COMPILER CONSTRUCTION
CS F363

Introduction to Compilers and Phases

Dr. R. Elakkiya, AP/CS

BITS Pilani
Dubai Campus

Text Book

- ✓Aho, Lam, Sethi and Ullman, “Compilers-Principles, Techniques & Tools”, Pearson/Addison-Wesley, Second Edition, 2013.

Study Materials

- ✓ **Text Book(s) [T]**
  - ✓ **T1:** Aho, Lam, Sethi and Ullman, “Compilers-Principles, Techniques & Tools”, Pearson/Addison-Wesley, Second Edition, 2013.
- ✓ **Reference Book(s) [R]**
  - ✓ **RB1:** Andrew W Appel & Jens Palsberg, “Modern Compiler Implementation in JAVA”, Cambridge University Press, Revised Edition, 2002.
  - ✓ **RB2:** Ravi Sethi, “Programming Languages: Concepts & Constructs”, Pearson International Edition, 2003, ISBN:0-321-21074-3.
- ✓ **Class Notes**
  - ✓ Follow LMS for updates

Scope and Objective of the Course

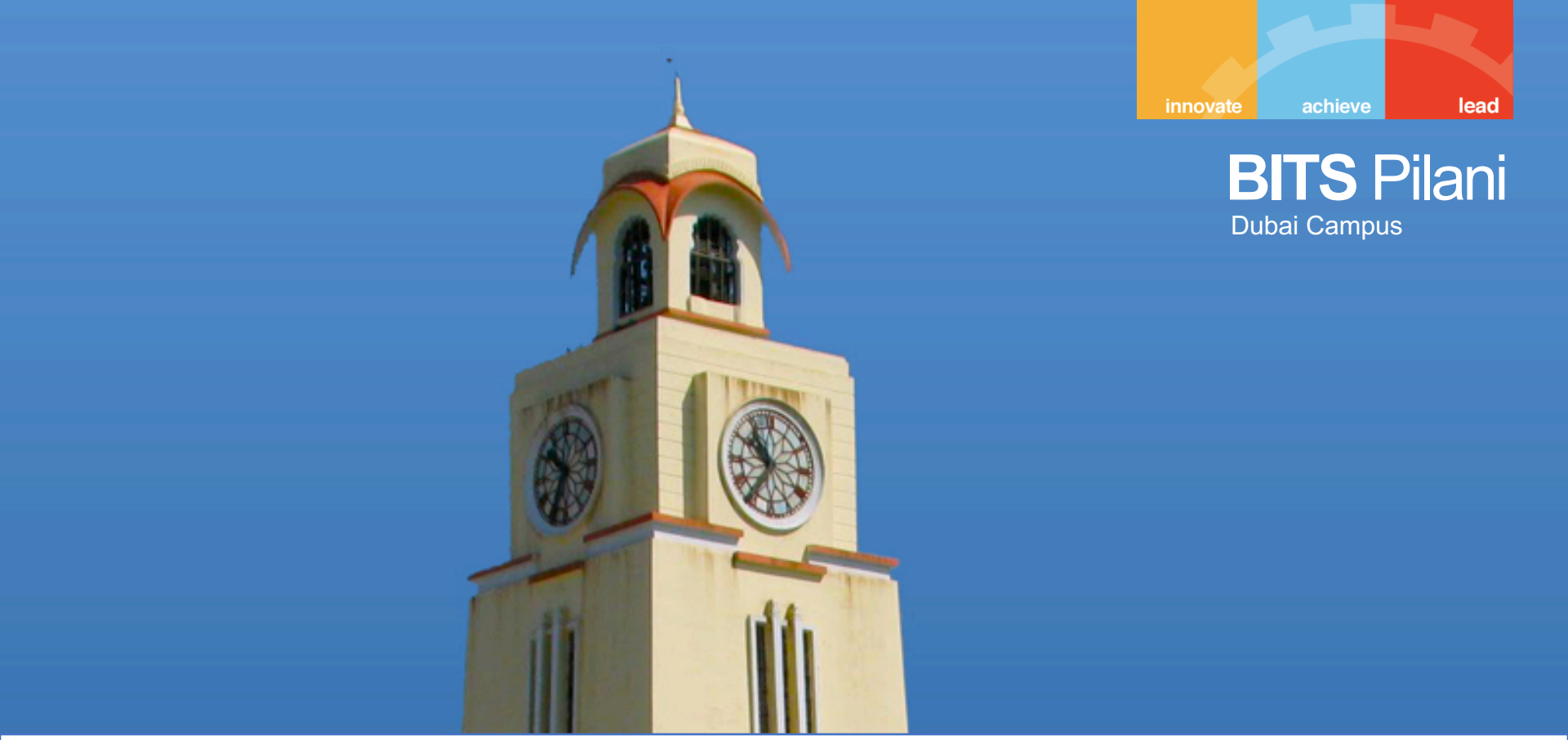
- ✓ Introducing the Lexical and Syntactic structure of a Language, the different phases of Compiler Design

- ✓ To understand and apply the principles of compiler design in the development of system software tools & applications

Evaluation Components

EC No.	Evaluation Components	Nature of Component	Weightage %	Date – Time & Duration	Venue
1	Midsem Exam	Open book*	30	29.03.24 FN	TBA
2	Lab Midsem Exam	Closed book	15	20.03.24(34), 21.03.24(89), 22.03.24(34) (3 batches)	333 Lab
3	Lab Comprehensive Exam	Closed book	15	TBA	333 Lab
4	Comprehensive Exam	Closed book	40	29.05.24 AN	TBA

* Only prescribed text book(s) and hand-written notes are permitted



Text Book Reading:

Chapter 1

Section 1.1, 1.2

Overview

- ✓In this Lecture
  - ✓Introduction
  - ✓Language Preprocessors
  - ✓Structure of Compiler
    - ✓Phases of Compiler

Introduction

Introduction

Introduction

Introduction

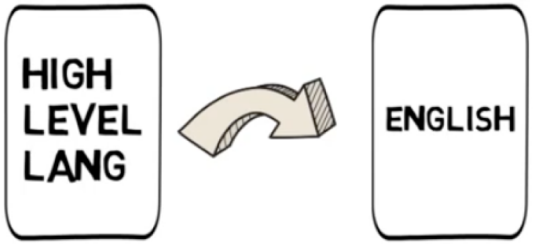

```
public class simple{
    public static void main(String[] args){
        System.out.println("Hello World")
    }
}
```

SIMPLE JAVA PROGRAM

Introduction


```
public class simple{
    public static void main(String[] args){
        System.out.println("Hello World")
    }
}
```

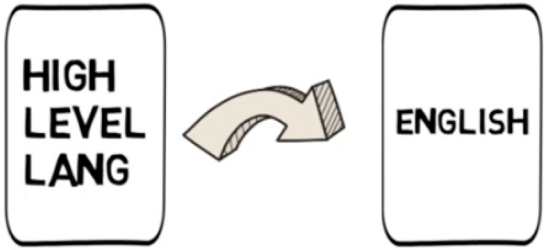
SIMPLE JAVA PROGRAM



Introduction

```
public class simple{
    public static void main(String[] args){
        System.out.println("Hello World")
    }
}
```

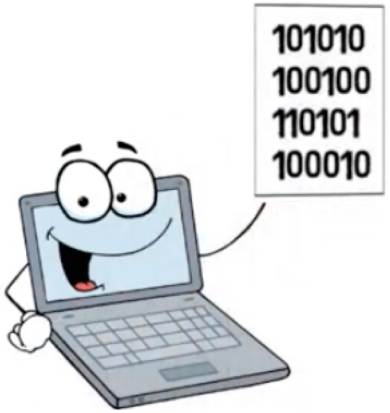
SIMPLE JAVA PROGRAM

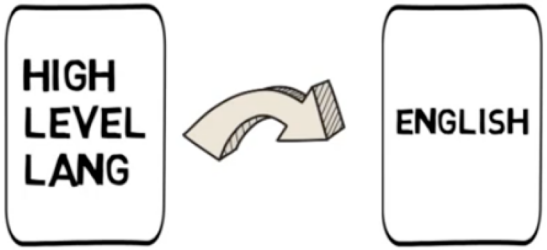


Introduction

```
public class simple{
    public static void main(String[] args){
        System.out.println("Hello World")
    }
}
```

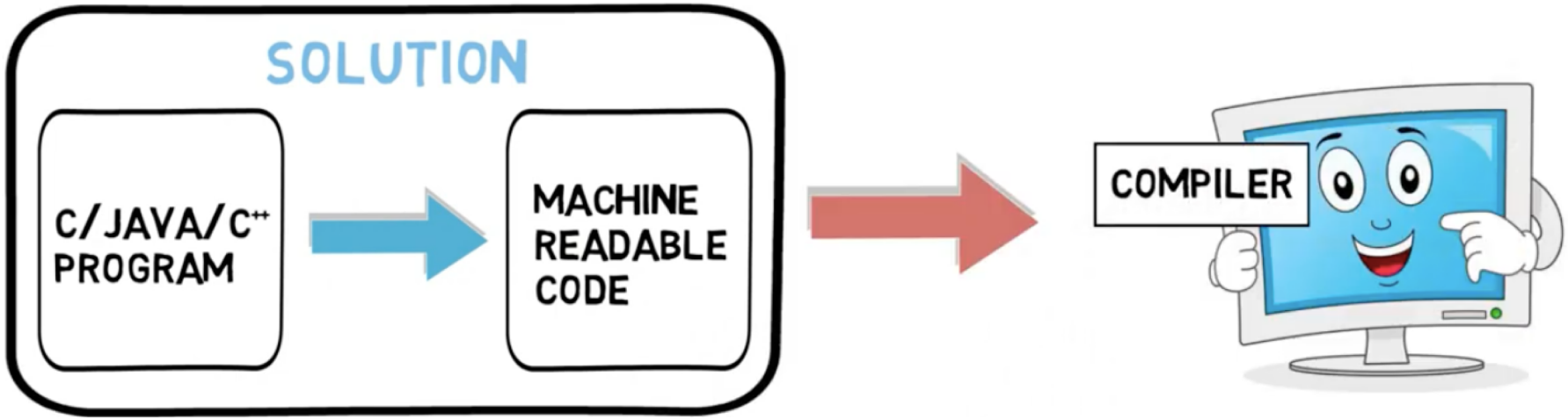
SIMPLE JAVA PROGRAM





Introduction

Introduction



Introduction

MACHINE READABLE CODE

SOLUTION

C/JAVA/C++ PROGRAM

MACHINE READABLE CODE

Introduction

- ✓What are Compilers?
  - ✓ System software (It is a program!)
  - ✓ Language processor
  - ✓ Translates source program into target program (i.e.) one representation/high level language program to another representation/low level language program

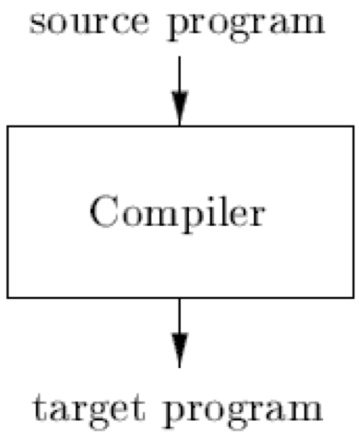
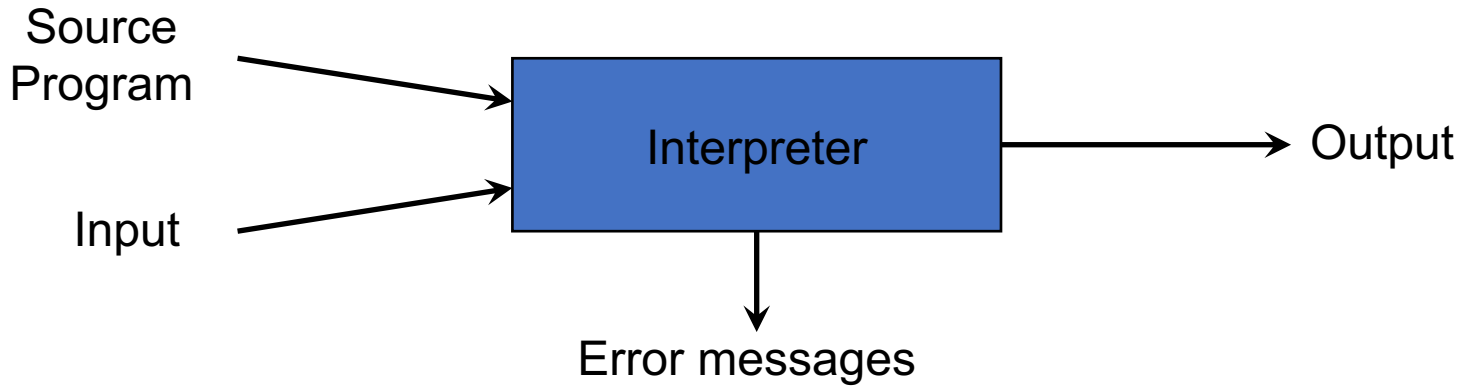


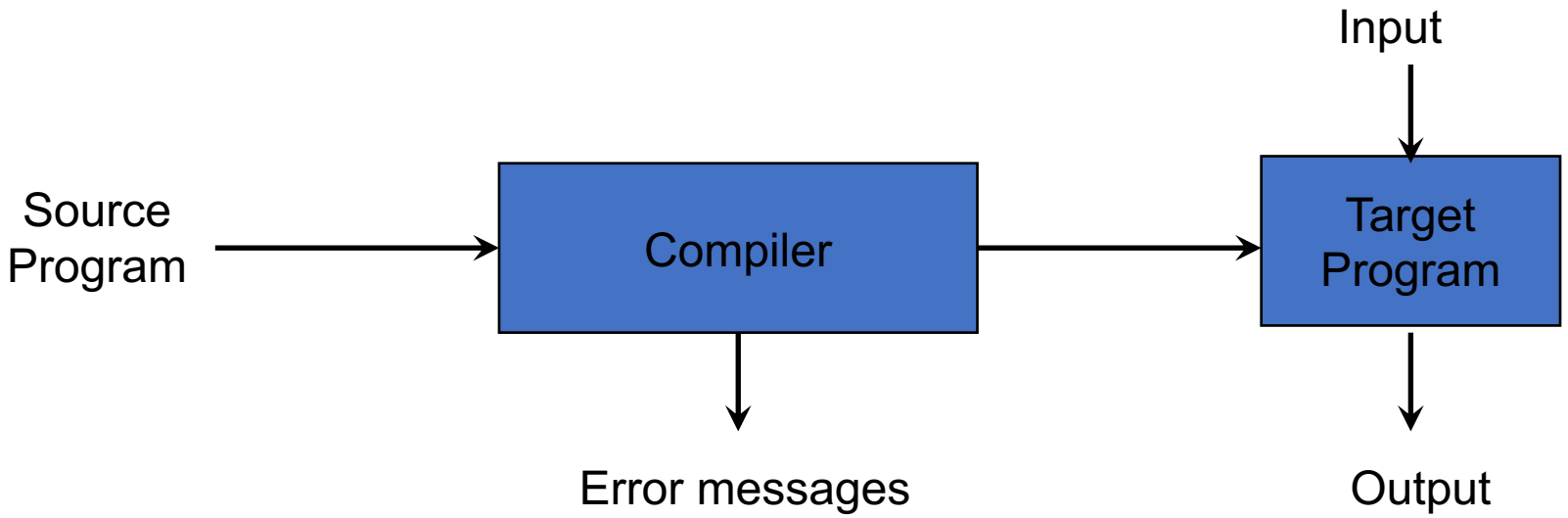
Figure 1.1: A compiler

Introduction

- ✓Difference between compiler and interpreter
  - ✓Interpreter
    - ✓Does not generate target program
    - ✓Execute the operations specified in the source program on inputs supplied by the user
    - ✓Give better error diagnostics than a compiler
    - ✓Slower than a compiler

Introduction





Language Processors

✓Java combines compilation and interpretation

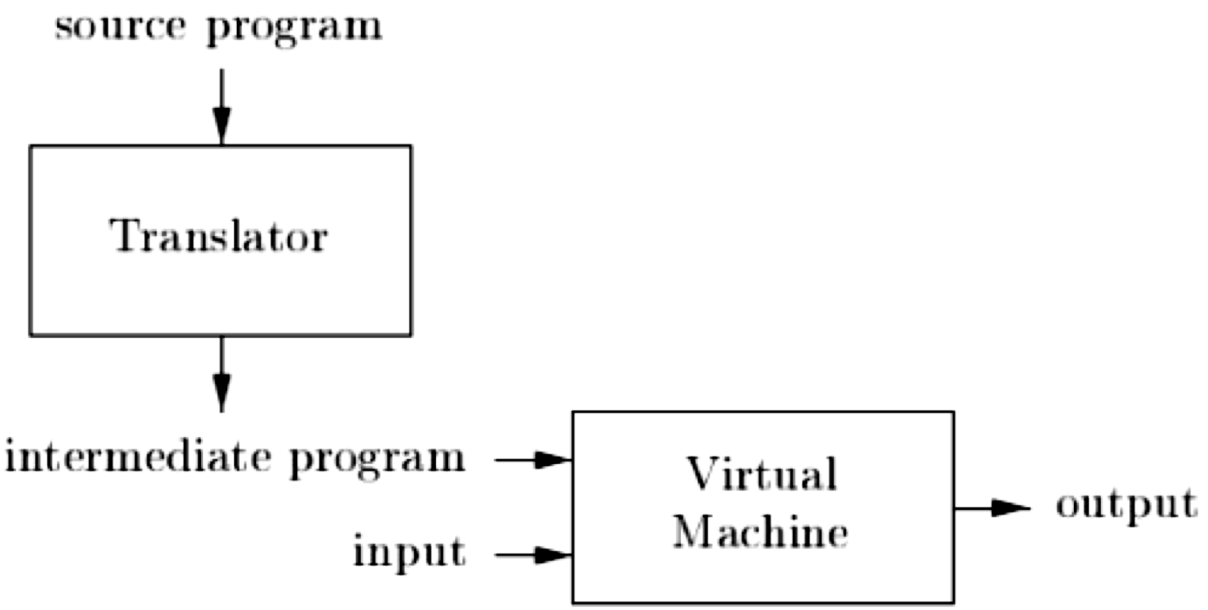
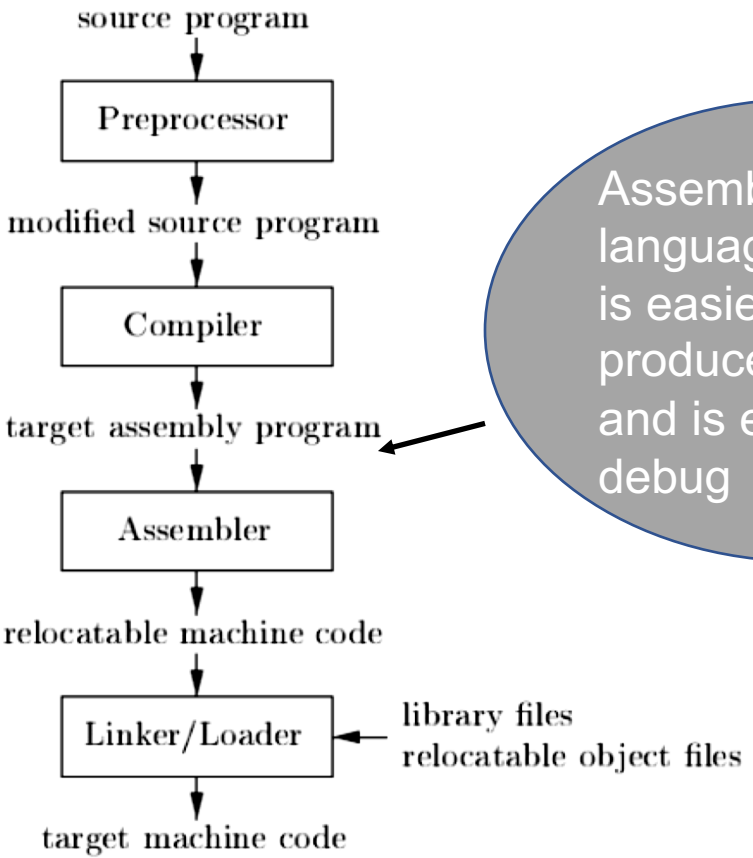


Figure 1.4: A hybrid compiler

Language Processors

✓A language-processing system / program development environment



linker resolves external memory addresses, where the code in one file may refer to a location in another file

Assembly language is easier to produce as output and is easier to debug

Figure 1.5: A language-processing system

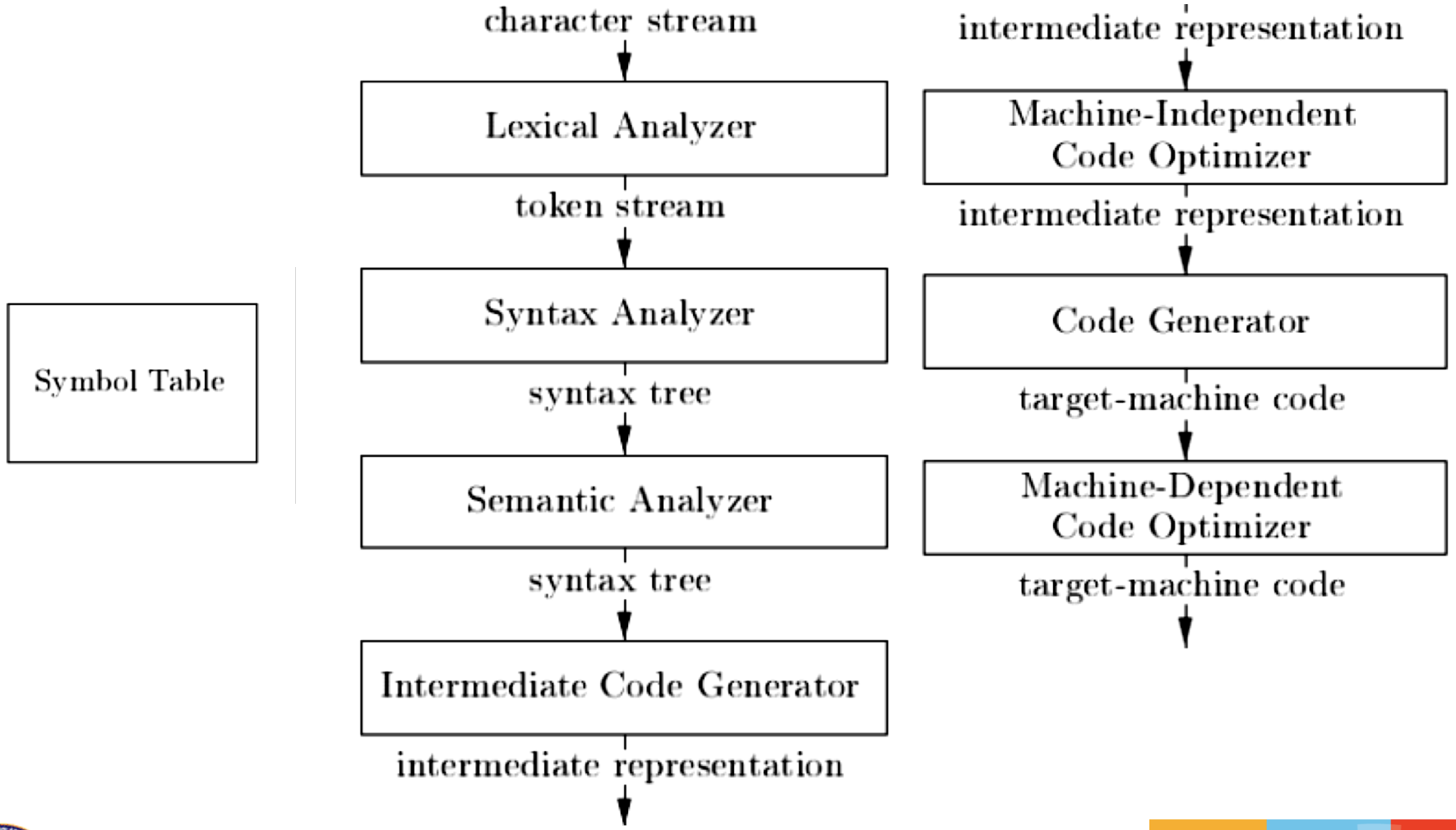
The Structure of a Compiler

- ✓Compiler has two parts
  - ✓analysis part / front end
  - ✓synthesis part / back end
- ✓Analysis part
  - ✓breaks up the source program into constituent pieces
  - ✓imposes a grammatical structure on them
  - ✓create an intermediate representation
  - ✓Provide error messages (syntax/semantics)
  - ✓collects information about the source program and stores it in a data structure called a symbol table

The Structure of a Compiler

- ✓Synthesis part
  - ✓Receives symbol table and intermediate representation from analysis part
  - ✓Constructs the desired target program

Phases of a Compiler



Example

- ✓Position = initial + rate * 60
- ✓Generate the target machine code for the given expression

Phases of a Compiler

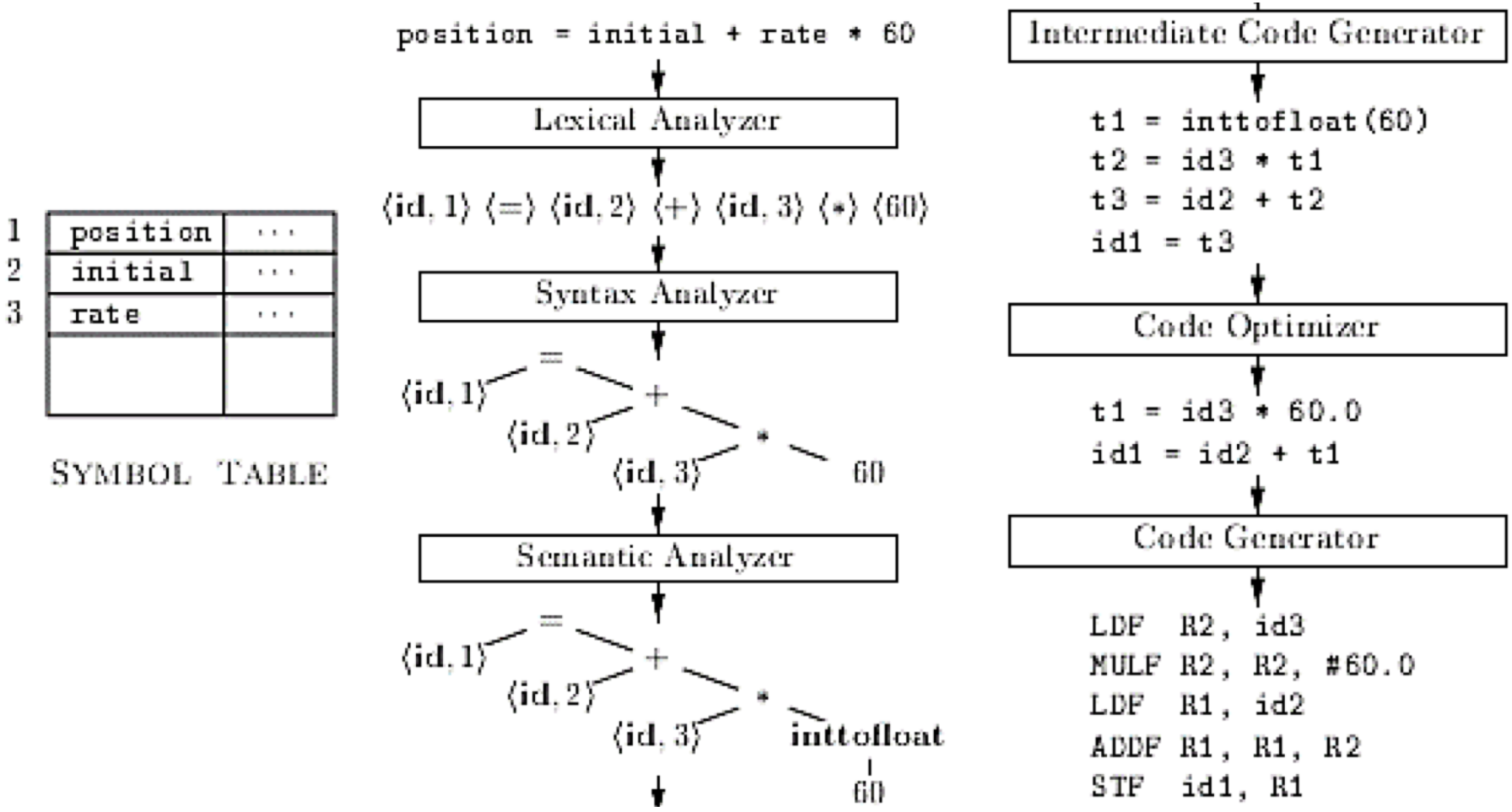


Figure 1.7: Translation of an assignment statement

Phases of a Compiler

- ✓ Symbol-Table
  - ✓ The symbol table is a data structure containing a record for each identifier (variables, constants, functions) with fields for the attributes of the identifier
  - ✓ Attributes of variables are name, type, scope, etc.
  - ✓ Attributes of procedure names, may provide information about the number and types of its arguments, the method of passing each argument (for example, by value or by reference), and the type returned

Phases of a Compiler

- ✓ Symbol-Table
- ✓ Accessed in every phase of compiler
  - ✓ The scanner, parser, and semantic analyzer put names of identifiers in symbol table.
  - ✓ The semantic analyzer stores more information (e.g. types) in the table.
  - ✓ The intermediate code generator, code optimizer and code generator use information in symbol table to generate appropriate code.
- ✓ Mostly use hash table for efficiency.

Phases of a Compiler

- ✓ Grouping of Phases into Passes
  - ✓ A pass reads an input file process it and writes an output file
  - ✓ It can involve several phases
  - ✓ Normal passes in compilers
    - ✓ front-end phases are combined into a pass
    - ✓ code optimization is an optional pass
    - ✓ back-end phase can be made into a pass

Phases of a Compiler

- ✓ Lexical Analysis/Scanning
  - ✓ Groups the characters of source program into meaningful sequences called lexeme
  - ✓ For each lexeme, the lexical analyzer produces as output a token
  - ✓ The token is passed on to syntax analysis phase
  - ✓ **token-name** is an abstract symbol
  - ✓ **attribute-value** points to an entry in the symbol table for this token (needed for semantic analysis and code generation)

Phases of a Compiler

- ✓ Syntax analysis/parsing
  - ✓ Uses tokens produced by the lexical analyzer determines if the syntax (structure) of the program is correct according to the (context-free) grammar of the source language
  - ✓ Creates a tree-like intermediate representation (parse tree/syntax tree) that depicts the grammatical structure of the token stream
  - ✓ Syntax tree: Each interior node represents an operation and the children of the node represent the arguments of the operation. It shows the order of operations

Phases of a Compiler

- ✓ Semantic analysis
  - ✓ Uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition
  - ✓ Gathers type information and saves it in either the syntax tree or the symbol table
  - ✓ Type checking: each operator has matching operands eg. array index must be an integer

Phases of a Compiler

- ✓ Semantic analysis
  - ✓ Some language specifications may allow type conversions called coercions
  - ✓ Eg. a binary arithmetic operator may be applied to either a pair of integers or to a pair of floating-point numbers. If the operator is applied to a floating-point number and an integer, the compiler may convert or coerce the integer into a floating-point number

Phases of a Compiler

- ✓ Intermediate Code (IC) Generation
  - ✓ IC: low-level or machine-like intermediate representation
  - ✓ Takes a parse tree from the semantic analyzer and generates a program in the intermediate language
  - ✓ Properties of IC:
    - ✓ easy to produce
    - ✓ easy to translate into the target machine
  - ✓ Three-address code (TAC)
    - ✓ which consists of a sequence of assembly-like instructions with three operands per instruction

$$x = y \; op \; z$$

  - ✓ Postfix notation

Phases of a Compiler

- ✓ Code Optimization
  - ✓ improve the intermediate code so that better (faster/use lesser resources like memory, space, registers and power) target code will result
  - ✓ Note: Optimization may consume time (eg. Optimizing compilers)
- ✓ Code optimization can be done:
  - ✓ after intermediate code generation (M/C independent code optimization)
    - ✓ performed on a intermediate code
  - ✓ after code generation (M/C dependent code optimization)
    - ✓ performed on a target code

Phases of a Compiler

- ✓ Error detection and reporting
  - ✓ Done at every phase if any error is identified it is reported and handled
  - ✓ Lexical error: Misspelling an identifier or keyword Eg. Fi (a == b) (fi can be an identifier, misspelled keyword or a function name. But lexical analysis considers it as identifier)
  - ✓ Syntax error: An arithmetic expression with unbalanced parenthesis
  - ✓ Semantic error: operator applied to incompatible operand types
  - ✓ Logical error: Infinite recursive call

Phases of a Compiler

- ✓ Code Generation
  - ✓ Takes as input an intermediate representation of the source program
  - ✓ Produces the target language (machine code or assembly language code)
  - ✓ Choose registers and memory locations for the variables in the program

References

- ✓Material covers text book ch1 sections1.1and1.2

Summary

- ✓In this Lecture
  - ✓Introduction
  - ✓Language Preprocessors
  - ✓Structure of Compiler
    - ✓Phases of Compiler

Thank you

Tutorial - I

- Let's consider a scenario where you're developing a compiler, and a programmer presents a financial calculation involving three variables:
  - **loan_amount** representing the total amount borrowed,
  - **interest_rate** denoting the annual interest rate,
  - **loan_term** indicating the duration of the loan in years.
- The programmer needs to determine the monthly loan repayment amount using the formula:
  - monthly_payment = loan_amount * (interest_rate / 12) / (1 - (1 + (interest_rate / 12))+(-loan_term * 12))
- As you guide the programmer through the compilation process, provide step-by-step solutions for each phase:
  - Identify and list the individual tokens present in the expression.
  - Construct the Syntax Tree representing the syntactic structure of the expression.
  - Check for any semantic errors or inconsistencies within the expression.
  - Create the intermediate code representation of the expression.
  - If possible, optimize the intermediate code to enhance performance. Display the optimized version.
  - Translate the optimized code into the final machine code, making it ready for execution.