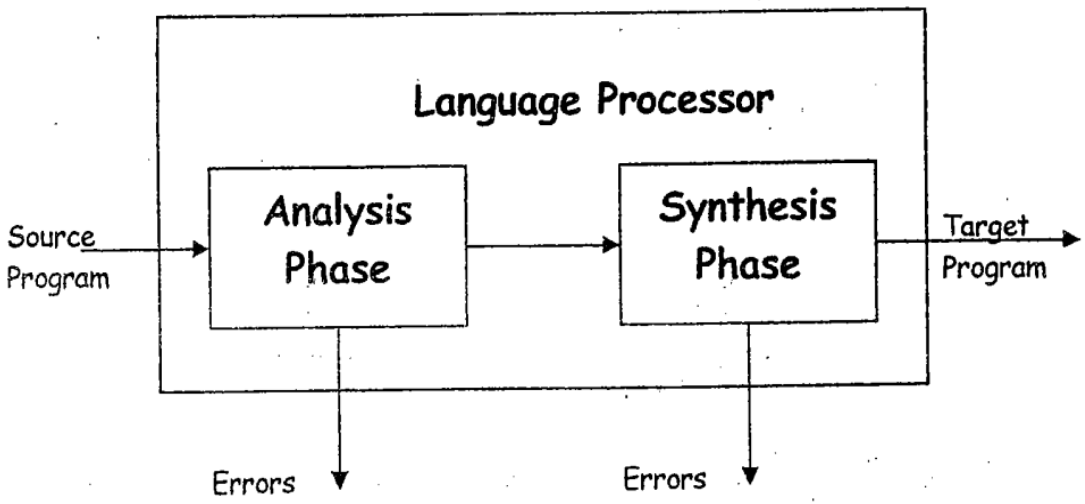
CHAPTER ONE
Introduction

- Programming languages are notations for describing computations to people and to machines.
- A program must be **translated** into a form in which it can be executed by a computer.
- One of the language processor that translate a high level language in to low level languages are compiler
- Language processor/Translator is a program , which converts a program written in any Source Language in to any other Destination language.
- A compiler translates (or compiles) a program written in a high-level programming language that is suitable for human programmers into the low-level machine language that is required by computers.
- During this process, the compiler will also **attempt to detect** and report obvious programmer mistakes.

- ***Language Processor = Analysis of Source Program + Synthesis of Target Program.***

➢ ***Some of the activates of LP:***

- ➢ Bridging the gap between application domain and execution domain
- ➢ Translating one level of language to another
- ➢ detect error in source during analysis and synthesis.
- ➢ Program generation and execution



- A compiler is a program that can read a program written in high level language, called the source language, and translate it into an equivalent program in another language – called the target language (see Figure 1.1).

- The target program is then provided the input to produce output.

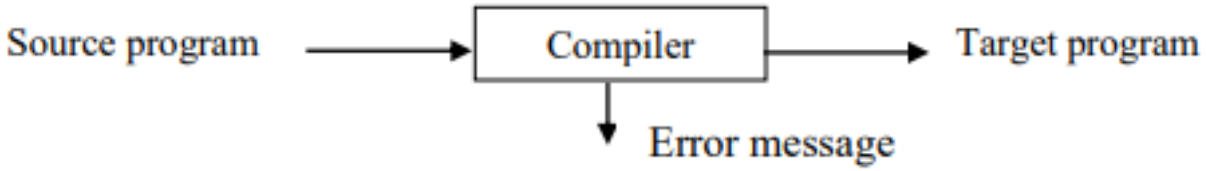


Figure 1.1: Compiler

- A compiler also reports any errors in the source program that it detects during the translation process.

- A compiler acts as a translator, transforming human-oriented programming languages into computer oriented machine languages.

- It is a program that translates an executable program in HLL language into an executable program in ML language.

- The design of compiler focus on the following activities:
  - ✓ **Scanning:-**tokenizing-
    - ✓ **Tokenization** is the process of **breaking up a sequence of strings into pieces** such as words, keywords, phrases, symbols and other elements called tokens.
  - ✓ **Parsing:-**validating the syntax and semantic validity of the given instructions.
  - ✓ **Creating the symbol table**:- for files and strings;
  - ✓ **Resolving the forward references:** symbol/variable is referenced before it is declared.
  - ✓ **object code conversion:-**Converting into the machine language

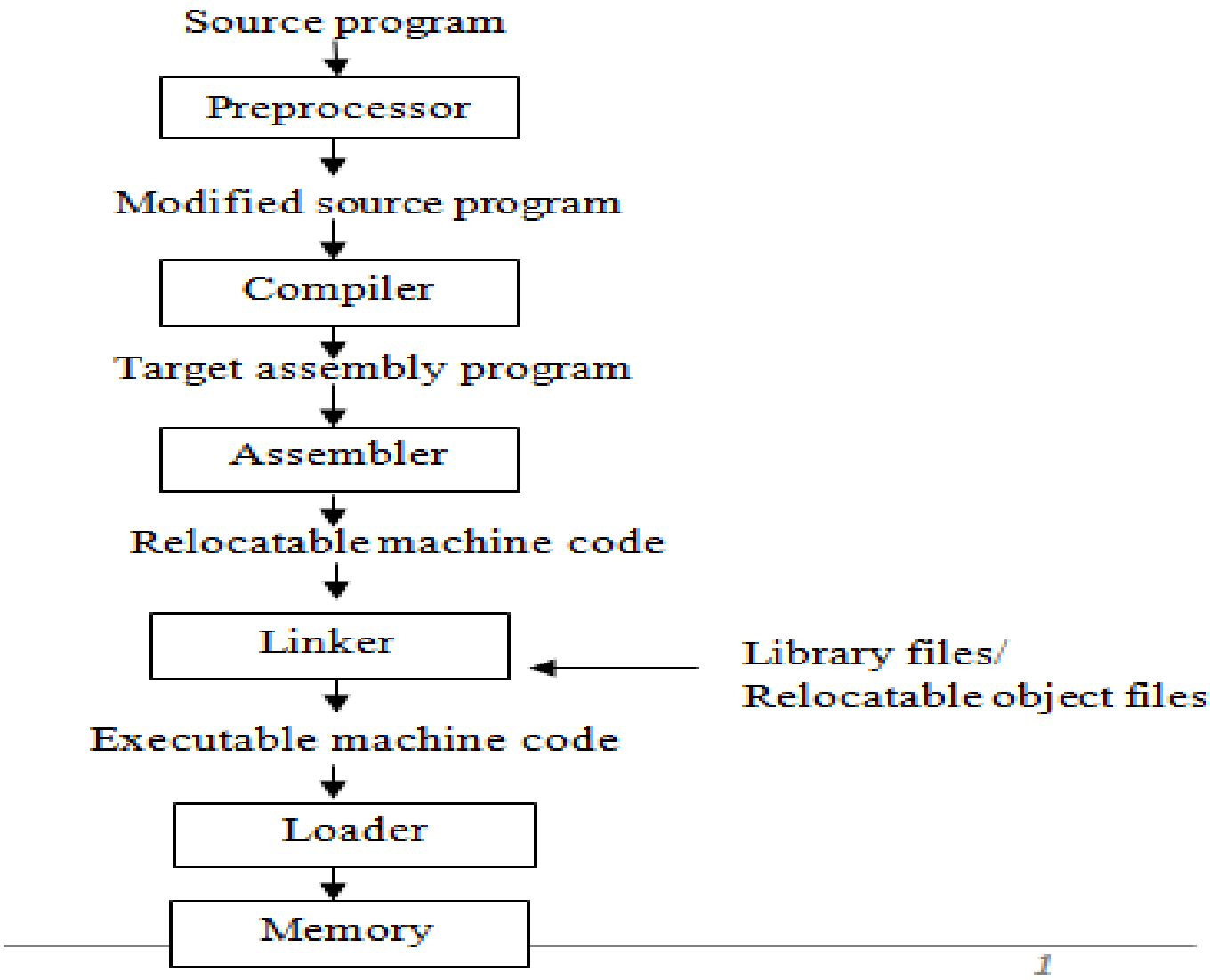


Figure 1.2: A language processing system

Preprocessor:

- ✓ A preprocessor is a tool that produces input for compilers.
- ✓ It deals with macro-processing, augmentation, file inclusion, language extension, etc.
- ✓ The modified source program is fed to a compiler.

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 step by step and converts it to an intermediate code, executes it.
- ✓ If an error occurs, an interpreter stops execution and reports it; whereas a compiler reads the whole program even if it encounters several errors.
- ✓ The machine-language target produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs. Why?
- ✓ An interpreter can usually give better error diagnostics than a compiler. why?
- ✓ **because interpreter executes the source program statement by statement.**

❖**Assembler:** translates assembly language programs into machine code. The output of an assembler is called an object file.

✓The assembly language program is then processed by a program called assembler that produces a re-locatable machine code as its output.

❖**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.

✓Large programs are often compiled in pieces, so that the re-locatable machine code may have to be linked with other re-locatable object files runs on the machine.

❖**Loader:** is responsible for loading executable files into **memory** and execute them.

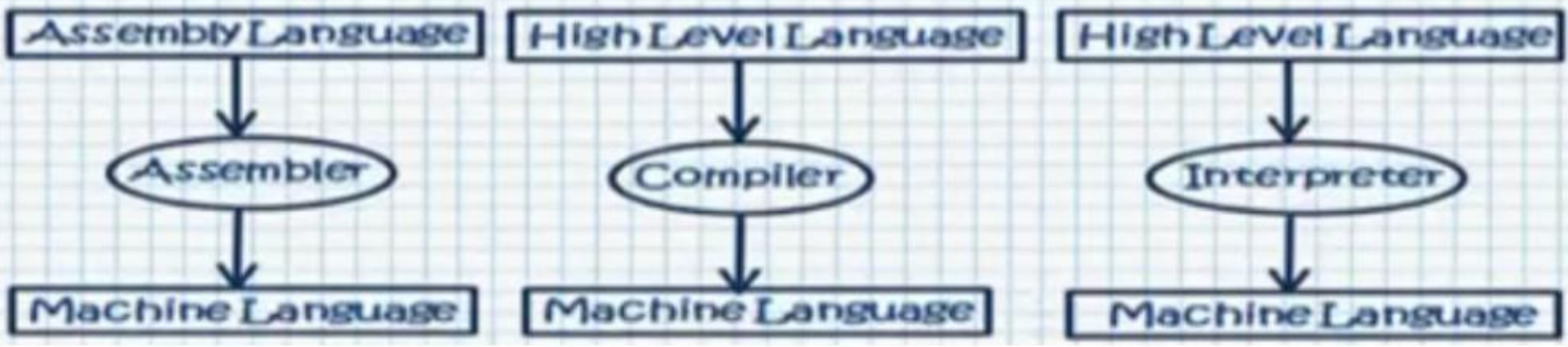
✓It calculates the size of a program (instructions and data) and creates memory space for it.

The phases of language processor/ Compiler

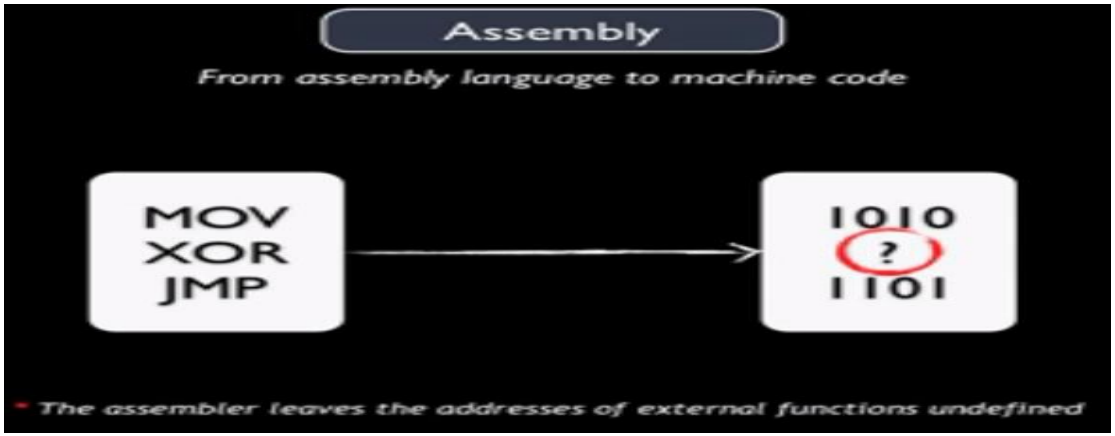
- ✓ The compilation process is a sequence of various phases. Each phase takes input from its previous stage, and it has its own representation of source program, and feeds its output to the next phase of the compiler.
- ✓ There are two phases responsible for mapping source program into a semantically equivalent target program: **analysis and synthesis phases.**
- ✓ In **analysis phase**, an intermediate representation is created from the given source program. The *analysis* part breaks up the program into essential pieces and executes grammatical structure on them.
- ✓ If analysis part detects errors (syntax and semantic), it provides informative messages.
- ✓ The analysis part also collects information about the source program and stores it in a data structure called ***symbol table***.
- ✓ The analysis phase ***reads the source program, divides it into core parts, and then checks for lexical, grammar, and syntax errors.***
- ✓ *Analysis (machine independent/language dependent phase) determines the operations implied by* the source program which are recorded in a tree structure.
- ✓ The **synthesis part** generates the desired target program from the intermediate representation and the information in the symbol table.
- ✓ *Synthesis (machine dependent/language independent phase) takes the tree structure and translates* the operations into the target program.

Assembler , Compiler and Interpreter

- Assembler: takes source code written in Assembly Language ↓ convert to machine language. fast ↓ small size executable but tedious to write
- Compiler: takes source code and converts it into executable code. Some compilers convert it into a binary file that must then be linked with several other libraries of code before it can execute, other compilers can compile straight to executable code ↓ other compilers convert it to a sort of tokenized code that still needs to be semi-interpreted by a VM
- Interpreter: does not compile code. Instead, it typically reads source code statement by statement, produces ↓ executes machine instructions on the fly. Most early forms of BASIC were interpreted languages. Slow execution



Assembler: It converts an assembly language(low-level) in to machine code (i.e. binary representation).



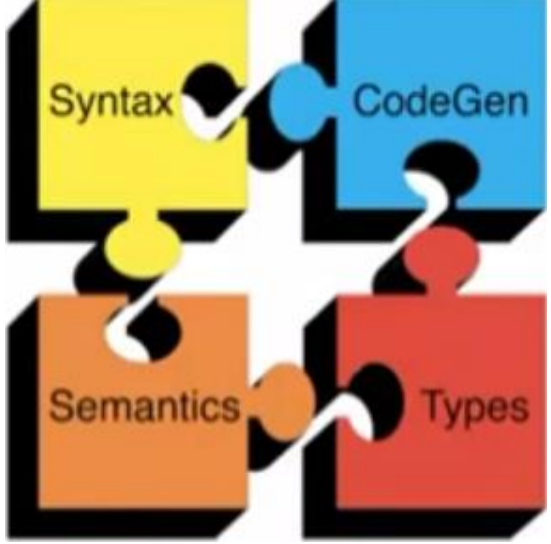
Function of loader

- ✓ The process of **Loading** consists of taking re-locatable machine code, altering the re-locatable addresses and placing the altered instructions and data in memory at the proper location.
- ✓ The **linker** allows us **to make a single program** from several files of re-locatable machine code.

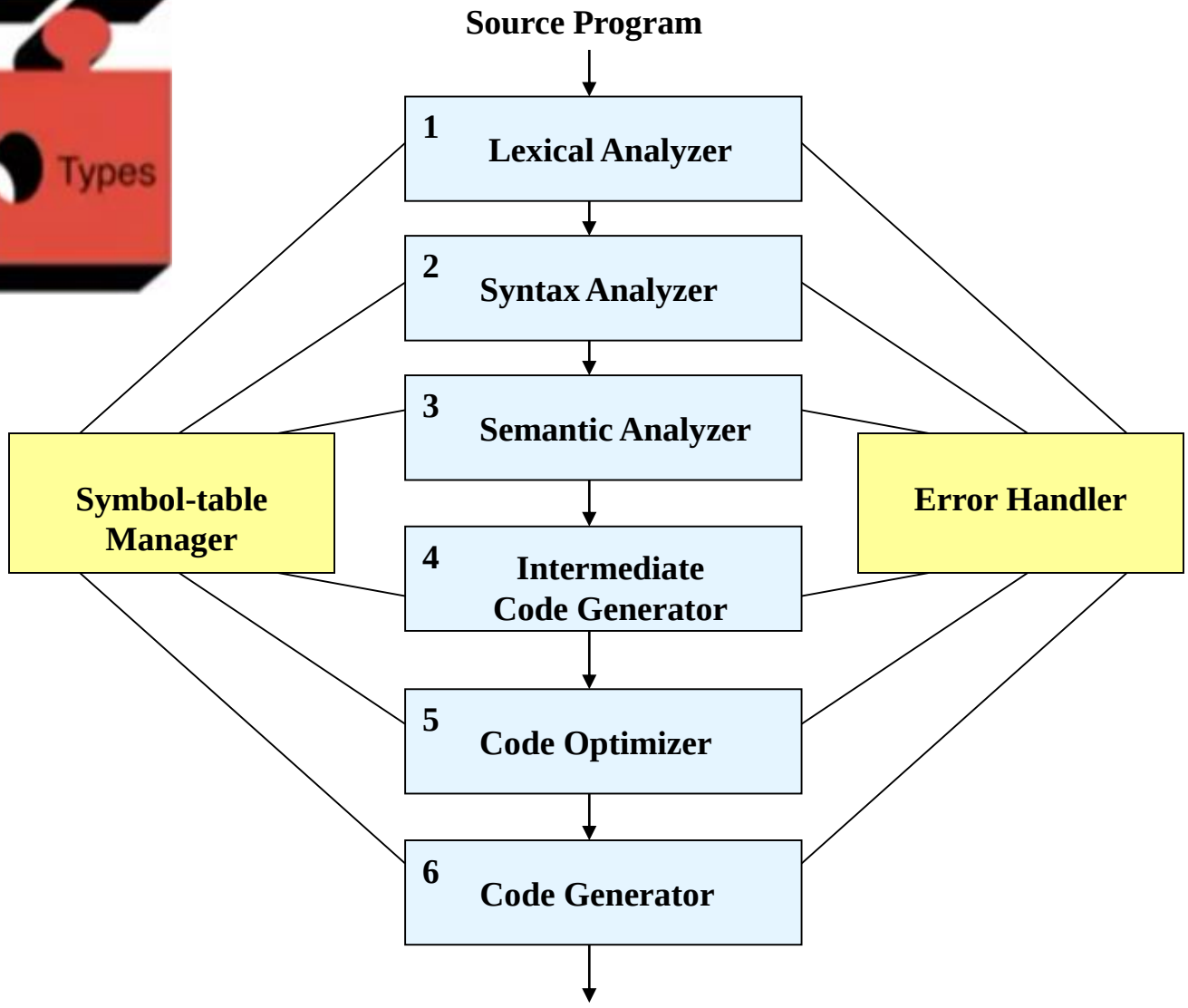
Analysis of the source program:

- Analysis consists of 3 phases:

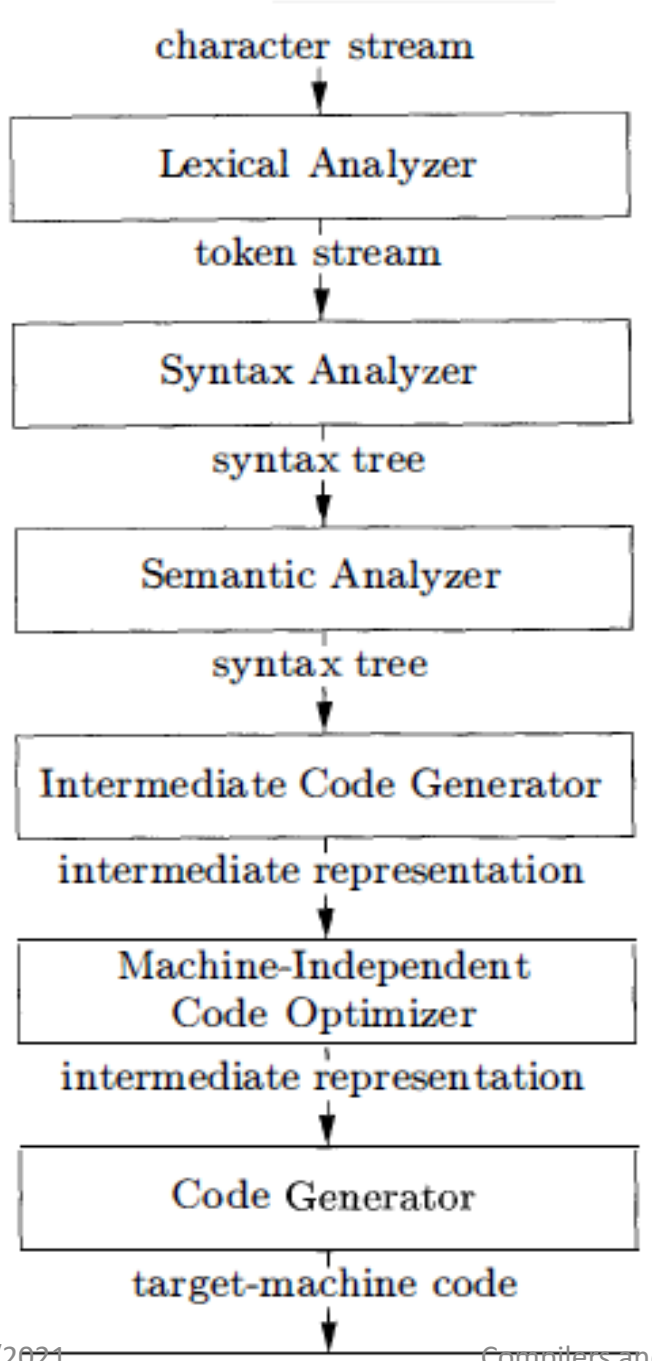
1. Linear analysis in which the stream of characters making up the source program is read from left-to-right and grouped in to tokens that are sequences of characters having a collective meaning.
2. Hierarchical Analysis in which characters or tokens are grouped hierarchically in to nested collections with collective meaning.
3. Semantic Analysis in which certain checks are performed to ensure that the components of a program fit together meaningfully.



Phases of a Compiler



Phases of a Compiler



Analysis of the source program:

The Analysis Phase consists of 3 phases:

- Linear analysis or Lexical analysis or Scanning.
- Hierarchical analysis or Syntax analysis or parsing.
- Semantic analysis.

The Synthesis Phase consists of 3 phases:

- Intermediate code Generation.
- Code Optimization.
- Code Generation.

Lexical Analysis

- ✓ The first phase of a compiler is called *lexical analysis* or *scanning*. The lexical analyzer reads the stream of characters making up the source program and groups them into meaningful sequences called lexemes.
- ✓ It reads a Stream of characters is grouped into tokens.
- ✓ Examples of tokens are identifiers, reserved words, integers, doubles or floats, delimiters, operators and special symbols

int a;

a = a + 2;

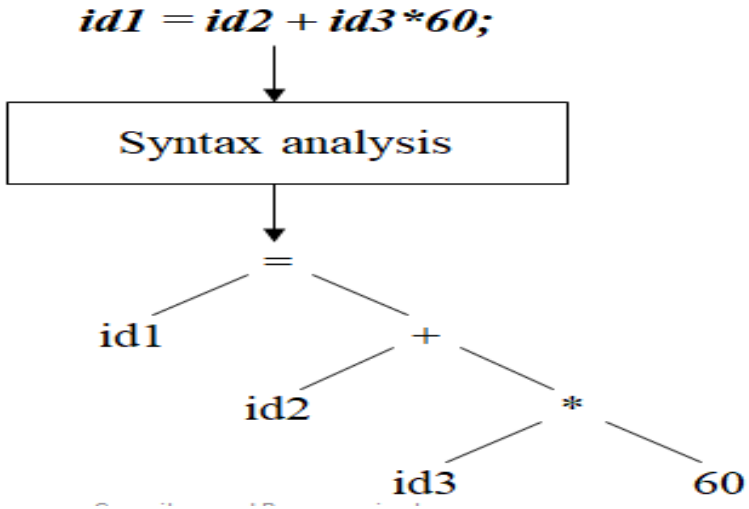
int	**reserved word**
a	**identifier**
;	**special symbol**
a	**identifier**
=	**operator**
a	**identifier**
+	**operator**
2	**integer constant**
;	**special symbol**

Syntax Analysis or Parsing

- ✓ The second phase of a compiler is ***syntax analysis*** **or** ***parsing***. The parser uses the tokens produced by the lexical analyzer to create a tree-like intermediate representation – called syntax tree – that depicts the grammatical structure of the token stream.

- ✓ In a syntax tree, each interior node represents an operation and the children of the node represent the influences of the operation.

Example of grammar rules:

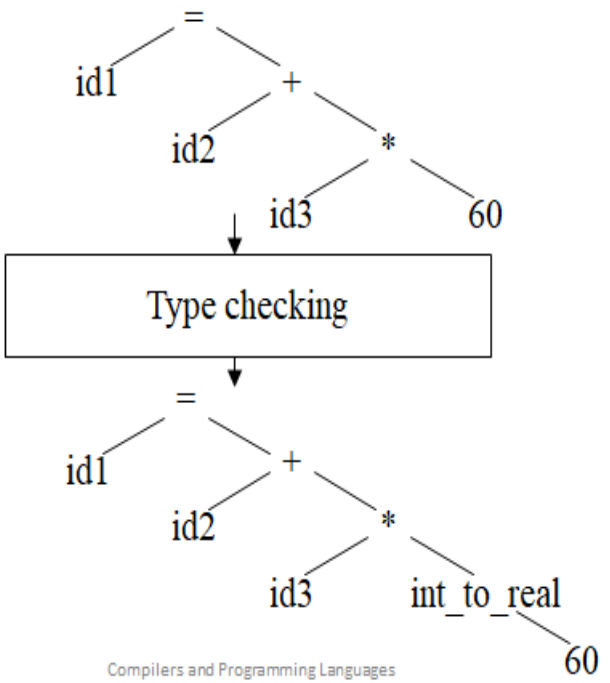


- ✓ This tree shows the order in which the operations in the assignment *position* = *initial* + *rate* * 60 are to be performed.

Semantic Analysis

- ✓ The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition.
- ✓ Checks source program for semantic errors (e.g. type checking).
- ✓ Uses hierarchical structure determined by syntactic analysis to determine operators and operands.
- ✓ Parse tree is checked for things that violates the semantic rules of the language
  - ✓ Semantic rules may be written with an attribute grammar

- Examples:
  - Using undeclared variables
  - Array variables used without array syntax
  - Type checking of operator arguments
  - Left hand side of an assignment must be a variable



- ***Intermediate Code Generation:*** After syntax and semantic analysis some compilers generate an intermediate representation of the source program.
- This representation should be ***easy to produce and easy to translate in to the target program.***
- Thus the intermediate code generation phase transforms the parse tree in to an intermediate language representation of the source program.
- ***Code optimization :*** *in this* phase improves the intermediate code i.e. it reduces the code by removing the repeated or unwanted instructions from the intermediate code, so that better target code will result. Usually better means faster, desired, consumes less power..etc
- ***Code Generation:*** in this phase converts the ***intermediate code*** in to a target code consisting of sequenced machine code or assembly code that perform the same task.
- ***Symbol Tables:*** A symbol table management or book keeping is a portion of the compiler which keeps track of the names used by the program and records information(attributes).
- The data structures used to record, this information is called a symbol table.

Error Handling

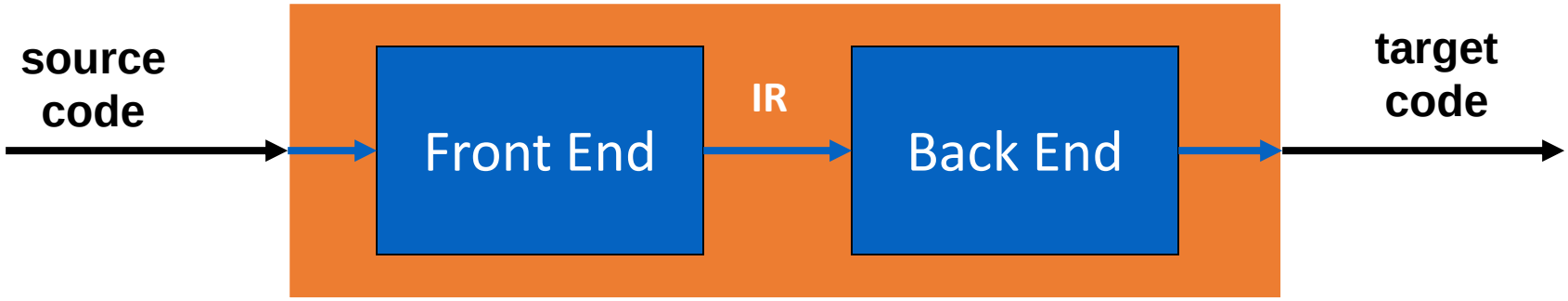
- ✓ Error handling and reporting also occurs across many phases
  - Lexical analyzer reports invalid character sequences.
  - Syntactic analyzer reports invalid token sequences.
  - Semantic analyzer reports type and scope errors.
- ✓ The compiler may be able to continue with some errors, but other errors may stop the process.
- ✓ The main functionality of a compiler is the detection and reporting of errors in the source program.

- detection
- Recovery/Repair
- correction

Front-end and back-end

- Compilation phases that are more dependent on the source language and to a less extent on the target language are called *front-end* .
- Compilation phases that are more dependent on the target language and less dependent on the source language are called *back-end.*
- Certain phases of the compiler can be grouped together and can be called as a PASS.
- Generally the first 4 phases are grouped together and called as pass1.
- This pass1 includes lexical analyzer, syntax analyzer, semantic analyzer and intermediate code generator phases.
- This section can be called as "front end" of the compiler.
- These are machine independent phases.
- The other 2 phases code generation and code optimizer are grouped together and called as pass2.
- This section can be called as "back end" of the compiler.
- These are machine dependent phases.

Front-end and back-end



Advantages and Disadvantages of Compilers:

- Advantages: are Optimization
  flexibility.
- Disadvantage is Lack of Speed.

Questions and Answers