Compiler design

Department of Computer Science
FACULTY OF TECHNOLOGY
Debre Makos University Burie Campus
Year IV Semester I

by…

Mr. Birku L.
B.Sc. CS, M.Sc. SE
The Academic year of 2015 E.C

Chapter Outline

- **Introduction**
- **What is a compiler?**
- **Compiler Design-Architecture**
    1. Compiler Front-End (Analysis)
    2. Compiler Back-End (Synthesis)
- **Translator**
- **Preprocessor, Assembler, Interpreter ,linker and Loader**
- **Compilation and Execution**
- **Single pass and Multiple pass**
- **The Phases of Compiler Design**
    1. Lexical Analysis
    2. Syntax Analysis
    3. Semantic Analysis
    4. Intermediate Code Generation
    5. Code Optimization
    6. Code Generation

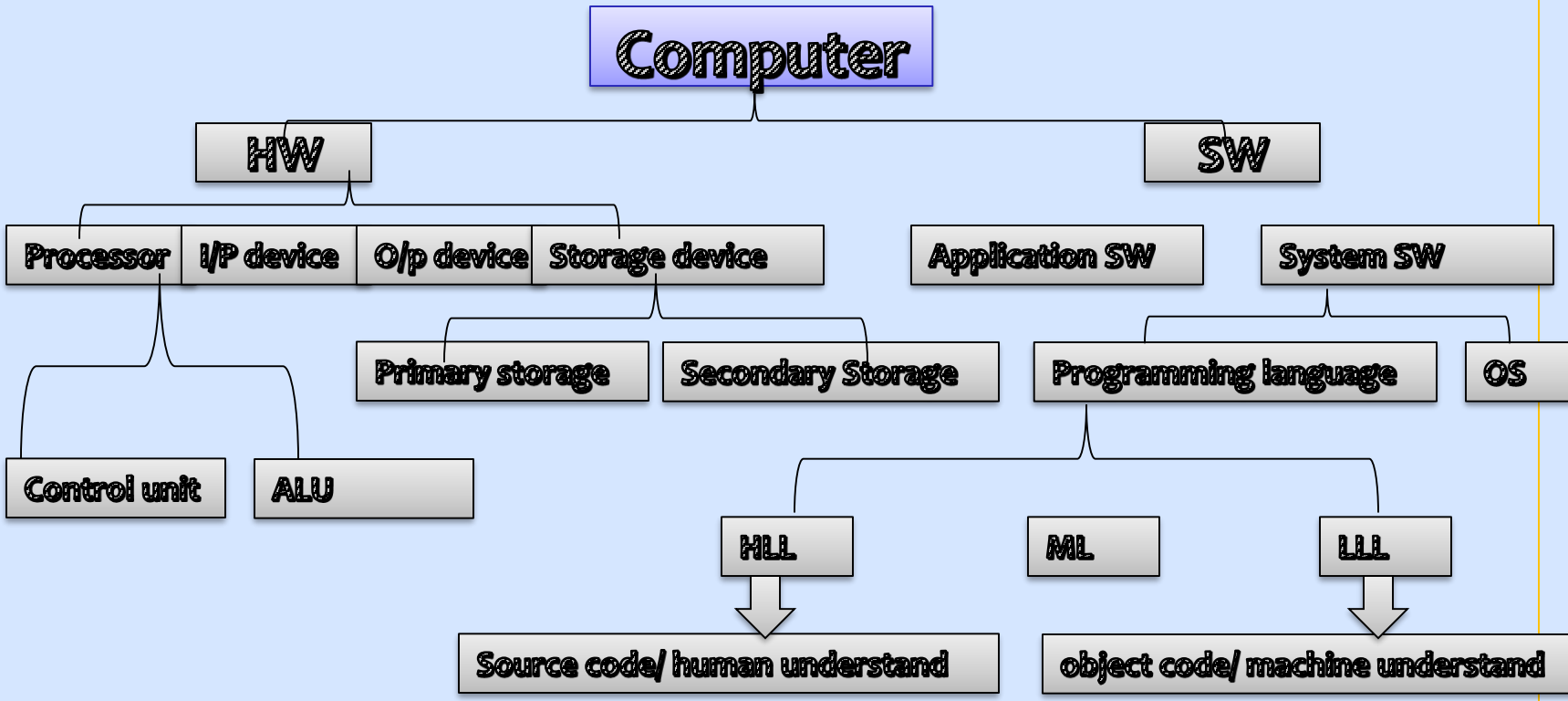
Introduction

What is computer?

- It is an electronic device which is originated from the word "compute" which related to performing arithmetic operations.
- It is also described as electronics device used to take input data, process it and generate meaning full information for end users.
- It is also an acronym word COMPTER stands for Commonly Operated Machine Particularly Used for Technological and Educational Research.

Con't...

Components of computers

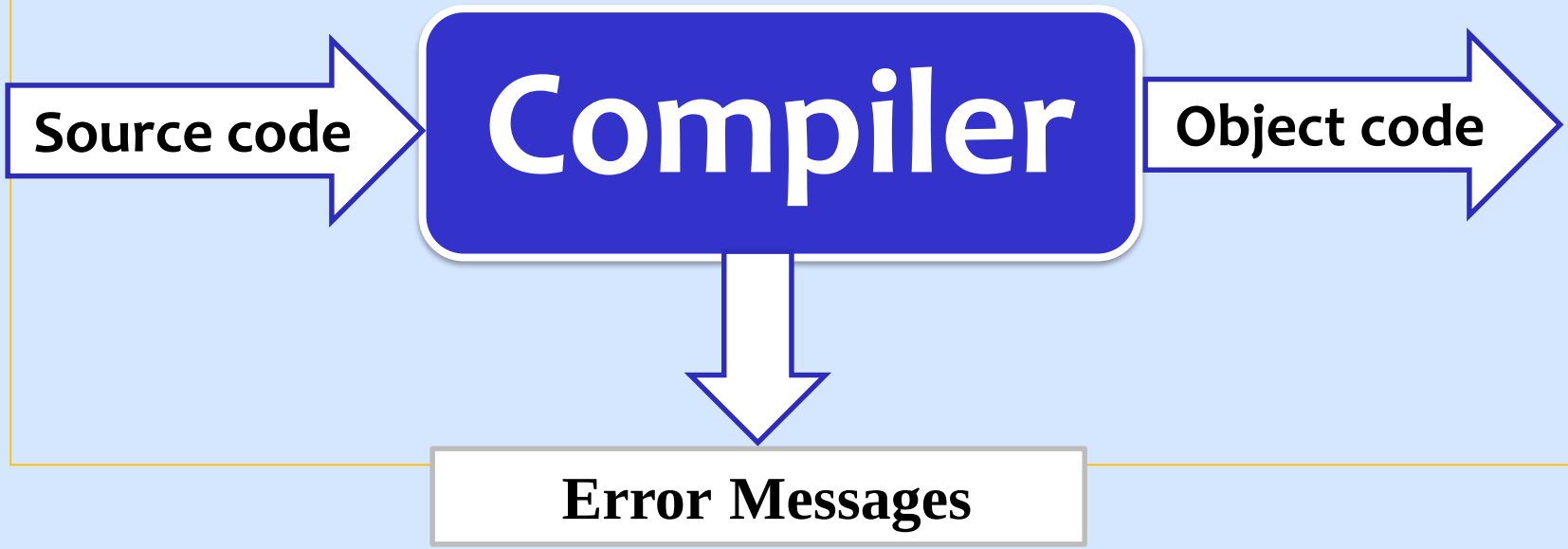


Example:

ABEBE?, 318? Written in which Language??

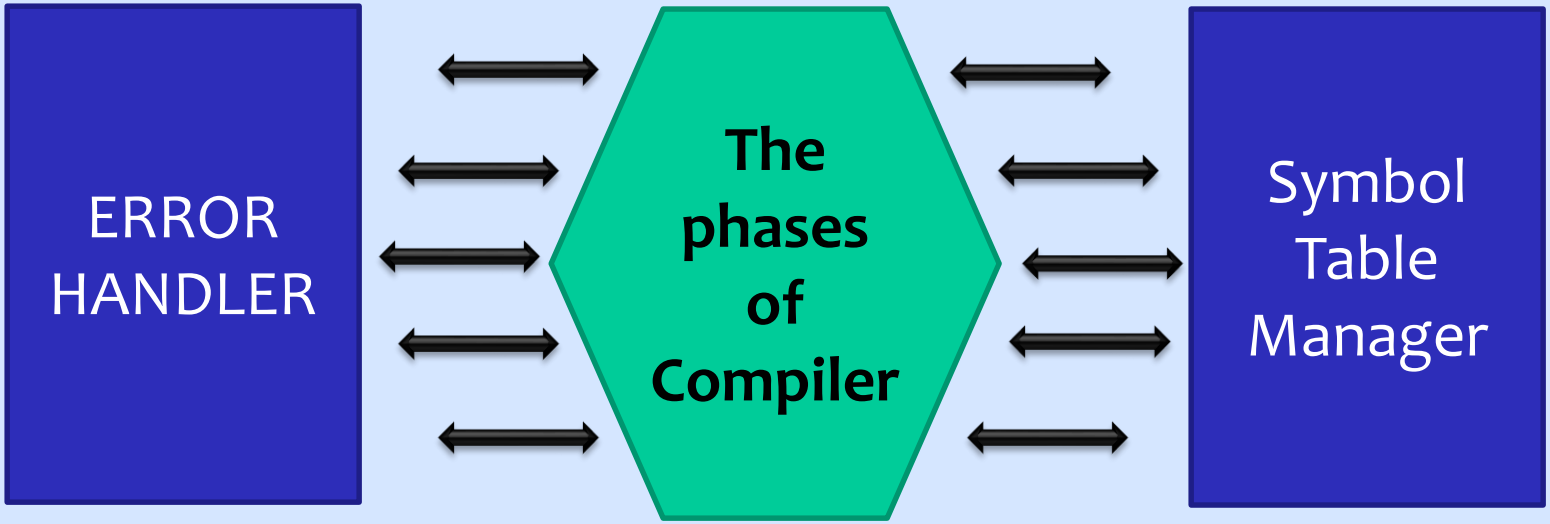
What is a Compiler?

A Compiler:- is a system software which translates (or compiles) a program written in a high-level programming language that is suitable for human programmers into the low-level machine language that is required by computers.



Compiler Design

During this process, the compiler will also attempt to spot and report obvious programmer mistakes by using Error handler, And reserved words are stored in the symbol table on the form of Id table and String Table.



Compiler Design-Architecture

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

1. Compiler Front-End (Analysis) The first three phases
2. Compiler Back-End (Synthesis) The last three phases

1. Compiler Front-End (Analysis)

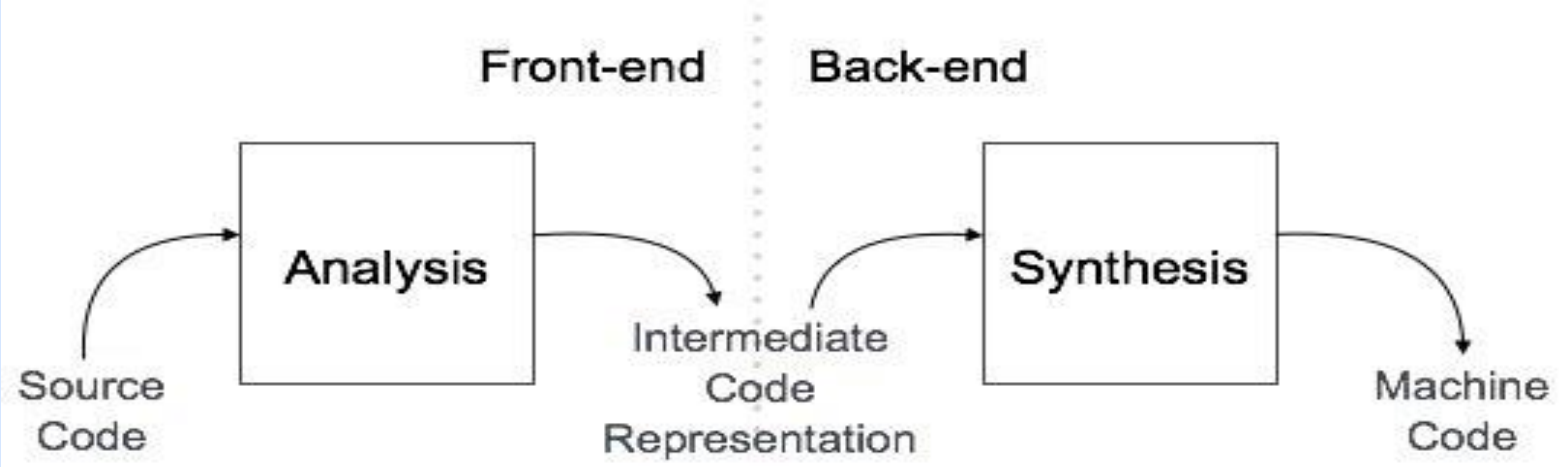
- The Analysis parts break up the source program into constituent pieces and create Intermediate representation of source program.
- Front- End is Machine Independent
- Front-End can be written in a high level language
- Re-use Oriented Programming

2. Compiler Back-End (Synthesis)

- The Synthesis part constructs the desired target code from the intermediate representation.
- Back-End is Machine Dependent
- Lessens Time Required to Generate New Compilers
- Makes developing new programming languages simpler

Compiler Design-Architecture

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



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

Note:- The first three phases are under compiler front-end (Analysis) and the last three phases are under Compiler Back-end (Synthesis).

Translator

- Translator is a mechanism which translates one language to many other language or else we can say a translator is usually translating from high level language to another high level language, or from low level language to Machine language or from HLL to LLL.

- A translator usually has a fixed body of code that is required to translate the program.

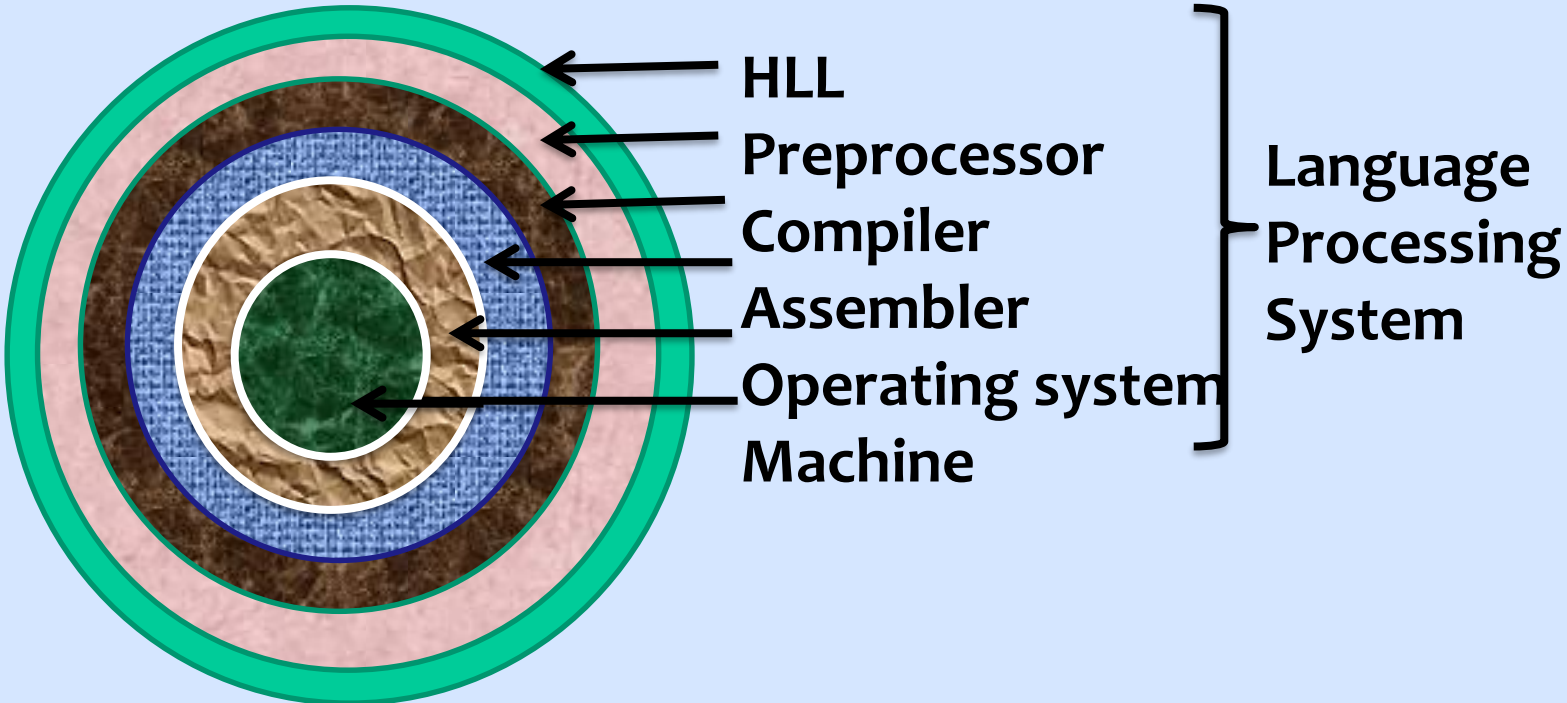
  E.g. From language A **To** ⟶ language B.

  language A and B might be HLL or LLL or M/CL

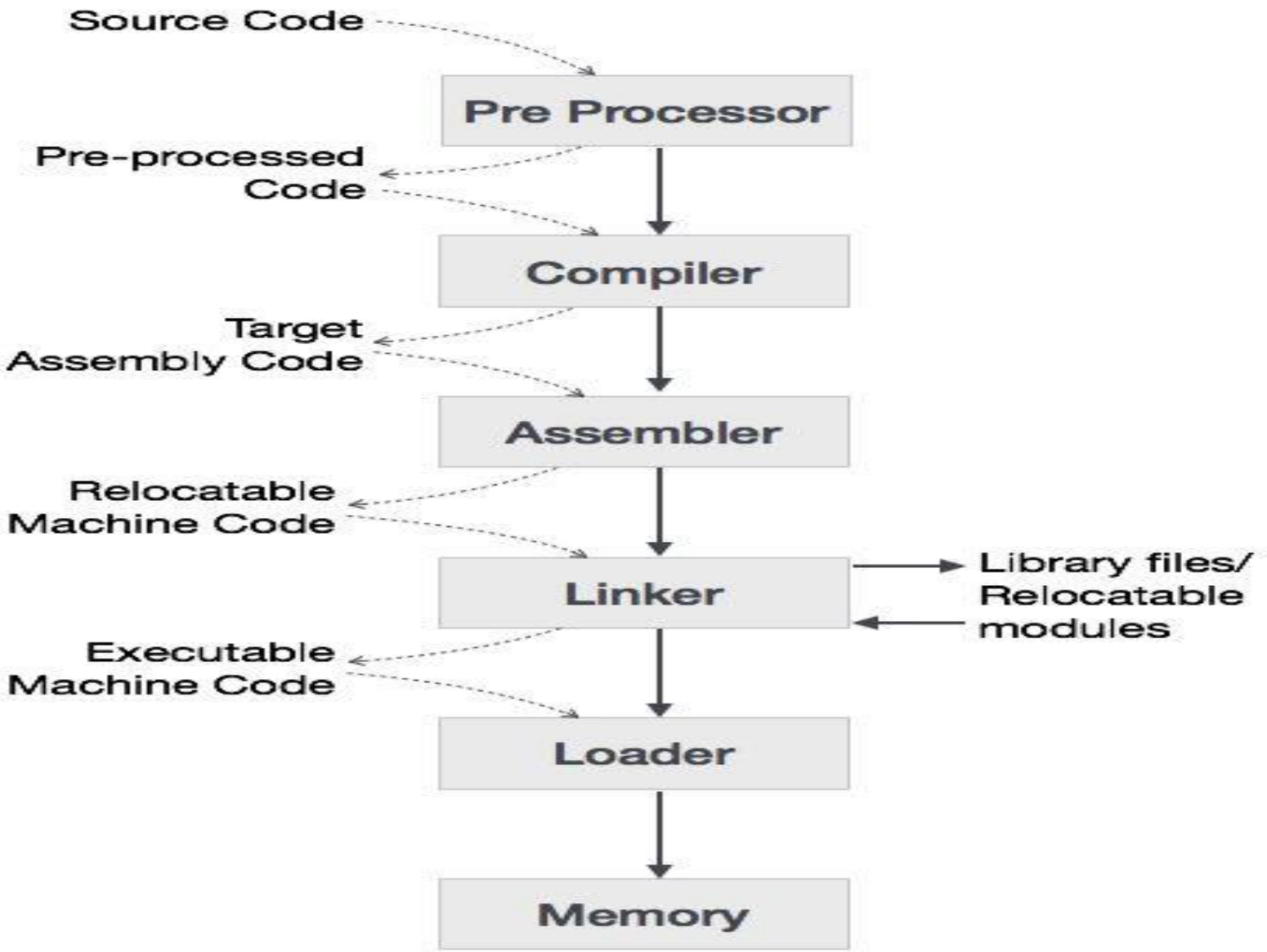
- Common Types of Translator
  - Compiler
  - Interpreter
  - Assembler

Language processing system

- 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 Operating System 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 convert high level language in to pure high level language and deals with macro-processing, augmentation; file inclusion, language extension, etc.
- The typical preprocessing operations include:

(a) file inclusion (Inserting named files). For example, in C,

<#include "header.h">

Remove tag

#include "header.h"

Then is replaced by the contents of the file header.h

(b) Expanding macros (shorthand notations for longer cons.)

for e.g. in C #define avg(x,y) (x+y/2)

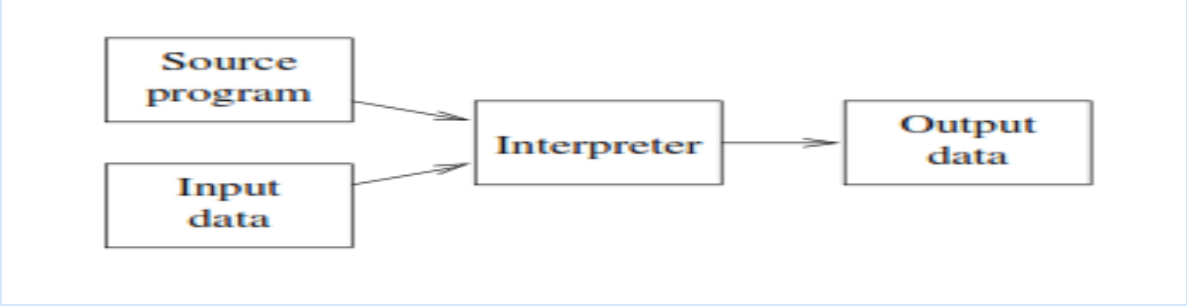
defines a macro avg, that when used in later in the program, is expanded by the preprocessor.

For example, a = avg(a,b) becomes a = (a+b/2) if a=3 and b=5 then a become 3+5/2 =4

Language processing system

Interpreter

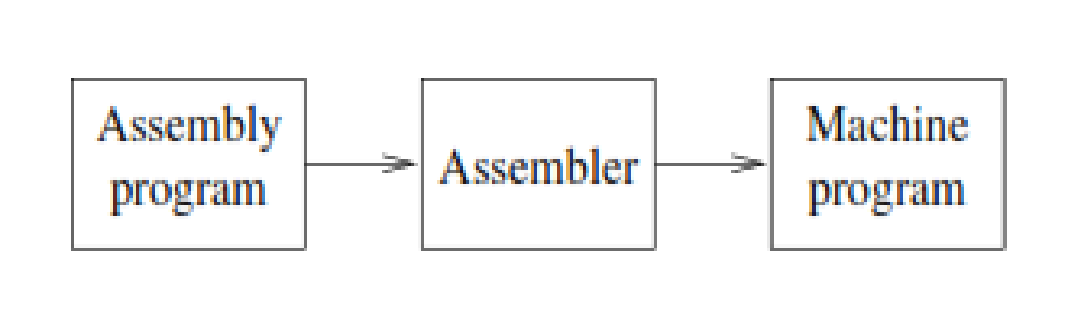
- An interpreter, like a compiler, translates high-level language into low-level machine language. The difference is:-
- A compiler reads the whole source code at once, creates tokens, checks semantics, generates intermediate code, executes the whole program and may involve many passes.
- An interpreter 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.



Language processing system

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.



There are two categories of Machine language

1. Relocatable Machine code : means you can load the machine code at any point in the computer then you can run.
2. Executable Machine code : also known as Absolute machine code which is executed and stored in a particular memory.

Language processing system

Linker

- Linker is a computer program that links and merges various object files together in order to make an executable file.
- 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 executes them. It calculates the size of a program (instructions and data) and creates memory space for it. It initializes various registers to initiate execution.

Compilation and Execution

- **Compilation** is a process of compiler that translation from one language, the input or source language, to another language, the output or target language.
- 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.
- **Execution** is a process which a computer or a virtual machine performs the instructions of a computer program to do a sequences of simple actions on the executing machine.
- Those actions produce effects according to the semantics of the instructions in the program.
- Programs for a computer may execute in a batch process without human interaction
- The term run is used almost synonymously. A related meaning of both "to run" and "to execute"

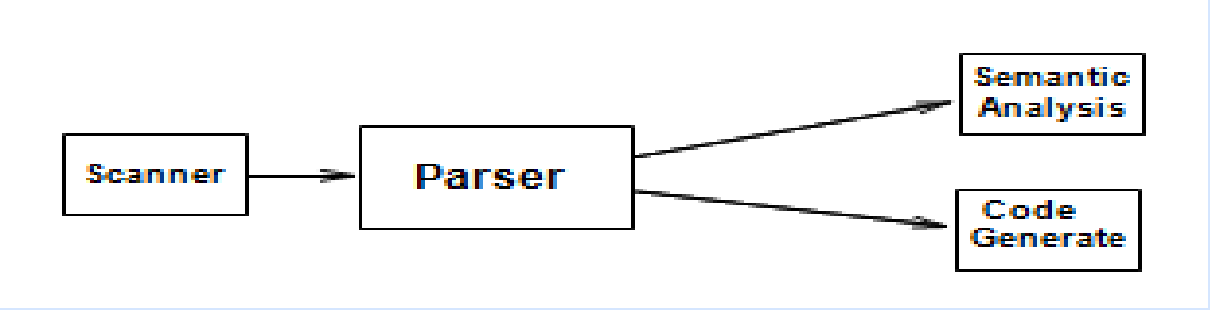
Single-pass and Multi-pass

A compiler can have many phases and passes.

- **Pass:** A pass refers to the traversal of a compiler through the entire program. A pass can have more than one phase.
- **Phase:** A phase of a compiler is a distinguishable stage of compiler.
- A Compiler pass can broadly be categorized into two based on the way they design.
    1. Single pass (one pass) compiler
    2. Multiple pass (wide pass) compiler

Single pass (one-pass) compiler

A Single-pass (one-pass) compiler is a compiler that passes through the parts of each compilation unit only once, immediately translati



Single-pass and Multi-pass

Advantages of Single pass

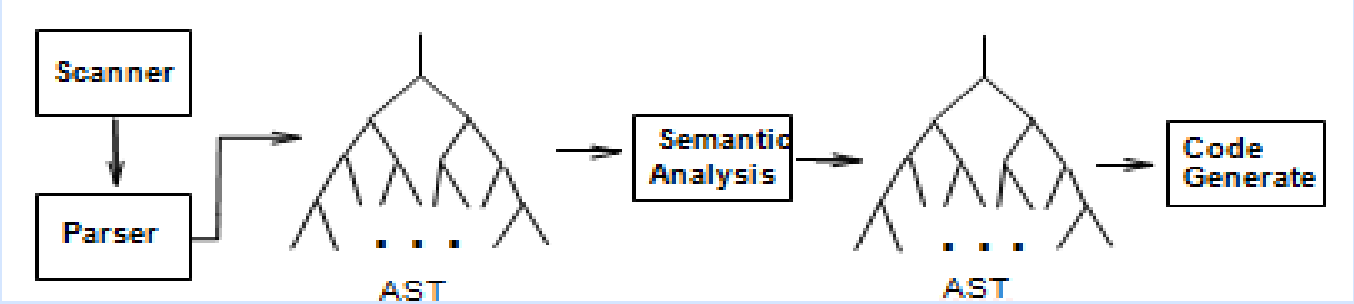
- A single pass (One-pass) compiler is smaller and faster than multi-pass compilers.

Disadvantage of Single pass

- A single pass (one pass) compiler is not possible to perform many of the sophisticated optimizations needed to generate high quality code.

Multi-pass (wide compilers) compiler

- A multi-pass compiler which converts the program into one or more intermediate representations in steps between source code and machine code, and which reprocesses the entire compilation unit in each sequential pass.



Single-pass and Multi-pass

Advantage of Multi-pass

- **Machine Independent:** Since the multiple passes include a modular structure, and the code generation decoupled from the other steps of the compiler, the passes can be reused for different hardware/machines.
- **More Expressive Languages:** Many programming languages cannot be represented with single pass compilers, so multi-pass compiler is more suitable

Disadvantage of Multi-pass

- Multi-pass compiler is a slower than one pass compiler,
- Major Drawback-Speed

Single-pass and Multi-pass

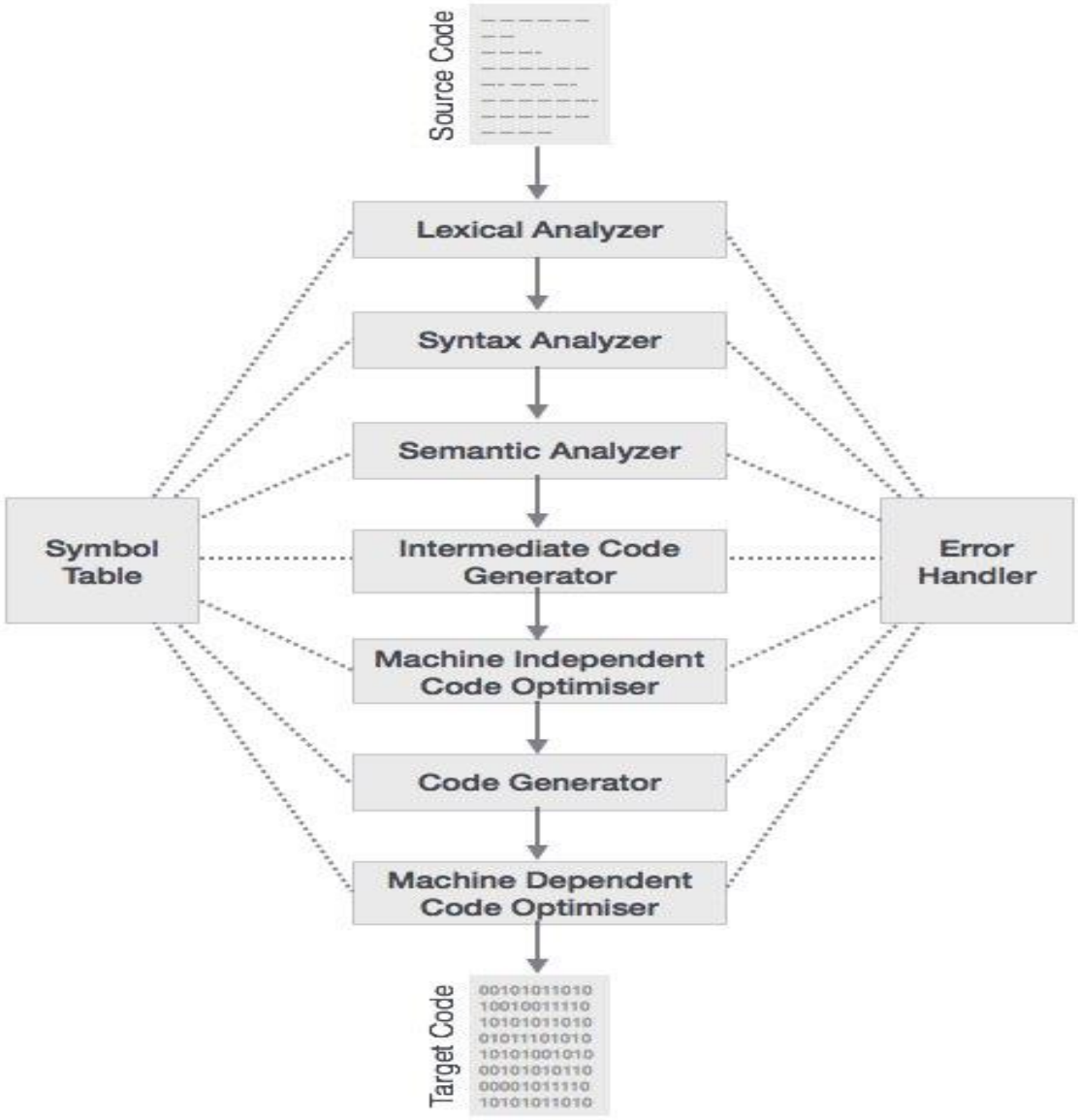
- **Single pass**
  - Creates a table of Jump Instructions
  - Forward Jump Locations are generated incompletely
  - Jump Addresses entered into a fix-up table along with the label they are jumping to
  - As label destinations encountered, it is entered into the table of labels
  - After all inputs are read, CG revisits all of these problematic jump instructions

- **Multiple pass**
  - No Fix-Up table
  - In the first pass through the inputs, CG does nothing but generate table of labels.
  - Since all labels are now defined, whenever a jump is encountered, all labels already have pre-defined memory location.
  - Possible problem: In first pass, CG needs to know how many Multi-Layer Insulation (MLI) correspond to a label.
  - Major Drawback-Speed

The phases of Compiler Design

Phase:

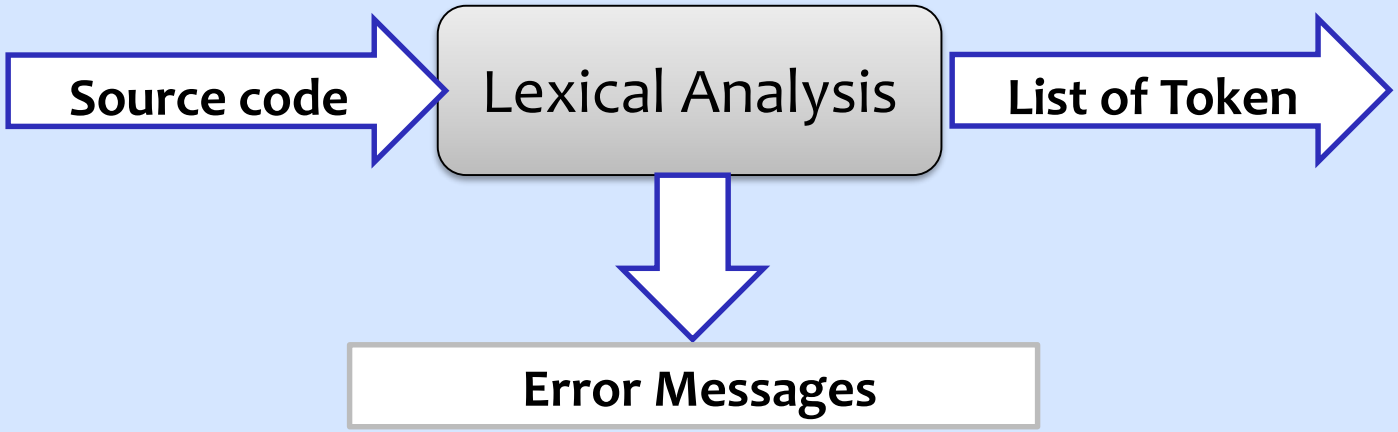
- A phase of a compiler is a distinguishable stage,
- which takes input from the previous stage, processes and yields output that can be used as input for the next stage.



The phases of Compiler Design

Lexical (Linear) analysis or scanning

- This is the initial part of reading and analyzing the program text: The text is read and divided into **tokens,**
- so the source code translated in to a list (stream) of token.by removing white space and comments



- Token is a smallest unit of a program, and symbolic names for the entities
- e.g. if for the keyword , and id for any identifier

The phases of Compiler Design

Lexical (Linear) analysis or scanning

- A pattern is a sequence of characters from the input constitutes a token; e.g. the sequence i, f for the token if ,
- A lexeme is a sequence of characters from the input that match a pattern
- E.g. if (age ==10)

Lexeme	Token	Pattern
if	Keyword(if)	i,f
(	special symbol	a left parenthesis
age	identifier(sum)	letter followed by seq. of alphanumeric l (l+d)*
==	assignment opr	an assignment operator
)	special symbol	a right parenthesis
10	number (10)	a seq. of number

Cont….

How many tokens are there?

- Int a=5;
- AVG=(num1+num2)/2;

The phases of Compiler Design

- Lexical analyzer represents these lexemes in the form of tokens as: **<token-name, attribute-value>**
- Token-name: an abstract symbol is used during syntax analysis,
- Attribute-value: points to an entry in the symbol table for this token.

Example: position =initial + rate * 60

1."position" is a lexeme mapped into a token (id, 1),
id is a token-name for identifier and
1 is an Attribute-value for position.

2. = is a lexeme that is mapped into the token (=).
Since this token needs no attribute-value, the lexeme itself is used as the name of the abstract symbol.

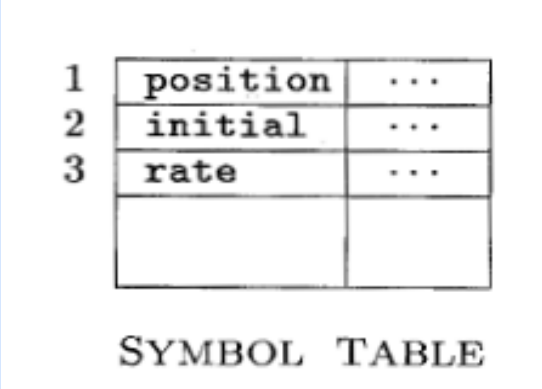
3. "initial" is a lexeme that is mapped into the token (id, 2),
2 points to the symbol-table entry for initial.

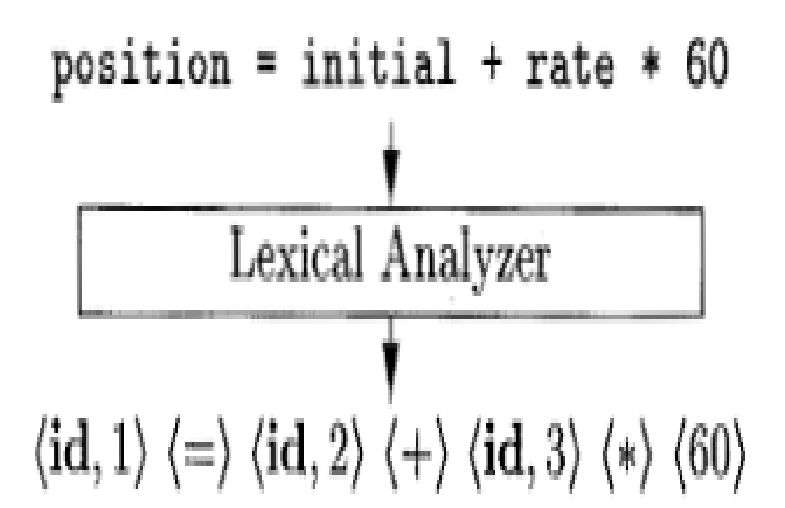
The phases of Compiler Design

4. + is a lexeme that is mapped into the token (+).
5. "rate" is a lexeme mapped into the token (id, 3), where 3 points to the symbol-table entry for rate.
6. * is a lexeme that is mapped into the token (*) .
7. 60 is a lexeme that is mapped into the token (60)

❖ So

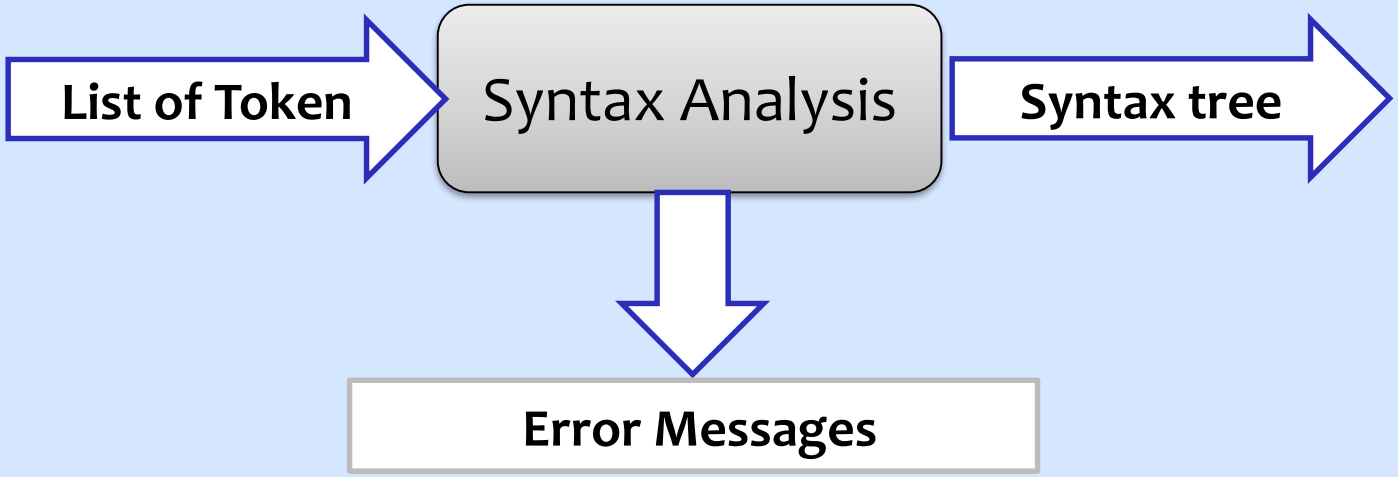
- position, initial, rate are identifier
- = is an assignment operator
- +,* are Arithmetic operator





The phases of Compiler Design

Syntax (Hierarchical) analysis or parsing This phase takes the list of tokens produced by the lexical analysis and arranges these in a tree-structure (called the syntax tree) that reflects the structure of the program. Also known as the heart of compiler.

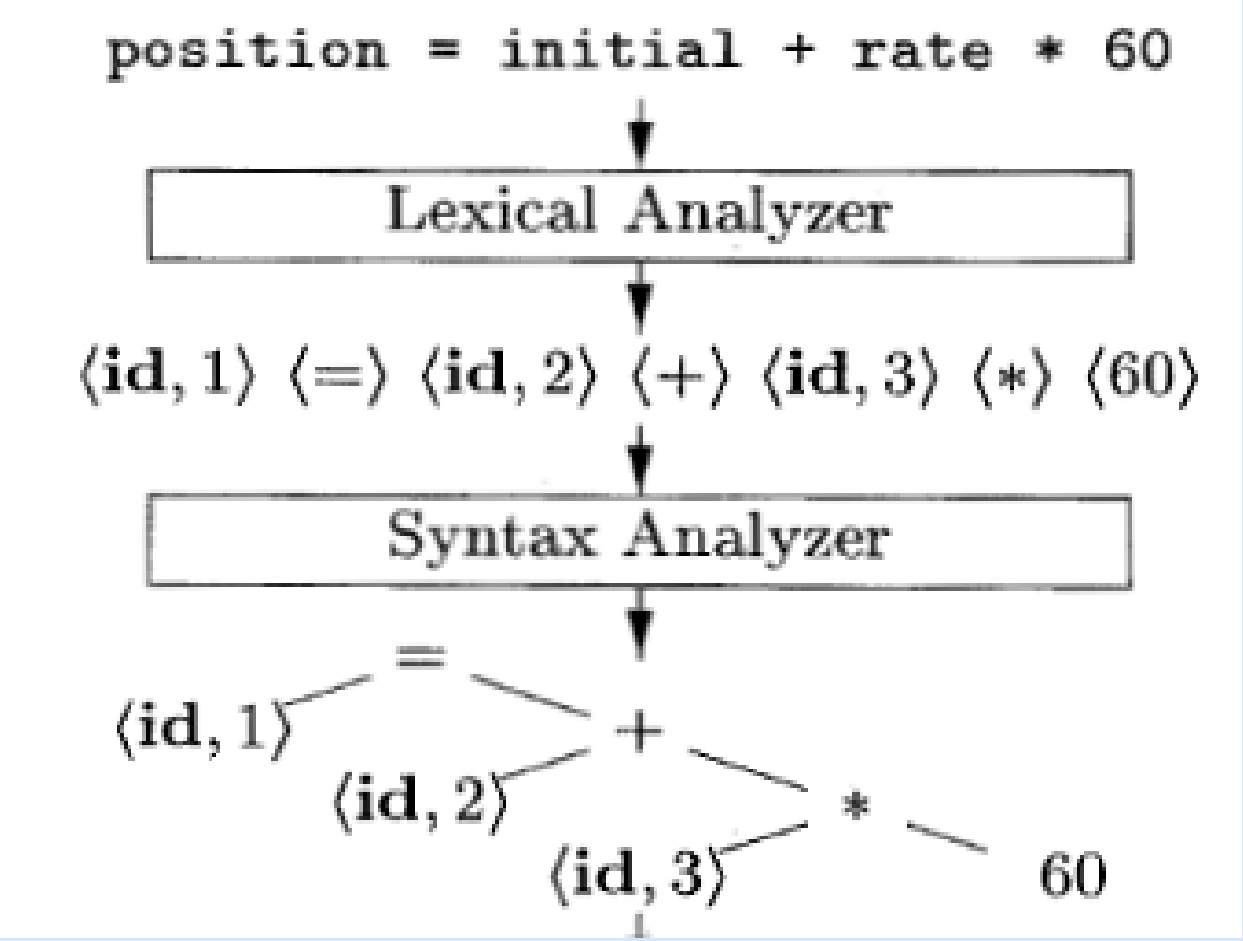


There are two algorithms to traverse a syntax tree

1. Top-down algorithm
2. Bottom-up algorithm

The phases of Compiler Design

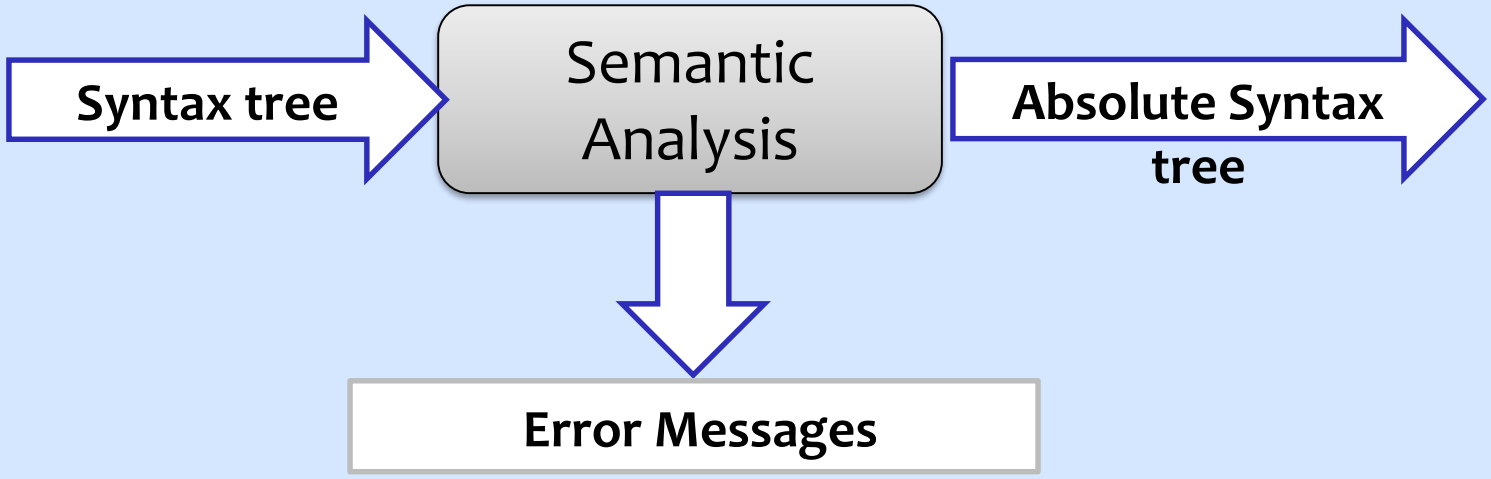
Example: position =initial + rate * 60



The phases of Compiler Design

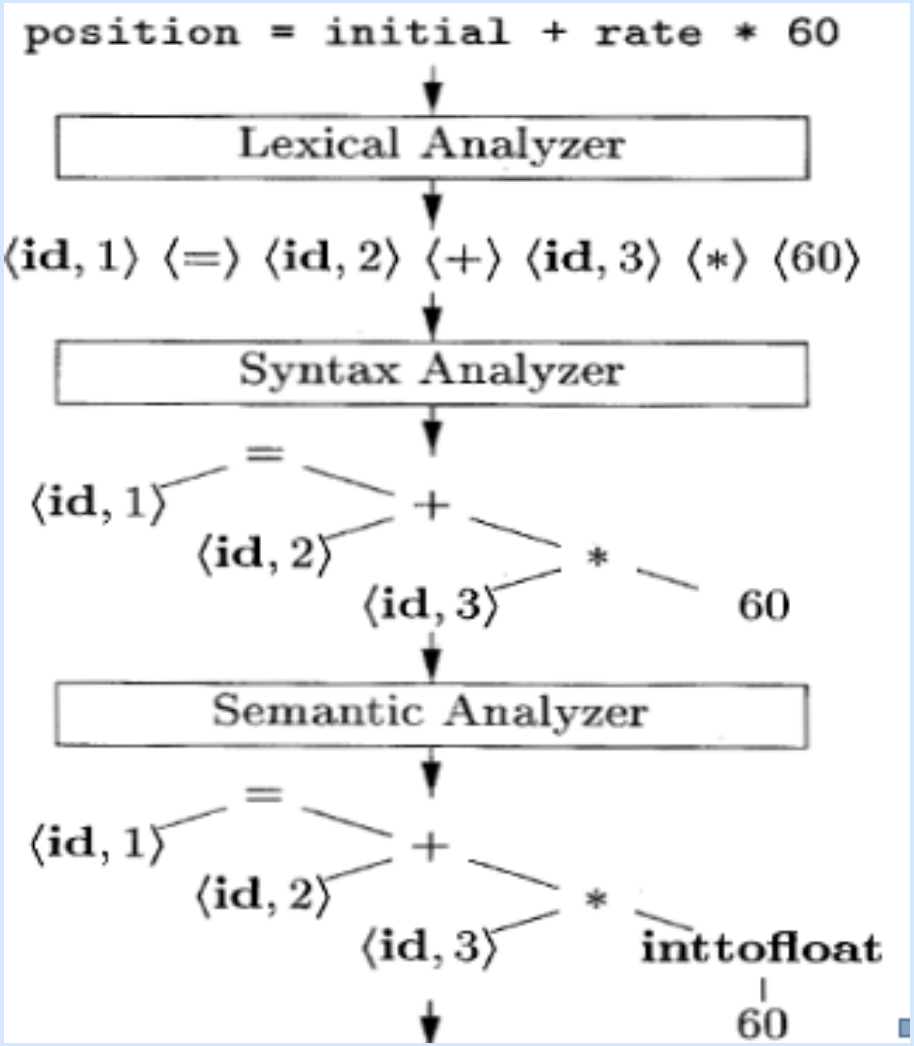
Semantic analysis This phase analyses the syntax tree to determine if the program violates certain consistency requirements,

- in practice semantic analysis are mainly concerned with type and error checking



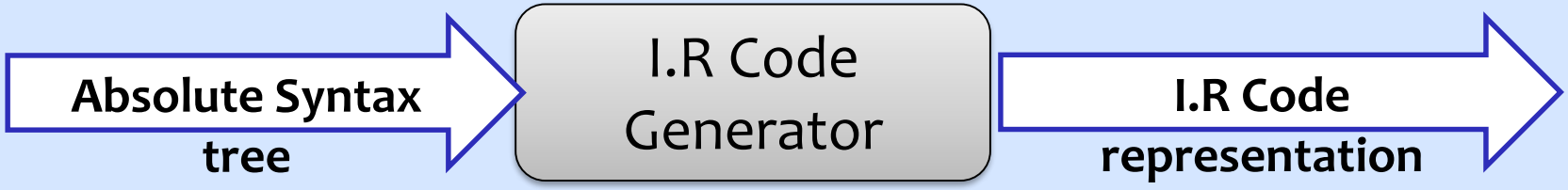
The phases of Compiler Design

Example: position =initial + rate * 60



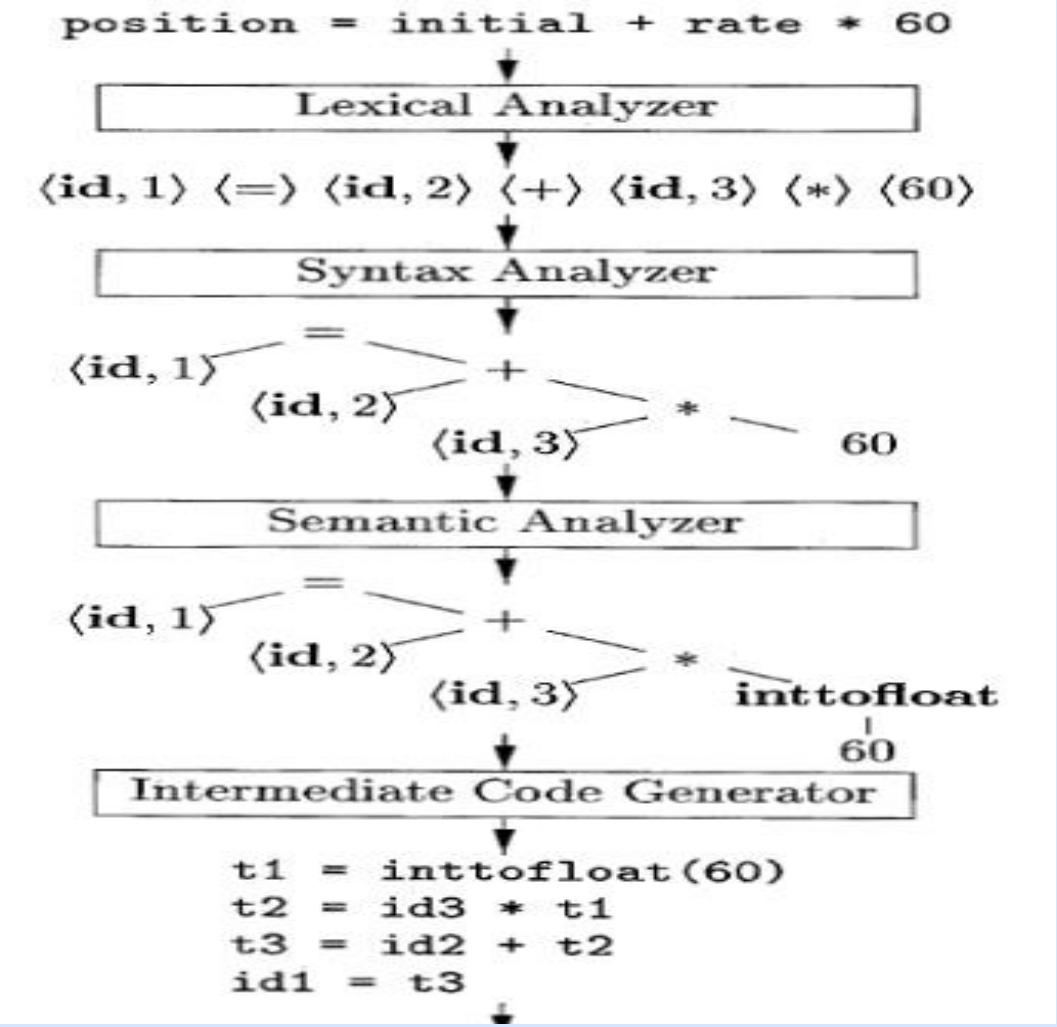
The phases of Compiler Design

- **Intermediate code generation** After semantic analysis the compiler generates an intermediate code of the source code for the target machine by using three address code.
- It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.



The phases of Compiler Design

Example: position =initial + rate * 60

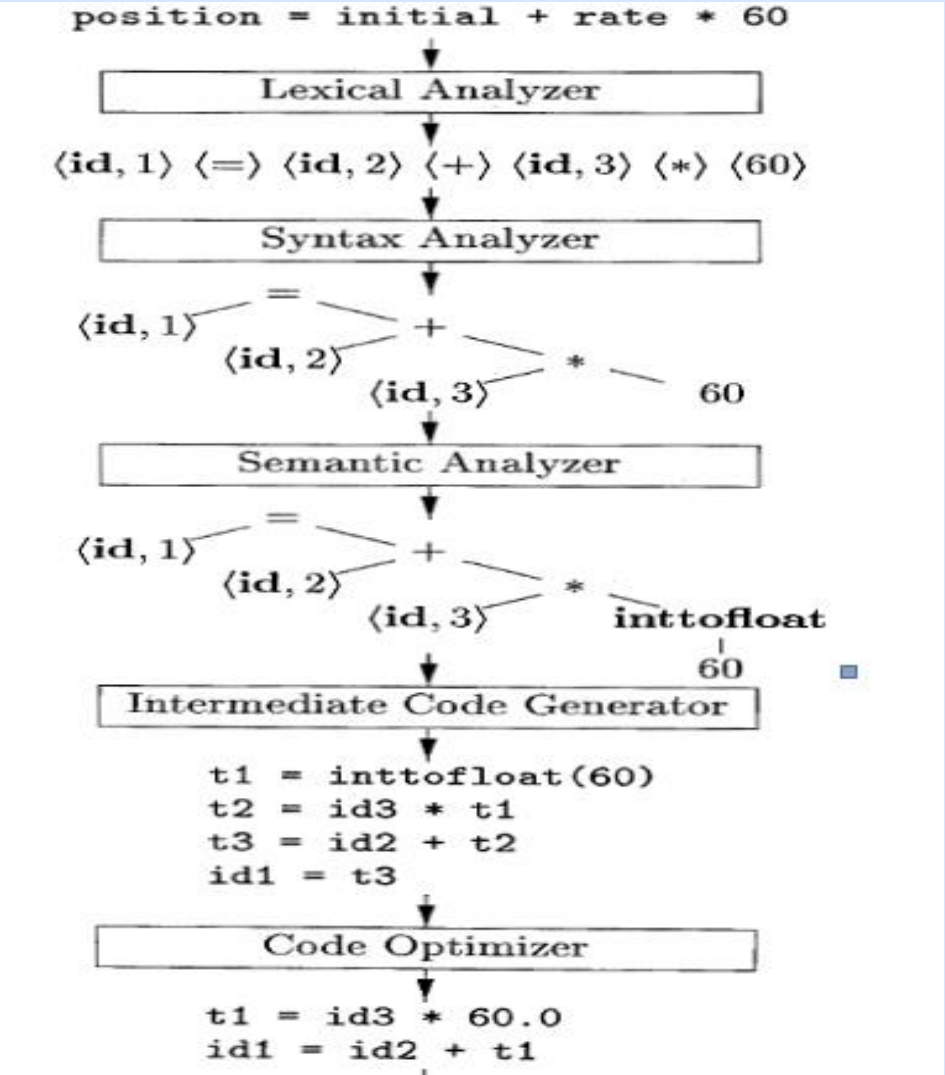


The phases of Compiler Design

- **Code optimization** The optimization phase, Assign specific CPU registers for specific values.
- Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).
- Main objective:
  - Maximize the utilization of the CPU registers
  - Minimize references to memory locations

The phases of Compiler Design

Example: position =initial + rate * 60



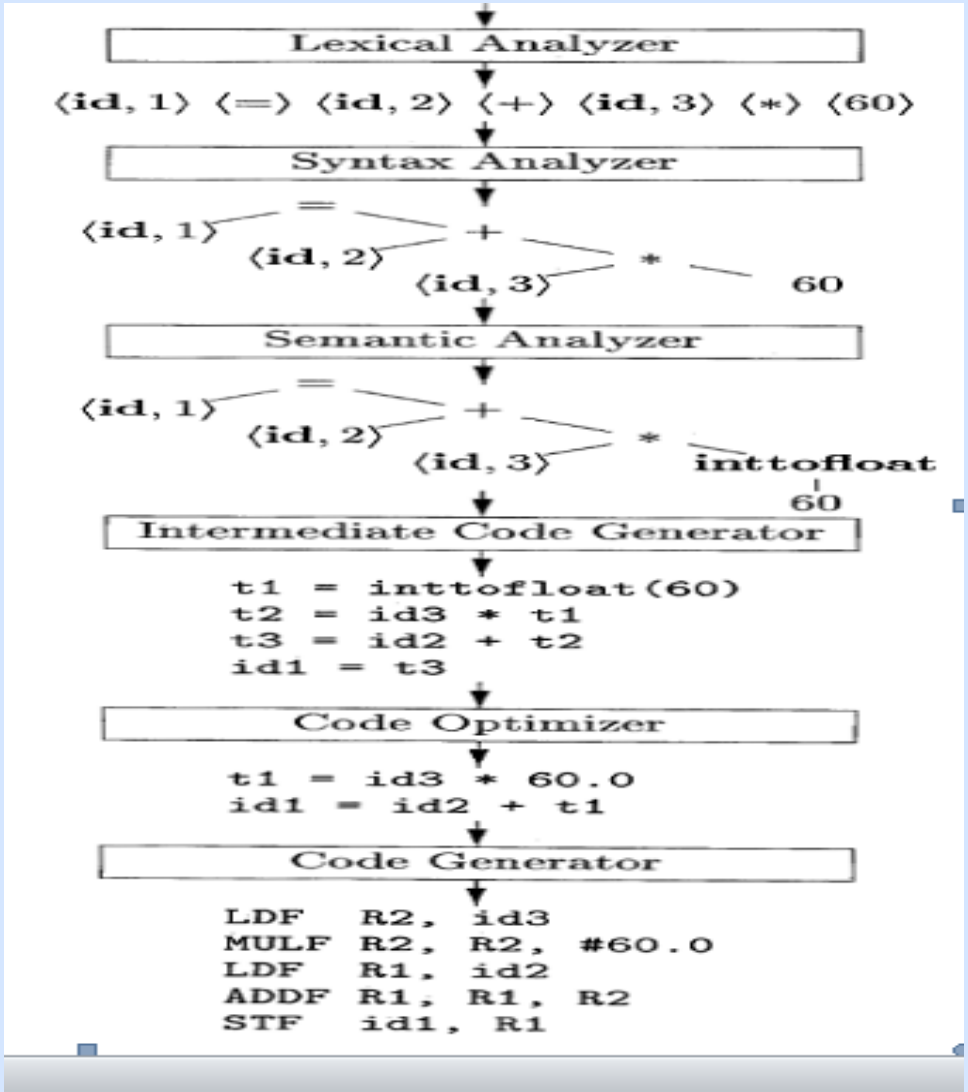
The phases of Compiler Design

- **Code generation** The intermediate language is translated to assembly language (a textual representation of machine code) for a specific machine architecture.

- The code generator translates the intermediate code into a sequence of (generally) re-locatable machine code.

The phases of Compiler Design

Example: position =initial + rate * 60



Symbol table

Symbol Table

- A symbol table is a data structure containing all the identifiers (i.e. names of variables, procedures etc.) of a source program together with all the attributes of each identifier.
- For variables, typical attributes include:
  - its type,
  - how much memory it occupies,
  - its scope.

For procedures and functions, typical attributes include:

- the number and type of each argument (if any),
- the method of passing each argument, and
- the type of value returned (if any).

Error Handling

Error Handling

Each of the six phases (but mainly the analysis phases) of a compiler can encounter errors. On detecting an error the compiler must:

- report the error in a helpful way,
- correct the error if possible, and
- continue processing (if possible) after the error to look for further errors.

Types of Error Errors are either syntactic or semantic:

Syntax errors are errors in the program text; they may be either lexical or grammatical:

(a) A lexical error is a mistake in a lexeme,

- for examples, typing far instead of for,

(b) A grammatical error is a one that violates the (grammatical) rules of the language,

- for example `if x = 7 y = 4` (missing open and close parenthesis).

Error Handling

Error Handling

(c) **Semantic errors** are mistakes concerning the meaning of a program construct; they may be either type errors, logical errors or run-time errors:

- **Type errors** occur when an operator is applied to an argument of the wrong type, or to the wrong number of arguments.
- **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.
- **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?

Questions