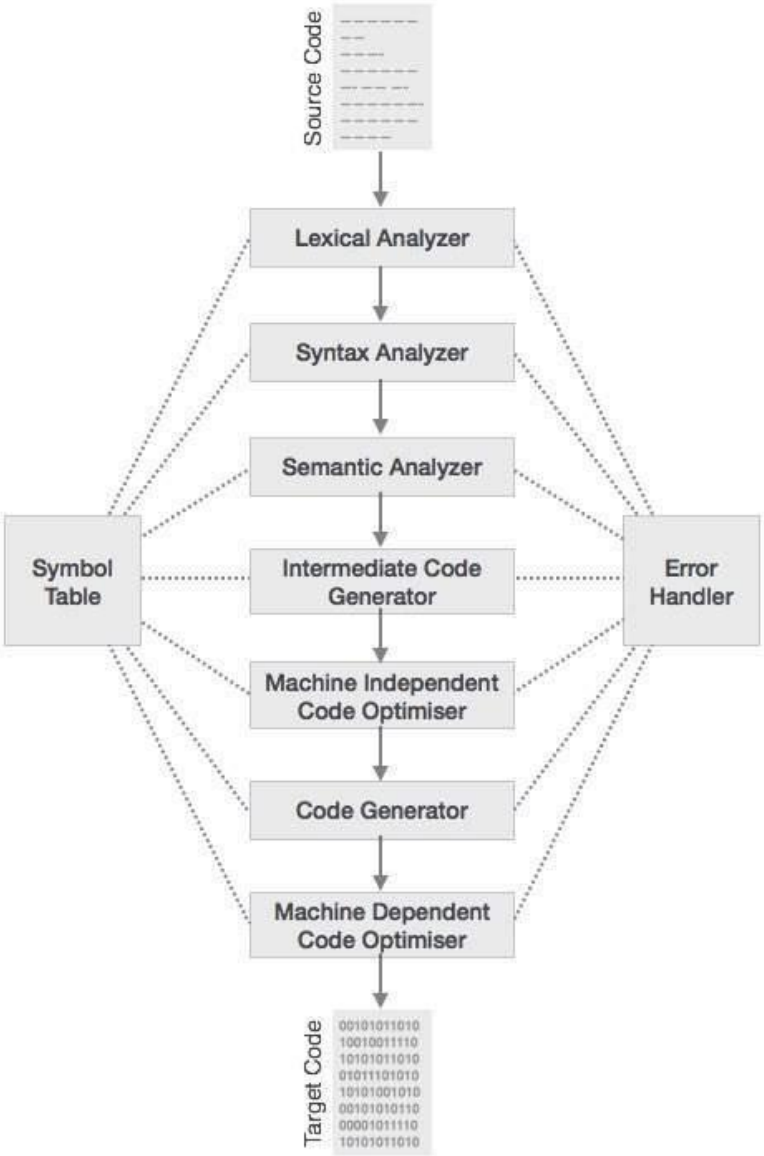
- The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler. Let us understand the phases of a compiler.
- There are 6 phases in a compiler. Each of this phase help in converting the high-level langue the machine code.

Phases of Compiler

The phases of a compiler are:

- Lexical analysis
- Syntax analysis
- Semantic analysis
- Intermediate code generator
- Code optimizer
- Code generator

Phases of Compiler



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.

Phase 1: Lexical Analysis

- The word “lexical” in the traditional sense means “pertaining to words”.
- In terms of programming languages, words are objects like variable names, numbers, keywords
- etc. Such words are traditionally called tokens.

Phase 1: Lexical Analysis

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

Phase 1: Lexical Analysis

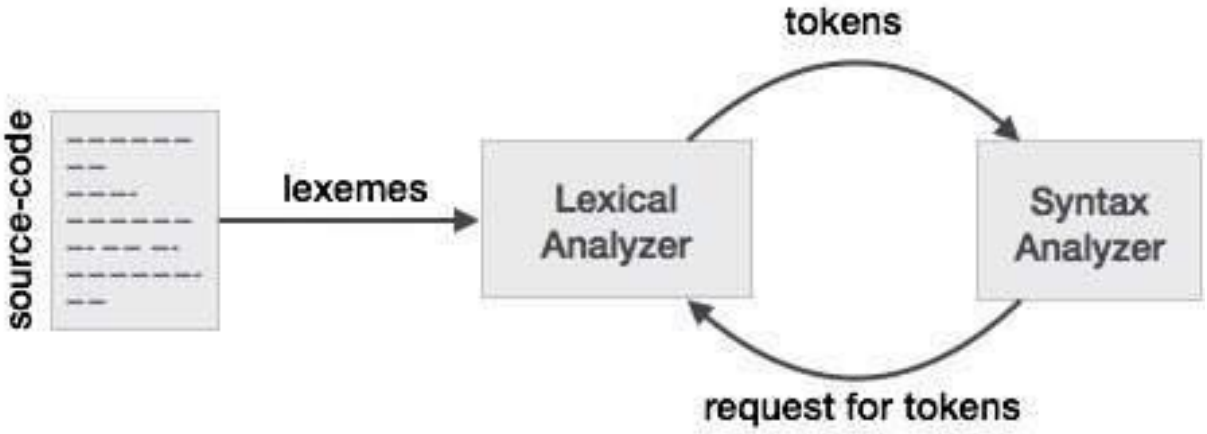
- Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences.
- The lexical analyzer breaks these syntaxes into a series of tokens

Removes whitespace or comments in the source code.

Phase 1: Lexical Analysis

If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.

Phase 1: Lexical Analysis



Phase 1: Lexical Analysis

```
 int max (  int  x, int y)
{
    return (x > y? x : y);
}
```

Total number of tokens of this code will be:

" int"

"max"

"("

Phase 1: Lexical Analysis

“ int”

“ x”

“ ”

’

“int”

“y”

“)”

“{“

“return”

Phase 1: Lexical Analysis

“(“

“x”

“>”

“Y”

“ ?”

“x”

“:”

“Y”

Phase 1: Lexical Analysis

“)”

“.”

’

“}”

Phase 1: Lexical Analysis

Example:
x = y + 10

Tokens

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

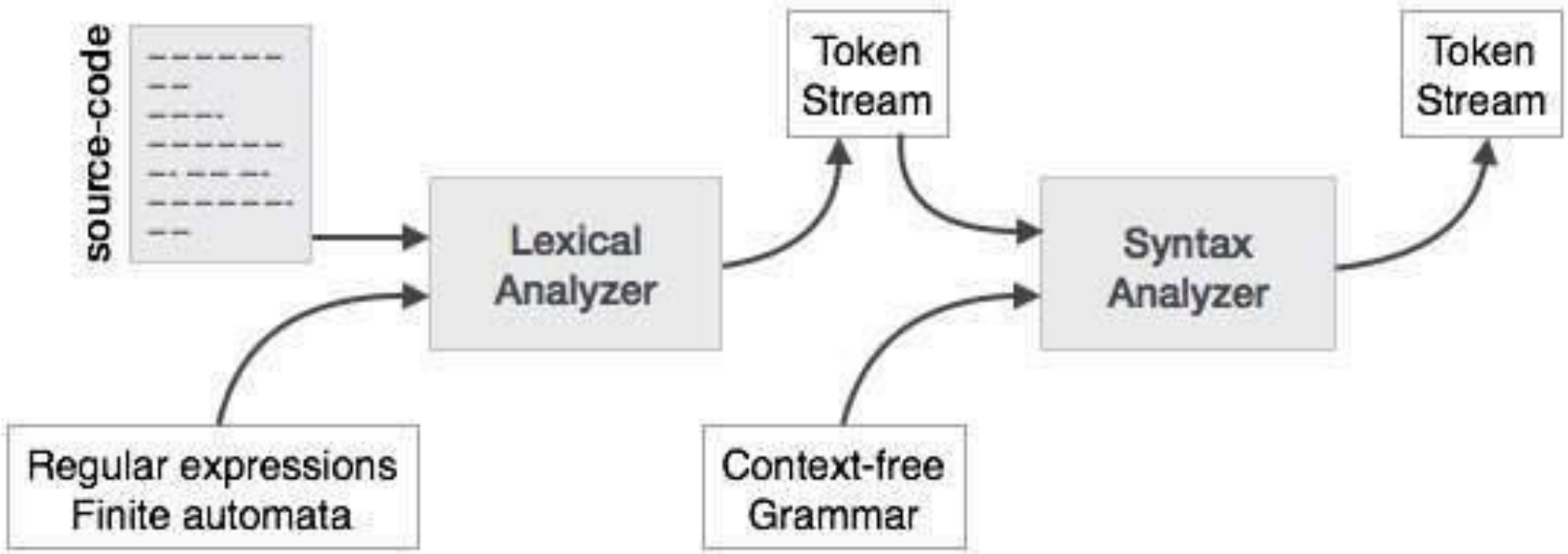
Phase 2: Syntax Analysis

- 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.

Phase 2: Syntax Analysis

- A syntax analyzer or parser takes the input from a lexical analyzer in the form of token streams. The parser analyzes the source code (token stream) against the production rules to detect any errors in the code. The output of this phase is a parse tree.

Phase 2: Syntax Analysis



Phase 2: Syntax Analysis

This way, the parser accomplishes two tasks, i.e., parsing the code, looking for errors and generating a parse tree as the output of the phase.

- Parsers are expected to parse the whole code even if some errors exist in the program.

Phase 2: Syntax Analysis

- This phase is also called the parsing phase. The following operations are performed in this phase: Obtain tokens from lexical analyzer.
- check whether the expression is syntactically correct.
- report syntax errors, if any.
- construct hierarchical structures called parse trees. These parse trees represent the syntactic structure of the program. Consider the statement X = Y + Z.

Phase 2: Syntax Analysis

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

Phase 2: Syntax Analysis

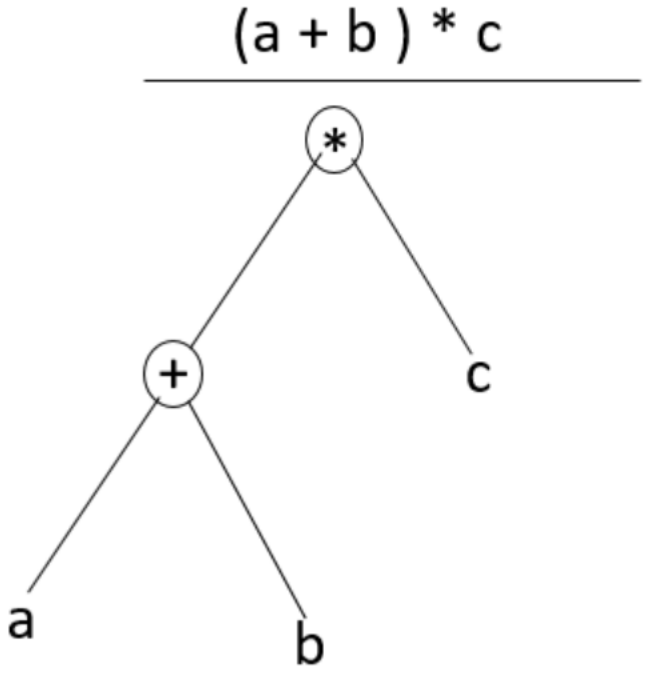
Example

- 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

Phase 2: Syntax Analysis

Solution



Phase 3: Semantic Analysis

- Semantics of a language provide meaning to its constructs, like tokens and syntax structure.
- Semantics help interpret symbols, their types, and their relations with each other. Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not.

Phase 3: Semantic Analysis

For example:

int a = “value”;

- should not issue an error in lexical and syntax analysis phase, as it is lexically and structurally correct, but it should generate a semantic error as the type of the assignment differs. These rules are set by the grammar of the language and evaluated in semantic analysis.

Functions of Semantic Analysis

The following tasks should be performed in semantic analysis:

- Scope resolution
- Type checking
- Array-bound checking

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.

Phase 3: Semantic Analysis

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

Phase 3: Semantic Analysis

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

Semantic Errors

We have mentioned some of the semantics errors that the semantic analyzer is expected to recognize:

- Type mismatch
- Undeclared variable
- Reserved identifier misuse.
- Multiple declaration of variable in a scope.
- Accessing an out of scope variable.
- Actual and formal parameter mismatch.

Semantic Errors

- Semantic errors are mistakes concerning the meaning of a program construct; they may be
- either type errors, logical errors or run-time errors:
- (a) Type errors occur when an operator is applied to an argument of the wrong type,
- or to the wrong number of arguments.

Semantic Errors

- Static semantic checks (done by the compiler) are performed at compile time
- Type checking
- Every variable is declared before used
- Check subroutine call arguments
- Check labels

Semantic Errors

- Dynamic semantic checks are performed at run time, and the compiler produces code that performs these checks
- Array subscript values are within bounds
- Arithmetic errors, e.g. division by zero
- Pointers are not dereferenced unless pointing to valid object
- A variable is used but hasn't been initialized

Semantic Analysis and Strong Typing

- A language is strongly typed "if (type) errors are always detected"
- Errors are either detected at compile time or at run time
- Examples of such errors are listed on previous slide
- Languages that are strongly typed are Ada, Java, ML, Haskell
- Languages that are not strongly typed are Fortran, Pascal, C/C++, Lisp

Errors

- Logical errors occur when a badly conceived program is executed, for example:
- while x = y do ... when x and y initially have the same value and the body of loop need not change the value of either x or y.
- (c) Run-time errors are errors that can be detected only when the program is executed,
- for example:
- var x : real; readln(x); writeln(1/x)
- which would produce a run time error if the user input 0.

Errors

- Syntax errors must be detected by a compiler and at least reported to the user (in a helpful
- way). If possible, the compiler should make the appropriate correction(s). Semantic errors are
- much harder and sometimes impossible for a computer to detect.

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.

Phase 4: Intermediate Code Generation

The syntax and semantic analysis generate a explicit intermediate representation of the source program.

The intermediate representation should have two important properties:

- It should be easy to produce,
- And easy to translate into target program.

Phase 4: Intermediate Code Generation

- ICR can be in a variety of forms:

One of the form is Three address code.

Three Address consists of a sequence of instructions, each of which has atmost three operands

Phase 4: Intermediate Code Generation

Functions of 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

Phase 4: Intermediate Code Generation

Example: 1

```
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 4: Intermediate Code Generation

Example: 2

- a = b + c * d;
- The intermediate code generator will try to divide this expression into sub-expressions and then generate the corresponding code.

- r1 = c * d;
- r2 = b + r1;
- r3 = r2 + r1;
- a = r3
- r being used as registers in the target program.

Phase 5: Code Optimization

- Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed.

- In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes. A code optimizing process must follow the three rules given below:

Phase 5: Code Optimization

- The output code must not, in any way, change the meaning of the program.
- Optimization should increase the speed of the program and if possible, the program should demand less number of resources.
- Optimization should itself be fast and should not delay the overall compiling process.

Phase 5: Code Optimization

Efforts for an optimized code can be made at various levels of compiling the process.

- At the beginning, users can change/rearrange the code or use better algorithms to write the code.

- After generating intermediate code, the compiler can modify the intermediate code by address calculations and improving loops.

Phase 5: Code Optimization

Optimization can be categorized broadly into two types :

- machine independent.
- machine dependent.

Phase 5: Code Optimization

- Machine-independent Optimization
- In this optimization, the compiler takes in the intermediate code and transforms a part of the code that does not involve any CPU registers and/or memory locations.
- For example:

```
do
{
   item = 10;
   value = value + item;
} while(value<100);
```

Phase 5: Code Optimization

- This code involves repeated assignment of the identifier item, which if we put this way:

```
Item = 10;
do
{
  value = value + item;
} while(value<100);
```

- should save the CPU cycles.

Phase 5: Code Optimization

Machine-dependent Optimization

- Machine-dependent optimization is done after the target code has been generated and when the code is transformed according to the target machine architecture. It involves CPU registers and may have absolute memory references

Phase 5: Code Optimization

- 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.

Phase 5: Code Optimization

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

Phase 5: Code Optimization

Example:
Consider the following code

```
a = intofloat(10)
b = c * a
d = e + b
f = d
```

It can also be written as:

```
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.

Phase 6: Code Generation

- 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.

Phase 6: Code Generation

Example

a = b + 60.0

Would be possibly translated to registers.

```
MOVF a, R1
MULF #60.0, R2
ADDF R1, R2
```

Symbol Table Management

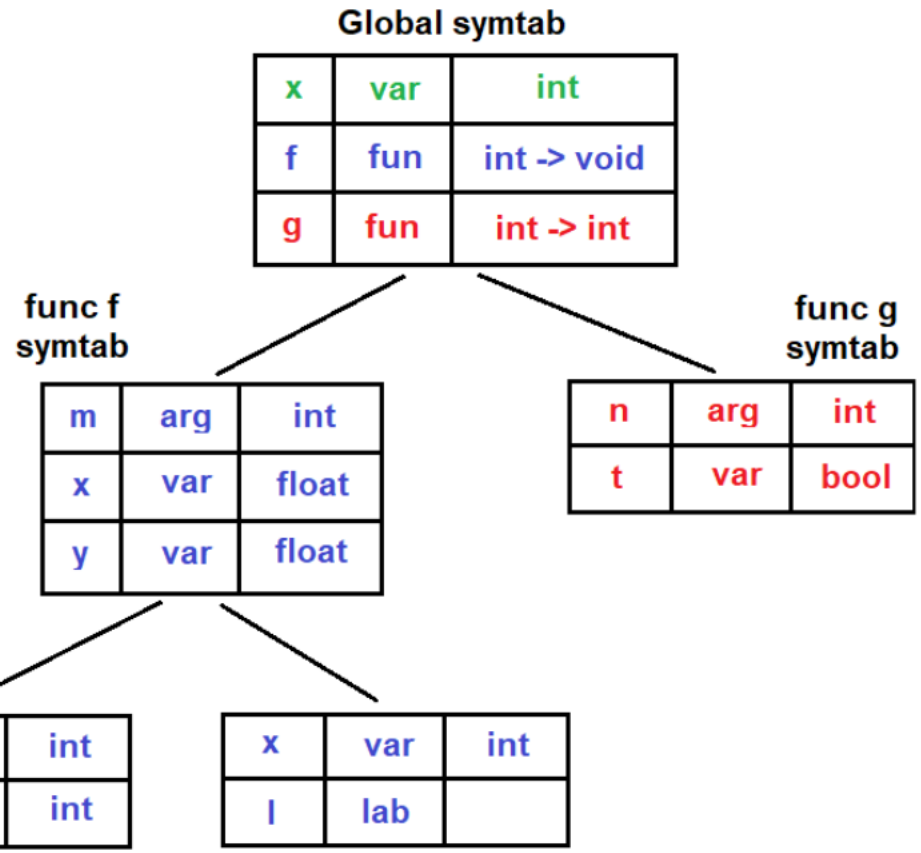
- A symbol table contains a record for each identifier with fields for the attributes of the identifier.
- This component makes it easier for the compiler to search the identifier record and retrieve it quickly.
- The symbol table also helps you for the scope management. The symbol table and error handler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

Symbol Table Management

```
int x;

void f(int m) {
  float x, y;
  ...
  { int i, j; ...; }
  { int x; l: ...; }
}

int g(int n) {
  bool t;
  ...;
}
```



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: When the memory is full or proper registers are not allocated
- Symbol tables: Error of multiple declared identifiers

Error Handling Routine

- 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.
- 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

Summary

- 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.