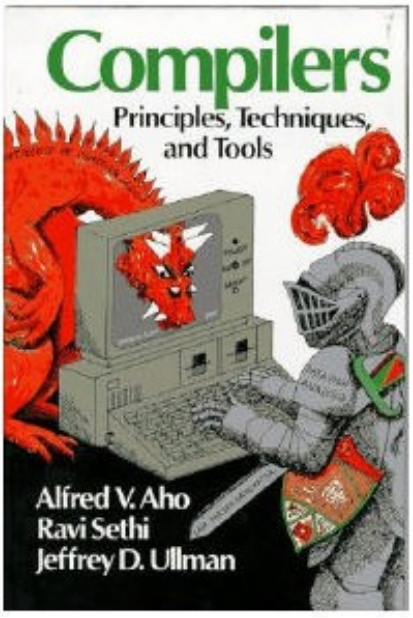
# Chapter # 1 Introduction to Compiler

Dr. Shaukat Ali

## Course Book

- The primary text for the course is: **Compilers – Principles, Techniques and Tools** by **Aho, Sethi and Ullman**
- This is also called the Dragon Book



# Why Study Compilers

**Reason #1: understand compilers and languages.**

- Understand the code structure.
- Understand language semantics.
- Understand relation between source code and generated machine code.
- Allow to become a better programmer and increase programmer productivity and increase portability

**Reason #2: nice balance of theory and practice.**

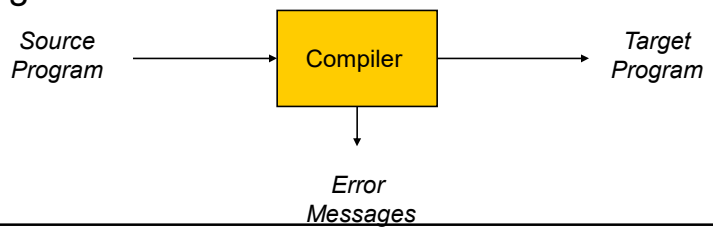
- Theory:
  - Mathematical models: regular expressions, automata, grammars, graphs.
  - Algorithms that use these models.
- Practice:
  - Apply theoretical notions to build a real compiler.

**Reason #3: programming experience.**

- Creating a compiler entails writing a large computer program which manipulates complex data structures and implement sophisticated algorithm
- Increasing programming capability

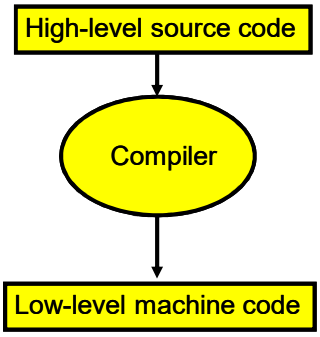
# Compilers

- A Compiler is a program (system software) that reads a program written in one language – ***the source language*** – and translates it into an equivalent program in another language – ***the target language***.
- During translation process, the compiler reports to its user the presence of errors in the source program.



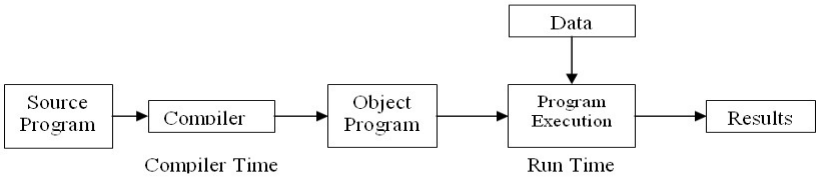
# Compilers

- Source language can be any high level computer programming language ranging from traditional programming language such as Fortran, C, Java etc to specialized language that have been written for a specific area of computer application such as LISP for AI etc.
- Target language may be another programming language (assembly language) or the machine language of a computer, depending upon the compiler.




# Compilation Process

- It takes the whole program at a time and either displays all of the possible errors in the program or creates an object program.
- The time at which the conversion of a source program to an object program occurs is called compile time.
- The object program is executed at run time.



## Properties of compilers

- It must generate a correct executable code
- The input program and the output program must be equivalent
  - The compiler should preserve the meaning of the input program
- Output program should run fast
- Compiler itself should be fast i.e., low compilation time
- Compiler should provide good diagnostics for programming errors

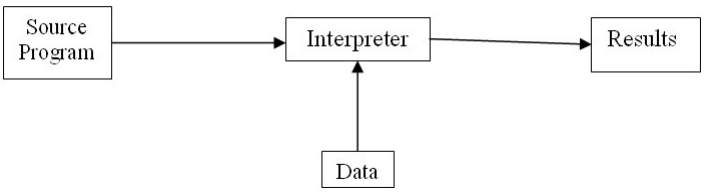
## Properties of compilers

- Compiler should support separate compilation
- Compiler should work well with debuggers
- Compile time should be maximally proportional to the code size

# Interpreter

- Interpreter is a system software that is used for the translation of high level language programs.
  - Directly execute the statements of source program rather than generating object code
- It is different from the compilers in a sense that:
  - It translates a program by taking one instruction at a time and produces the results before taking the next instruction.
  - It can identify only one error at a time.
  - It does not produces the object program.
  - Needs retranslation
    - Makes it slow than compilers by a factor of 10 times

# Interpreter



## Assembler

- Assembler is a translator (software) that particularly converts a program written in assembly language into machine language.
- Assembly language is called low-level language.
  - Because there is one to one correspondence between the assembly language statements and machine language statements.
  - Symbolic form of the machine language, makes it easy to translate
  - Compiler generates assembly language as its target language and assembler translate it into object code

## Linker

- Separate program often part of operating system
- Collects code in object file(s) into a file that is directly executable
- Object code produced by a compiler or assembler --- machine code that has not yet been linked
- Linker creates executable machine code
  - Connects an object program to the code of standard library functions and to resources supplied by operating system
    - For example, memory allocation and input and output devices etc.

## Loader

- The loader reads the reloadable machine code
  - Alters its addresses by adding the starting position of main memory block to them and loads the code into main memory

## The Context of a Compiler

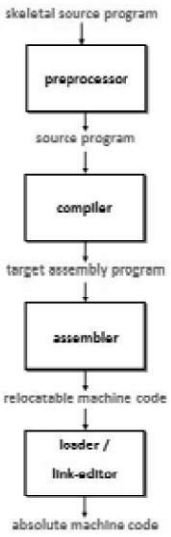
- In addition to compiler, several other programs may be required to create an executable target program.
  - A source program may be divided into modules stored in separate files. The task of collecting the source program is the responsibility of another program called preprocessor.
  - The target program created by the compiler may require further processing before it can be run.

## The Context of a Compiler

- The compiler creates the assembly code that is translated by an assembler into machine code.
- The linker together the machine code with some library routines into the code that actually run on the machine.

## The Context of a Compiler

**A Language-Processing System (Compilation)**



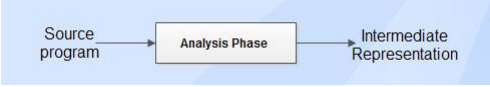
# Analysis and Synthesis Model

- The structure of compiler consists of two parts:
  - Analysis part
  - Synthesis part

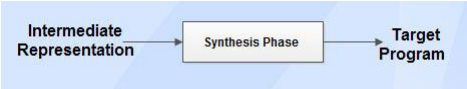
- Analysis part
  - Analysis part breaks the source program into constituent pieces and imposes a grammatical structure on them which further uses this structure to create an intermediate representation of the source program
  - It is also termed as front end of compiler

# Analysis and Synthesis Model

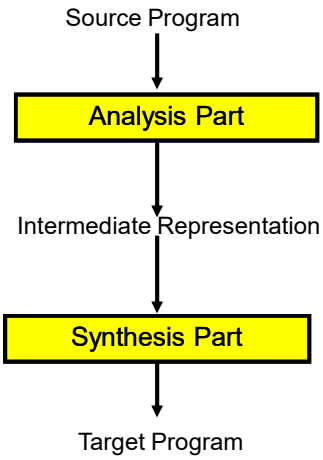
  - Information about the source program is collected and stored in a data structure called symbol table




- Synthesis part
  - Synthesis part takes the intermediate representation as input and transforms it to the target program
  - It is also termed as back end of compiler



# Analysis and Synthesis Model




# The Phases of a Compiler

- A compiler operates in phases, each representing distinct logical operation, each of which transforms the source program from one representation into another
  - A phase is a single module of compiler
  - A phase is a program which convert a source program from one representation/implementation into another representation/implementation
  - A logical activity that transform the source program from one representation into another representation
- In practice, some of the phases may be grouped together

# The Phases of Compiler

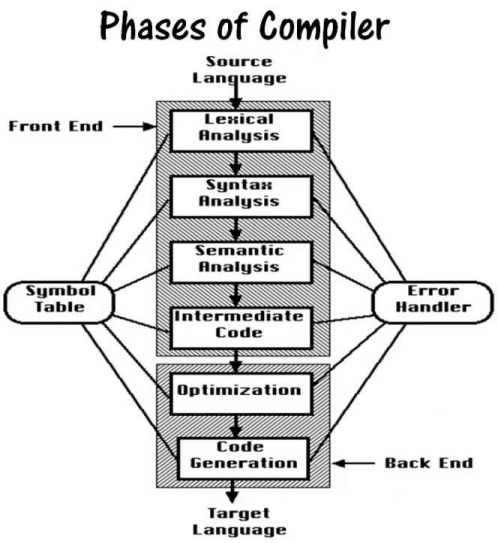
- A compiler consists of six phases:
- Formal phases – characterizes the basic activities of compiler
  - Lexical Analysis.
  - Syntax Analysis.
  - Semantic Analysis.
  - Intermediate Code Generation.
  - Code Optimizer.
  - Code Generation.

Analysis Portion

Synthesis Portion

- Informal phases – helps in performing the basic activities of compiler
  - Symbol-Table Management and Error Handling, that interact with the six phases are also informally considered as phases

# The Phases of a Compiler



# Lexical Analysis.

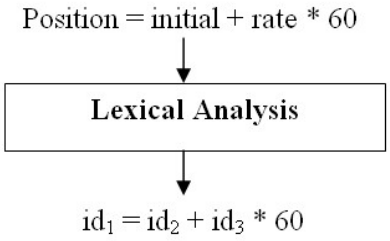
- It is also called Linear Analysis or Scanner
- It reads the stream of characters making up the source program from left-to-right and grouped into tokens (the sequence of characters having a collective meaning), meaningful units
- For example, the characters in the assignment statement:

position = initial + rate * 60

would be read into the following tokens

# Lexical Analysis.

- Tokens:
  1. The identifier position
  2. The assignment symbol =
  3. The identifier initial
  4. The plus sign +
  5. The identifier rate
  6. The multiplication sign *
  7. The number 60




- Once a token is generated the corresponding entry is made in the symbol table
- White paces are removed
  - The blanks separating the characters of these tokens would normally be eliminated during lexical analysis
  - Carriage returns are removed (.i.e., \r)

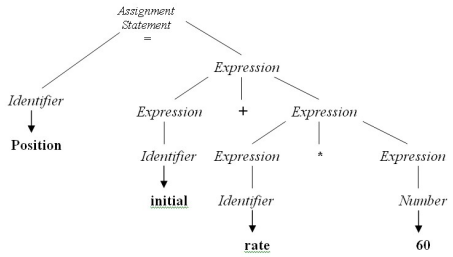
## Syntax Analysis

- It is also called Parsing or Hierarchical Analysis
- Parser converts the tokens produced by lexical analyzer into a tree like representation called parse tree
  - It involves grouping of the tokens of the source program into grammatical phrases using source language grammar
    - The parser checks if the expression made by the tokens is syntactically correct. Similar to performing grammatical analysis on a sentence in a natural language
- The grammatical phrases of the source program are represented by a parse tree/syntax tree
  - Syntax tree is a compressed representation of the parse tree in which the operators appear as interior nodes and the operands of the operator are the children of the node for that operator

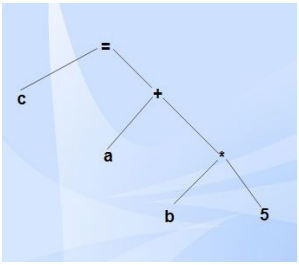
## Syntax Analysis

- The hierarchical structure of a program is expressed by recursive rules
- For example, the rules for the definition of expression are:
  1. Any identifier is an *expression*
  2. Any number is an *expression*
  3. If $expression_1$ and $expression_2$ are expression, then so are
     1. $expression_1$ * $expression_2$
     2. $expression_1$ + $expression_2$
     3. ( $expression_1$ )
- *Thus by rule (1) initial and rate are expressions.*
- *By rule (2) 60 is an expression*
- *By rule (3), we can first infer that rate * 60 is an expression and finally that initial + rate * 60 is an expression*

# Syntax Analysis



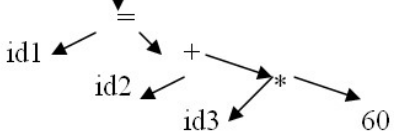

Parse Tree for position = initial + rate * 60




# Example

$id_1 = id_2 + id_3 * 60$

**Syntax Analysis**



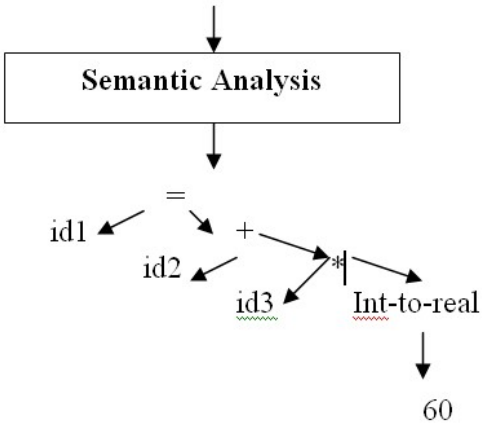
## Semantic Analysis

- The function of the semantic analyzer is to determine the meaning of the source program.
  - Concerned with meanings – checks meaningfulness of the statements in the source program
  - It checks the source program for semantic errors and gathers type information
- It uses the parse tree/syntax tree produced by the syntax analysis phase whether the parse tree constructed follows the rules of language.
  - For example, assignment of values is between compatible data types, and adding string to an integer.
- The semantic analysis performs type checking
  - Here the compiler checks that each operator has operands that are permitted by the source language specification

## Semantic Analysis

- Type information is gathered and stored in the symbol table or in syntax tree for the next phase code generation
- However, many language specification permit some operand coercions.
  - When a binary arithmetic operator is applied to an integer and real. The compiler may need to convert an integer to a real.
- Also identify semantic errors
  - For example, many programming language definitions require a compiler to report an error every time a real number is used to index an array
- The semantic analyzer produces an annotated syntax tree as an output

## Semantic Analysis




## Intermediate Code Generation

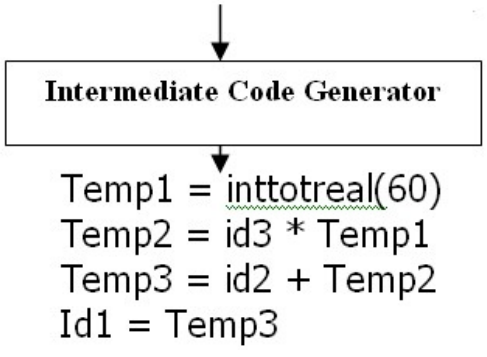
- After semantic analysis, some compilers generates an explicit intermediate representation of the source program
- An intermediate representation is a program for an abstract machine
  - It is in between the high-level language and the machine language.
- An intermediate representation should have two important properties:
  - It should be easy to produce and understand
  - It should be easy to translate into to the target program – machine code

## Intermediate Code Generation

- Intermediate representation can have a variety of forms and one is the “three-address space”.
  - Three-address space is like the assembly language which consists of a sequence of instructions, each of which has at most three operands
  - Each three-address space has at most one operator in addition to the assignment
  - The instructions should be in the order in which the compiler has to decide that in which order operations are to be done
  - The multiplication precedes the addition in the source program.

## Intermediate Code Generation

- The compiler must generate a temporary variable to hold the value computed by each instruction.
- Some “three-address space” instructions have fewer than three operands.



## Code Optimization

- The code optimization phase attempts to improve the intermediate code, so that faster-running machine code will result
  - To produce more efficient object/target program to execute faster.
  - To efficiently use memory
  - To yield better performance
- To remove redundant code without changing the meaning of program
  - Achieved through code transformation while preserving semantics
- There is a great variation in the amount of code optimization different compilers perform.
  - It is optional - Compiler can be either optimizing compiler or dirty compiler

## Code Optimization

- Optimizing compiler
  - The compilers, that do the most called “optimizing compilers”
    - A significant fraction of the time of the compiler is spent on this phase
- Dirty compiler
  - Do not provide code optimization
- Code optimization can slow down the compilation process
  - Some simple optimization can improve running time of which improve the running time of target program without slowing compilation process
  - Therefore code optimization and compilation time are directly proportional to each other
  - Therefore code optimization and running time are inversely proportional to each other

# Code Optimization - Examples

- Constant Folding

```
x := 32          becomes    x := 64
x := x + 32
```

- Unreachable Code

```
goto L2
x := x + 1          ← unneeded
```

- Flow of control optimizations

```
goto L1          becomes    goto L2
 …
L1: goto L2
```

- Algebraic Simplification

```
x := x + 0  ← unneeded
```

# Code Optimization - Examples

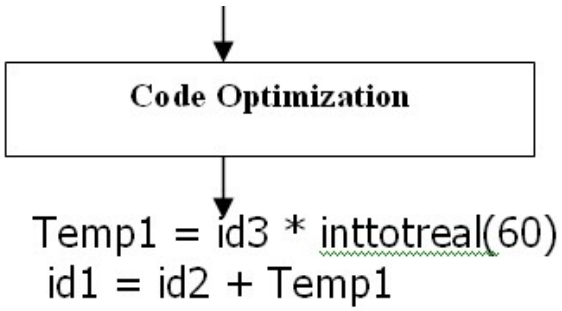
- Dead code

```
x := 32  ← where x value is not changed
           after statement
y := x + y          → y := y + 32
```

- Reduction in strength

```
x := x * 2          → x := x + x
```

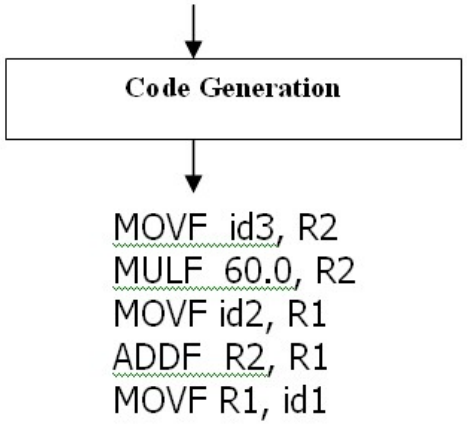
## Code Optimization.




## Code Generation

- The final phase of the compiler is the generation of the target program, consisting of normally relocatable machine code or assembly code.
- Memory locations are selected for each of the variable used by the program.
- Generate the target code form the intermediate code
  - Intermediate instructions are each translated in to the sequence of machine instructions that perform the same task
  - Operations are converted into OP-Codes
  - Registers are loaded with variables Then,.

# Code Generation




# Symbol Table Management

- Serves as a dictionary for the compiler
- A data structure where a compiler records the identifiers used in the source program and collect information about various attributes of each identifier
  - 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 Management

- Identifier can be either variable name or function name
  - In case of variable name, the attributes may provide information about:
    - The type of variable
    - The storage allocated or the storage class
    - Size
    - Its scope (Where in the program it is valid)
  - In case of procedure, the attribute could be
    - The name.
    - The number and types of its argument.
    - The method of passing arguments (by value or by reference)
    - The type returned

# Symbol Table Management

- Lexical analyzer enters the identifiers detected in the source program into symbol table but cannot determine the other relevant attributes of the identifier.
- The other phases enter information about identifiers in to the symbol table and then uses these information in various ways.

## Error Detection and Reporting

- An error is ab abnormal condition in source program which either stop the compilation or generate undesired result
- The basic tasks are
  - Error detection
  - Error handling
  - Error reporting
  - Error recovery
- Each phase of compiler can encounters errors. However 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 helpful.

## Error Detection and Reporting

- Error detection
  - The lexical phase can detect errors where the characters coming in the input do not form any token of the language – violate lexical rules of a language
  - Syntax analysis phase detects an error when the token stream violates the structure rules (syntax) of the language
  - Semantic analysis tries to detect constructs that have the right syntactic structure but no meaning to the operation involved.
    - For example, if we try to add two identifiers, one of which is the name of the array and the other is the name of a procedure.

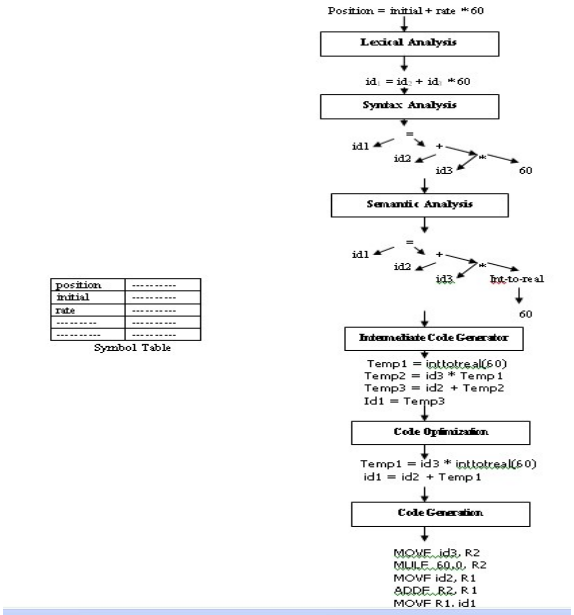
## Error Detection and Reporting

- Error handler
  - Error handler contains a set of routines for handling error encountered in any phase of the compiler
  - Each phase has specific error
    - Every phase of compiler will call an appropriate routine of the error handler as per their specifications
- Error reporting
  - Reporting error to the developer
  - Performed by the error handler

## Error Detection and Reporting

- Error recovery
  - Error recovery is the use of information to automatically correct an error
  - An example --- Internal type casing
  - Difficult and require a lot of knowledge
  - Not implemented in the compilers

# Translation of a statement.




# Types of Errors

- Types of error
  - Lexical error
  - Syntax error
  - Semantic error
  - Logical error
  - Fatal error
  - Spurious error

# Types of Errors

- Lexical Error
  - A sequence of characters does not form any valid token of the language
    - Example
      - An identifier name begin with letter followed by any combination of letters and digits
      - Misspelling of keywords
- Syntax error
  - The stream of tokens violates the grammar rules of a language

# Types of Error

  - Example
    - Statement not ending with semicolon (;)
    - Expression like S = A + * B
    - Unbalanced parenthesis or curly braces
- Semantic Error
  - Statements not meaningful
    - Example
      - Adding character with integer
    - Indexing an array with a floating value
    - Expression like arrayname[1] + function name
    - Identifier multiple declaration within the same scope

# Types of Error

- Logical Error
  - Error in the algorithm of the source program which would produce undesired output
  - Compiler cannot detect logical error
    - Example
      - Infinite loop
      - Array index out of bound
      - Division by zero
- Fatal Error
  - Error occur during run-time of a program
  - Halt the program

# Types of Error

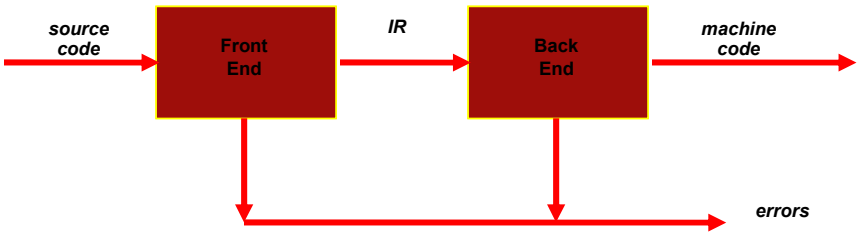
    - Example
      - Accessing invalid memory location
      - Code error not handled by exception handling
        - Opening connection to a database
        - Opening a file from the hard drive
      - Operating system handling interrupt
- Spurious Error
  - Error made by compiler during the error recovery
    - Example
      - A function fi () could be converted into if ()

## Front End and Back End

- The phases are collected into a front end and a back end
  - Similar to the division into analysis and synthesis parts
- The front end contains of those phases that depends primarily on the source language and not on the target machine language
  - Contains Lexical analysis, Syntax analysis, Creation and management of Symbol table, Semantic analysis and the generation of intermediate code
  - Code optimization (if a compiler have) is also part of the front end part
  - Front end also include the error handling that goes along with each of these phases

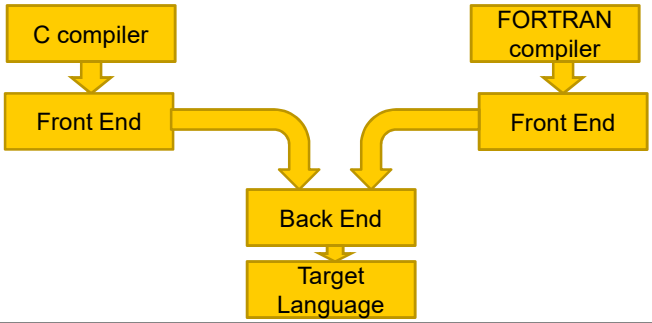
## Front End and Back End

- The back end includes those phases of the compiler that depends on the target machine language
  - Does not depend on the source language just like the intermediate language
  - Code generation is part of the back end



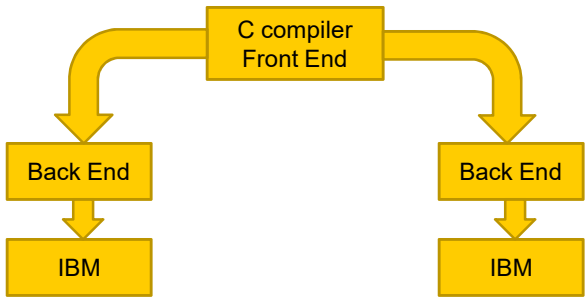
## Advantages of Front End and Back End Separation

- To write a new compiler for the same machine, only the front end of the compiler changes while the back end remains the same
  - For example, a C compiler with front end and back end. For a FORTRAN compiler only front end will be needed and C back end will be used



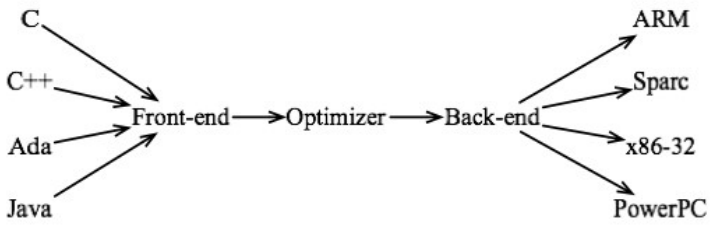

## Advantages of Front End and Back End Separation

- To write a compiler for new machine then the front end of the compiler remain the same and the back end of the compiler will change
  - A C compiler ay have more than one back ends, each for a different machine



## Advantages of Front End and Back End Separation

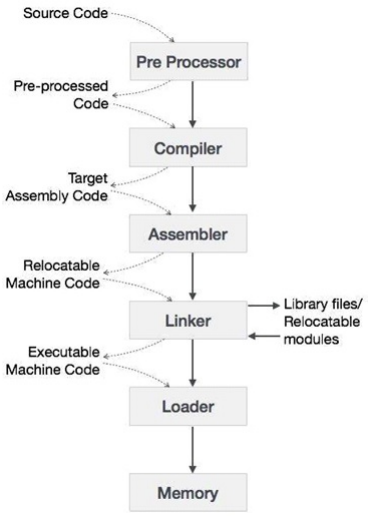
- To write a combined front end for multiple languages and a combined back end to create target program for different machines




## Cousins of Compiler

- Software/programs that are related with compiler
  - Not specific parts of compiler but to carry out some functions/processing as that of compiler
  - To help compiler by performing operations like compiler
- Cousins of compiler are
  - Preprocessor
  - Assembler
  - Linker and loader

# Cousins of Compiler




# Preprocessor

- As the name indicates “pre-process” – to do something before formally starting the compilation
- 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
- The preprocessor is executed before the actual compilation of code begins, therefore the preprocessor digests all these directives before any code is generated by the statements

## Preprocessor

- Preprocessor may perform the following functions
  - Macro processing
  - File inclusion
  - Rational preprocessing
  - Conditional Compilation
  - Language extension

## Macro Processing

- A **macro** is a rule or pattern that specifies how a certain input sequence (often a sequence of characters) should be mapped to an output sequence (also often a sequence of characters) according to a defined procedure
- The mapping process that instantiates (transforms) a macro into a specific output sequence is known as *macro expansion*
- Macro preprocessor deals with
  - Macro definition
  - Macro use

# Macro Processing

- Macro Definition
  - To define preprocessor macros we can use #define construct in C++
  - It consists of name of macro and body forming its definition
  - Its format is: #define identifier replacement
    - Identifier can be any valid name
    - This replacement can be an expression, a statement, a block or simply anything
    - Example
      - #define PI 3.14
      - #define LIGHT_SPEED 299792458 // Speed of light
      - #define circleArea(r) (3.1415*(r)*(r)) //function like macro

# Macro Processing

- Macro Use
  - When the preprocessor encounters this directive, it replaces any occurrence of identifier in the rest of the code by replacement
  - The preprocessor does not understand C++, it simply replaces any occurrence of identifier by replacement

```
#define TABLE_SIZE 100
int table1[TABLE_SIZE];
int table2[TABLE_SIZE];
```

  - After the preprocessor has replaced TABLE_SIZE, the code becomes equivalent to

```
int table1[100];
int table2[100];
```

# Example

```c
#include <stdio.h>
#define PI 3.1415

int main()
{
     float radius, area;
     printf("Enter the radius: ");
     scanf("%d", &radius);
     // Notice, the use of PI
     area = PI*radius*radius;
     printf("Area=%.2f",area);
     return 0;
}
```

```c
#include <stdio.h>
#define PI 3.1415
#define circleArea(r) (PI*r*r)

int main()
{
     int radius;
     float area;

     printf("Enter the radius: ");
     scanf("%d", &radius);
     area = circleArea(radius);
     printf("Area = %.2f", area);

     return 0;
}
```

# 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
- There are two ways to specify a file to be included:
  - #include "file“
  - #include <file>
- The only difference between both expressions is the places (directories) where the compiler is going to look for the file

# File Inclusion

- In the first case where the file name is specified between double-quotes
  - The file is searched first in the same directory that includes the file containing the directive
    - In case that it is not there, the compiler searches the file in the default directories where it is configured to look for the standard header files
- If the file name is enclosed between angle-brackets <>
  - The file is searched directly where the compiler is configured to look for the standard header files
  - Therefore, standard header files are usually included in angle-brackets, while other specific header files are included using quotes

# Relational Preprocessor

- These processors augment older languages with more modern flow of control and data structuring facilities.
  - For example, such a preprocessor might provide the user with built-in macros for constructs like while-statements or if-statements, where none exist in the programming language itself
    - If an old language does not support “if” or “while”, using relational preprocessor we can include it due to macro (i.e. #)

# Conditional Compilation

- Instruct preprocessor whether to include certain chuck of code or not – allows programmer to compile one part of his program leaving the remaining program un-compiled
- It's similar like a if statement. However, there is a big difference you need to understand
  - The if statement is tested during the execution time to check whether a block of code should be executed or not whereas, the conditionals is used to include (or skip) certain chucks of code in your program before execution

# Conditional Compilation

- **Uses of Conditional**
  - Use different code depending on the machine, operating system
  - Compile same source file in two different programs
  - To exclude certain code from the program but to keep it as reference for future purpose
- **How to use conditional?**
  - To use conditional, #ifdef, #if, #defined, #else and #elseif directives are used
  - **#ifdef Directive**

```
#ifdef MACRO
conditional codes
#endif
```

Here, the conditional codes are included in the program only if MACRO is defined.

# Conditional Compilation

- **#if, #elif and #else Directive**

```
#if expression
conditional codes
#endif
```

Here, expression is a expression of integer type (can be integers, characters, arithmetic expression, macros and so on). The conditional codes are included in the program only if the expression is evaluated to a non-zero value.

- The optional #else directive can be used with #if directive

```
#if expression
       conditional codes if expression is non-zero
#else
        conditional if expression is 0
#endif
```

# Language Extension

- These processors attempt to add capabilities to the language by using built-in macros
  - For example, the language equal is a database query language embedded in C
  - Statements begging with ## are taken by the preprocessor to be database access statements unrelated to C and are translated into procedure calls on routines that perform the database access
- The behavior of the compiler with respect to extensions is declared with the #extension directive:
  - #extension extension_name : behavior
  - #extension all : behavior
  - extension_name is the name of an extension

# Assembler

- Typically a modern **assembler** creates object code by translating assembly instruction mnemonics into opcodes, and by resolving symbolic names for memory locations and other entities
- There are two types of assemblers based on how many passes through the source are needed to produce the executable program
  - **One-pass assemblers** go through the source code once and assumes 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, then use the table in a second pass to generate code
-

# Assembler

- The advantage of a one-pass assembler is speed, which is not as important as it once was with advances in computer speed and capabilities
- The advantage of the two-pass assembler is that symbols can be defined anywhere in the program source
  - As a result, the program can be defined in a more logical and meaningful way
  - This makes two-pass assembler programs easier to read and maintain

## Linker

- A **linker** is a program that takes one or more objects generated by a compiler and combines them into a single executable program.
- Three tasks:
  - Searches the program to find library routines used by program, e.g. printf(), math routines.
  - Determines the memory locations that code from each module will occupy and relocates its instructions by adjusting absolute references
  - Resolves references among files Loader

## Loader

- A **loader** is the part of an operating system that is responsible for loading programs, one of the essential stages in the process of starting a program
- Loading a program involves reading the contents of executable file - the file containing the program text - into memory, and then carrying out other required preparatory tasks to prepare the executable for running
- Once loading is complete, the operating system starts the program by passing control to the loaded program code

# Loader

- **Steps for loaders :**
  - Read executable file's header to determine the size of text and data segments
  - Create a new address space for the program
  - Copies instructions and data into address space
  - Copies arguments passed to the program on the stack
  - Jumps to a startup routine that copies the program's arguments from the stack to registers and calls the program's main routine