

# Outline

1

- Introduction about compiler design
- Phases of a Compiler
- Compiler Construction Tools
- Type of compiler
- Advantage and disadvantage of Compiler Design

# Introduction

2

- The **compiler** is software that converts a program written in a high-level language (Source Language) to a low-level language (Object/Target/Machine Language/0, 1's).
- A translator or language processor is a program that translates an input program written in a programming language into an equivalent program in another language.
- The compiler is a type of translator, which takes a program written in a high-level programming language as input and translates it into an equivalent program in low-level languages such as machine language or assembly language.

# Cont...

3

- The program written in a high-level language is known as a source program, and the program converted into a low-level language is known as an object (or target) program.
- Without compilation, no program written in a high-level language can be executed.
- The process of translating the source code into machine code involves several stages, including lexical analysis, syntax analysis, semantic analysis, code generation, and optimization.

# Definitions

4

**Compile: collect material into a list, volume.**

What is a compiler?

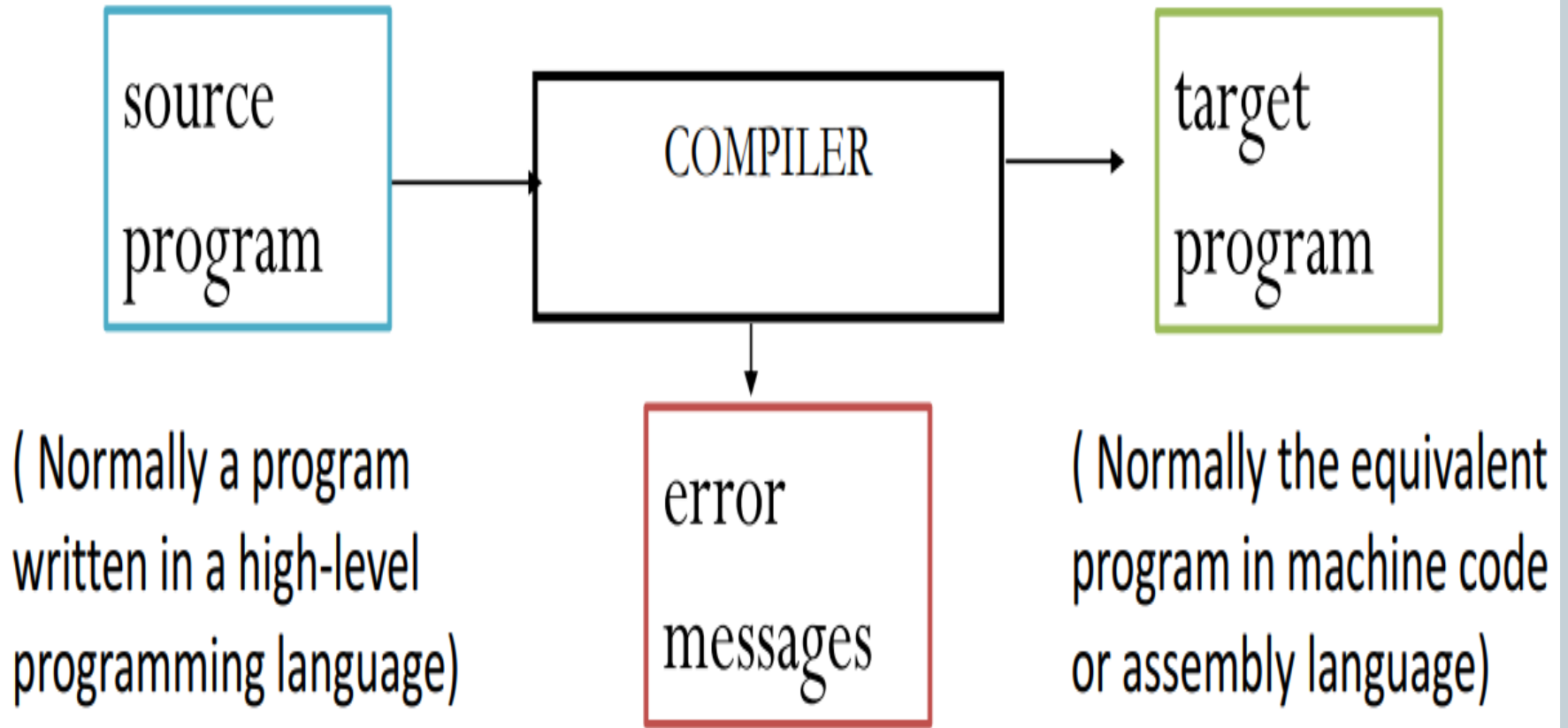
- A program that accept as input a program text in a certain language and produces as output a program text in another language .
- A program that reads a program written in one language (source language) and translate it into an equivalent program in another language(target language).

What is an interpreter?

- A program that reads a source program and produces the results of executing this source.
- We deal with compilers many of these issues arise with interpreters.

# Cont...

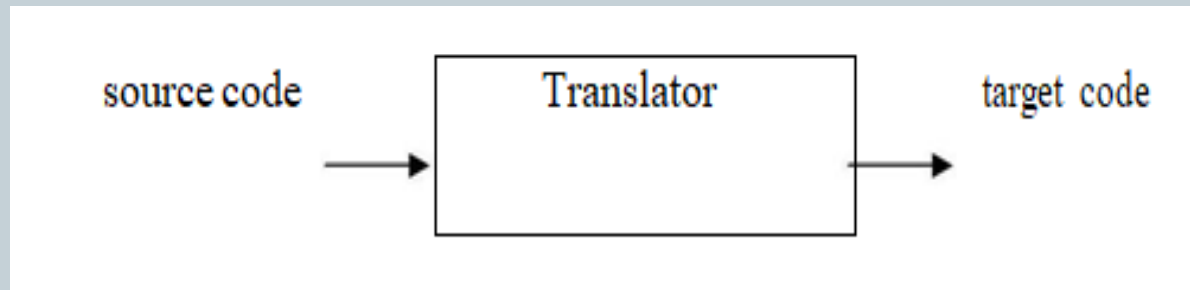
5



# LANGUAGE TRANSLATORS

6

- ▶ **Translator:** It is a program that translates **one** language to another.



## ✱ Types of Translator:

- ✓ Compiler
- ✓ Interpreter
- ✓ Assembler

# Compiler

7

- ✓ The function of the compiler is to accept statements and translate them into sequences of machine language operations.
- ✓ It is important to bear in mind that when processing a statement such as  $x = x * 9;$
- ✓ The compiler does not perform the multiplication.
- ✓ The compiler generates, as output, a sequence of instructions, including a "multiply" instruction.

# Cont...

8

✓ If a portion of the input to a C++ compiler looked like this:

$A = B + C * D;$

✓ The output corresponding to this input might look something like this:

- LOD R1, C     // Load the value of C into reg 1
- MUL R1, D     // multiply the value of D by reg 1
- STO R1, TEMP1 // Store the result in TEMP1
- LOD R1, B     // Load the value of B into reg 1
- ADD R1, TEMP1 // Add value of Temp1 to register 1
- STO R1, TEMP2 // Store the result in TEMP2
- MOV A, TEMP2   // Move TEMP2 to A, the final result



# Interpreter

9

- ✓ It is one of the translators that translate high level language to low level language.
- ✓ An interpreter is another commonly used language processor.
- ✓ An interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user.

high level language

Input

Interpreter

low level language

During execution, it checks line by line for errors.

# Assembler

10

- It translates assembly level language to machine code.

assembly language



Assembler



machine code

Example: Microprocessor 8085, 8086.

# Properties of Compiler

11

- ✓ Correctness
  - ✓ Correct output in execution.
  - ✓ It should report errors
  - ✓ Correctly report if the programmer is not following language syntax.
- ✓ Efficiency
- ✓ Compile time and execution.
- ✓ Debugging / Usability.

# Compiler vs interpreter

12

- A compiler takes entire program and converts it into object code which is typically stored in a file.
- ✓ The object code is also referred as binary code and can be directly executed by the machine after linking.
- ✓ Examples of compiled programming languages are C and C++.
- An Interpreter directly executes instructions written in a programming or scripting language without previously converting them to an object code or machine code.
- ✓ Examples of interpreted languages are Perl, Python and Matlab.

# Cont...

13

Following are some interesting facts about interpreters and compilers.

- 1) Both compilers and interpreters convert source code (text files) into tokens,
  - both may generate a parse tree, and both may generate immediate instructions.
  - The basic difference is that a compiler system, including a (built in or separate) linker, generates a stand alone machine code program, while an interpreter system instead performs the actions described by the high level program.
- 2) Once a program is compiled, its source code is not useful for running the code. For interpreted programs, the source code is needed to run the program every time.
- 3) In general, interpreted programs run slower than the compiled programs.
- 4) Java programs are first compiled to an intermediate form, then interpreted by the interpreter.

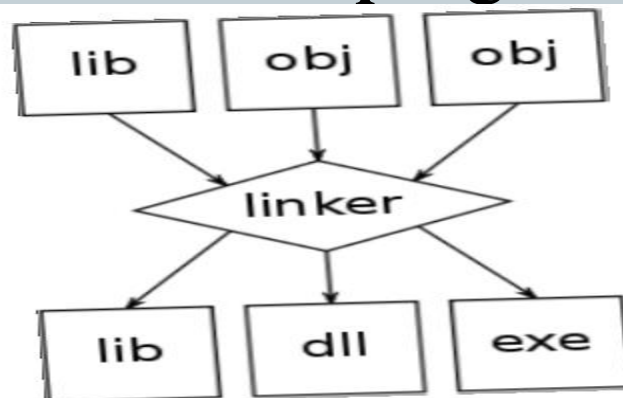
# Programs related to compilers

14

- ❑ **Assemblers** - convert a program in assembly language to machine language



- ❑ **Linkers** - a computer program that takes object files generated by compilers or assemblers and combines them into a single executable program.



# Cont...

15

- ✓ **Loaders**- loads an executable into memory and starts it running.
- ✓ **Editors** – programs used to write/edit source codes
- ✓ **Debuggers** – programs which are used to determine execution errors in a compiled errors
- ✓ **Preprocessors** – A source program may be divided into modules stored in separate files.
  - The task of collecting the source program is sometimes entrusted to a separate program, called a *preprocessor*.
  - It may also expand short hands, called macros, into source language statements.

# Cont...

16

✓ A preprocessors produce input to compilers.

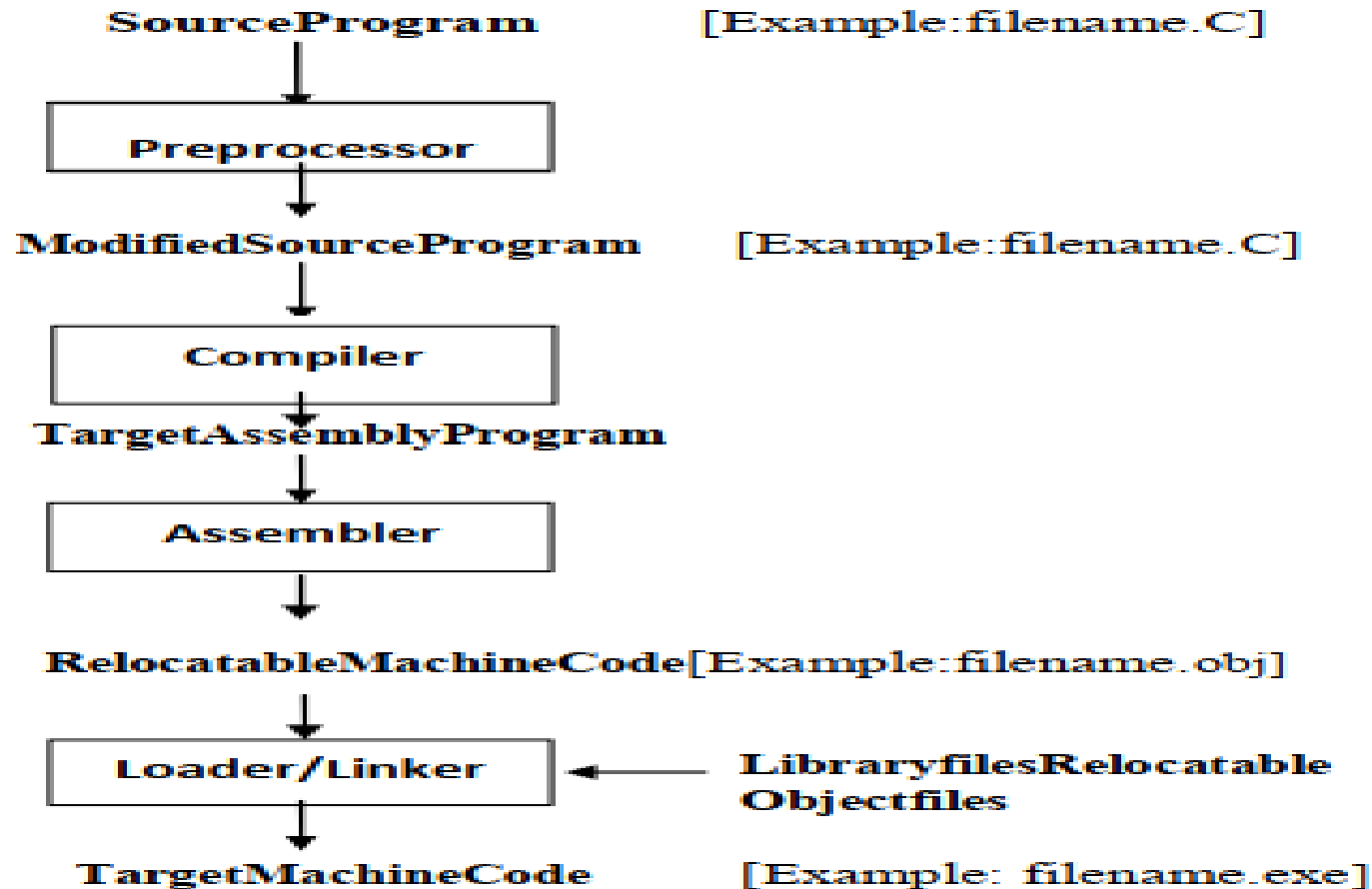
✓ They may perform the following functions.

1. Expands short hands/macros into source language statements File inclusion: A preprocessor may include header files into the program text.
2. Rational preprocessor: these preprocessors augment older languages with more modern flow-of control and data structuring facilities.
3. Language Extensions: These preprocessor attempts to add capabilities to the language by certain amounts to build-in macro.
4. Collects all the modules, files incase if the source program is divided into different modules stored at different files.



# Language Processing System

17



# The Phases of a Compiler

18

## **Analysis (front part): *Lexical, Syntax, and Semantic analysis***

- ✓ Breaks up the source program into constituent pieces
- ✓ Creates an intermediate representation of the source program
- ✓ Reports any error detected
- ✓ Stores source program info in a data structure called a symbol table
- ✓ Machine Independent/Language Dependent. b/c they depend primarily on the source language

# Cont...

19

## 👉 **Synthesis (Back part):** Code Generation + Code Optimization

- ✓ Constructs the desired **target program** from the intermediate representation and the information in the symbol table.
- ✓ Compilation process operates as a sequence of phases, each of which transforms one representation of the source program to another.
- ✓ Machine Dependent. **b/c they depend on the target machine/Language independent**
- NB: **Intermediate code generation** is between front end and back end

# Cont...

20

- Compiler is not a single box that maps a source program into a target program.

There are two parts to this mapping: analysis and synthesis

- ✱ Analysis (Machine Independent/Language Dependent)
- ✱ Synthesis (Machine Dependent/Language independent)

# Cont...

21

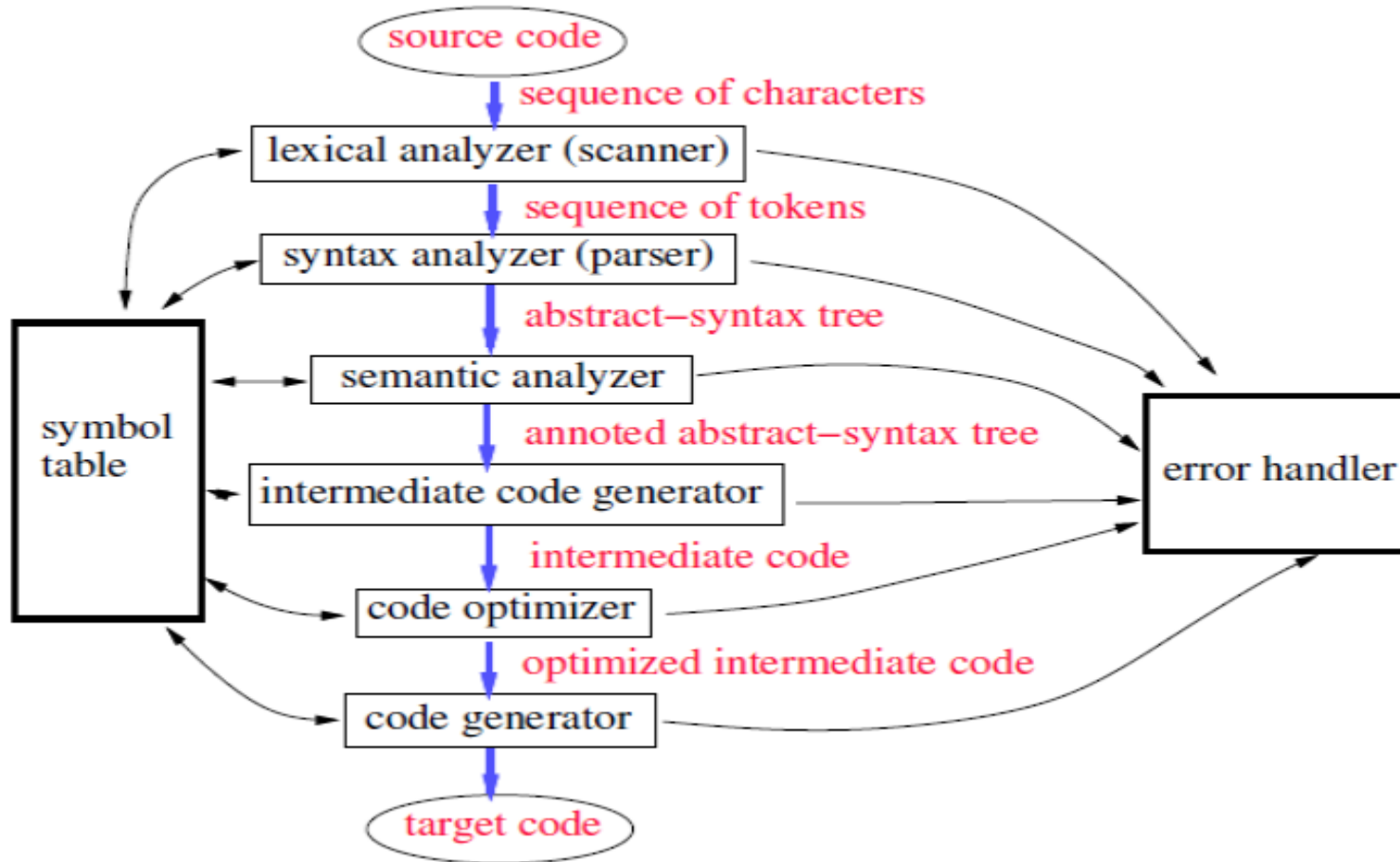


Figure : The Phases of a Compiler

# Linear/Lexical Analysis (Scanner)

22

⇒ Also called the *Lexer*

## How it works:

- ▷ Reads characters from the source program.
- ▷ Groups the characters into **lexemes** (sequences of characters that "go together").
- ▷ Each lexeme corresponds to a **token**;

i.e. **For each lexeme**, the lexical analyzer produces as output a token of the form (token-name, attribute-value)

- ▷ The scanner returns the next token (maybe some additional information) to the parser.
- ▷ The scanner may also discover lexical errors (e.g., erroneous characters).
- ▷ Start symbol table with new symbols found

# Cont...

23

- Tokens include e.g.:
  - “Reserved words”: do, if, float, while
  - Special characters: ( { , + - = ! /
  - Names & numbers: myValue, 3.07e02
- The definitions of what a **lexeme** , **token** or **bad character** is depend on the definition of the source language.
- ❖ Examples of tools for lexical analysis are
  - Lex
  - flex

# Cont...


24

- Examples of words are:
  - ✓ **Key words** - while, void, if, for, ...
  - ✓ **identifiers** - declared by the programmer
  - ✓ **operators** - +, -, \*, /, =, ==, ...
  - ✓ **numeric constants** - numbers such as 124, 12.35, 0.09E-23, etc.
  - ✓ **character constants** - single characters or strings of characters enclosed in quotes.
  - ✓ **special characters** - characters used as delimiters such as . ( ) , ; :
  - ✓ **comments** - ignored by subsequent phases. These must be identified by the scanner, but are not included in the output.



# Cont...

25

 **For example**, in the assignment statement **a=b+c\*2**, the characters would be grouped into the following tokens:

- The identifier1 'a'
- The assignment symbol (=)
- The identifier2 'b'
- The plus sign (+)
- The identifier3 'c'
- The multiplication sign (\*)
- The constant '2'

# Syntax Analysis

26

 The *syntax analysis phase* is often called the *parser*.

- ✓ The parser will check for proper **syntax**, issue **appropriate error** messages, and determine the underlying structure of the source program.
- ✓ The output of this phase may be a stream of *atoms* or a collection of *syntax trees*.
- ✓ An atom is an atomic operation, or one that is generally available with one machine language instruction(s) on most target machines.
- ✓ For example, **MULT**, **ADD**, and **MOVE** could represent atomic operations for multiplication, addition, and moving data in memory.

# Cont...

27

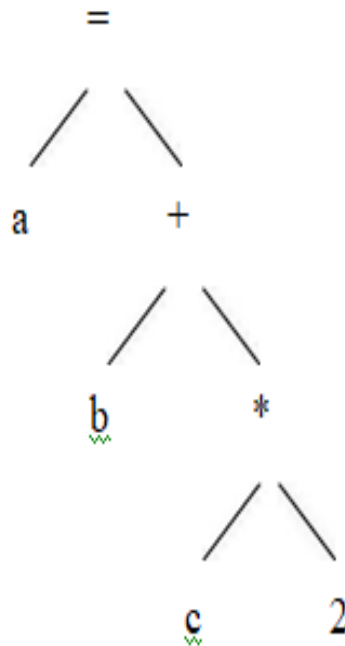
- ✓ Constructs a parse tree from symbols
- ✓ Language **grammar** defined by set of rules that identify legal (meaningful) combinations of symbols
- ✓ Each application of a rule results in a node in the parse tree
- ✓ Parser applies these rules repeatedly to the program until leaves of parse tree are “atoms”
- ✓ If no pattern matches, it’s a syntax error
- ✓ Each atom consists of three or four parts:
  - ▶ an operation, one or two operands, and a result.

# Cont...

28

Source code: `a=b+c*2`

They are represented using a syntax tree as shown below:

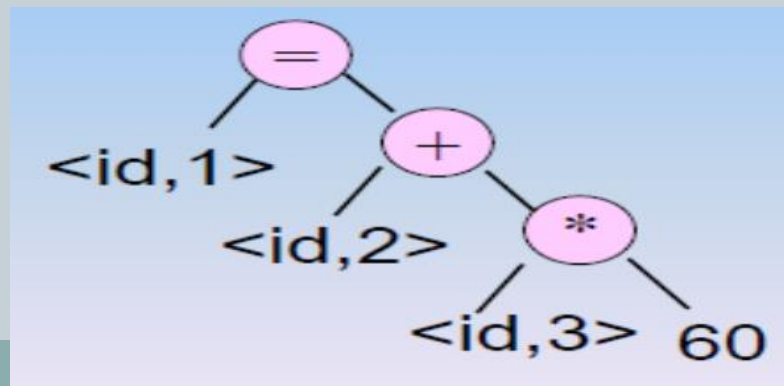


## Cont...

29

Source code: **position = initial + rate \* 60;**

- ✓ Interior nodes of the tree are OPERATORS;
- ✓ a node's children are its OPERANDS;
- ✓ each sub-tree forms a logical unit .
- ✓ The sub-tree with \* at its root shows that \* has higher precedence than +, the operation “rate \* 60” must be performed as a unit, not “initial + rate”.



# Cont...

30

- ✓ A **syntax tree** is the tree generated as a result of syntax analysis.
- ✓ This analysis shows an error when the syntax is incorrect.
- **Sample problem**
  - ✓ Show a syntax tree for the C/C++ statement
  - `if (A+3<400) A = 0; else B = A*A;`
  - ✓ Assume that an if statement consists of three sub trees,
    - ✓ one for the condition,
    - ✓ one for the consequent statement, and
    - ✓ one for the else statement, if necessary.

# Semantic Analysis

31

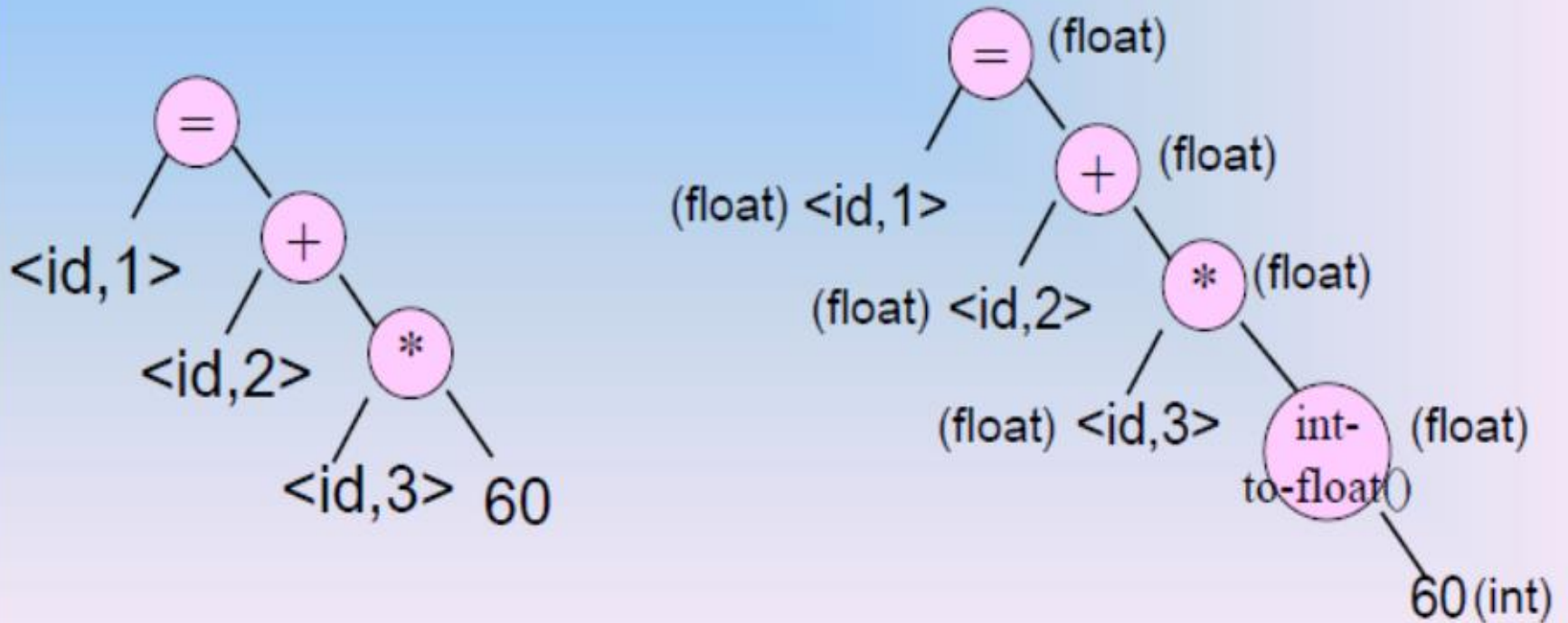
 Checks for “static semantic” errors, e.g., type errors

- ✓ It checks the source programs for semantic errors and gathers type information for the subsequent code generation phase.
- ✓ It uses the syntax tree to identify the operators and Operands of statements.
- ✓ An important component of semantic analysis is **type checking**.
- ✓ It gets input from the syntax analysis as parse tree and checks whether the given syntax is correct or not and
- ✓ It performs type conversion of all the data types into real data types.

# Cont...

32

## Example with before and after




**The most Important** activity in This Phase:

- **Type Checking** - the compiler checks that each operator has operands that are permitted by the source language specification.




# Intermediate Code Generator

33

 In the **code generation** phase, atoms or syntax trees are translated to machine language (binary) instructions, or to assembly language,

 Symbolic addresses (statement labels) are translated to relocatable memory addresses at this time.

 For target machines with several CPU registers, the code generator is responsible for **register allocation**.

# Cont...

34

For example, an ADD atom might be translated to three machine language instructions:

- ✓ Load the first operand into a register,
- ✓ Add the second operand to that register, and
- ✓ Store the result, as shown for the atom (ADD, A, B, Temp):

```
LODR1, A // Load A into reg. 1
```

```
ADDR1, B // Add B to reg. 1
```

```
STOR1, Temp // Store reg. 1 in Temp
```

# Cont...

35

- ✓ Translates from abstract-syntax tree to intermediate code

One possibility is **3-address code**

- ✓ Each statement contains at most 3 operands; in addition to “:=”
- ✓ An “easy” and “universal” format that can be translated into most assembly languages.
- ✓ Here's an example of 3-address code for the abstract-syntax tree shown on the preceding slide.

**t1 = intofloat (60)**

**t2 = id3 \* t1**

**t3 = id2 + t2**

**id1 = t3**

# Code Optimization

36

- ✓ The *global optimization* : the purpose is simply to make the object program more efficient in space and time.
- ✓ It involves examining the sequence of atoms put out by the parser
- ✓ To find redundant or unnecessary instructions or inefficient code.
- ✓ Since it is invoked before the code generator and called machine-independent optimization.

## Cont...

37

Example of global optimization is shown below:

```
for (i=1; i<=100000; i++)  
  x = sqrt (y); // square root function  
  cout << x+i << endl;  
}
```

- ✓ In the global optimization phase, the compiler would move the assignment to x out of the loop in the object program:

```
x = sqrt (y); // loop invariant  
for (i=1; i<=100000; i++)  
  cout << x+i << endl;
```

- ✓ This would eliminate 99,999 unnecessary calls to the sqrt function at run time.

# Cont...

38

## Example 2

```
LODR1,A      // Load A into register 1
ADDR1,B      // Add B to register 1
STOR1,TEMP1  // Store the result in TEMP1*
LODR1,TEMP1  // Load result into reg 1*
ADDR1,C      // Add C to register 1
STOR1,TEMP2  // Store the result in TEMP2
```

Note that some of these instructions (those marked with \* in the comment) can be eliminated without changing the effect of the program, making the object program both smaller and faster:

```
LOD R1,A // Load A into register 1
ADDR1,B // Add B to register 1
ADD R1,C // Add C to register 1
STOR1,TEMP // Store the result in TEMP
```

# Cont...

39

- ✓ It gets the intermediate code as input and produces optimized intermediate code as output.
- ✓ This phase reduces the redundant code and attempts to improve the intermediate code that faster-running machine code will result.
- ✓ During the code optimization, the result of the program is not affected.
- ✓ Improve the efficiency of intermediate code.

# Cont...

40

- ✓ To improve the code generation, the optimization involves
  - ✓ Deduction and removal of dead code (unreachable code).
  - ✓ Calculation of constants in expressions and terms.
  - ✓ Collapsing of repeated expression into temporary string.
  - ✓ Loop unrolling.

```
t1 = intofloat(60)
```

```
t2 = id3 * t1
```

```
t3 = id2 + t2
```

```
id1 = t3
```



```
t2 = id3 * 60.0
```

```
id1 = id2 + t2
```

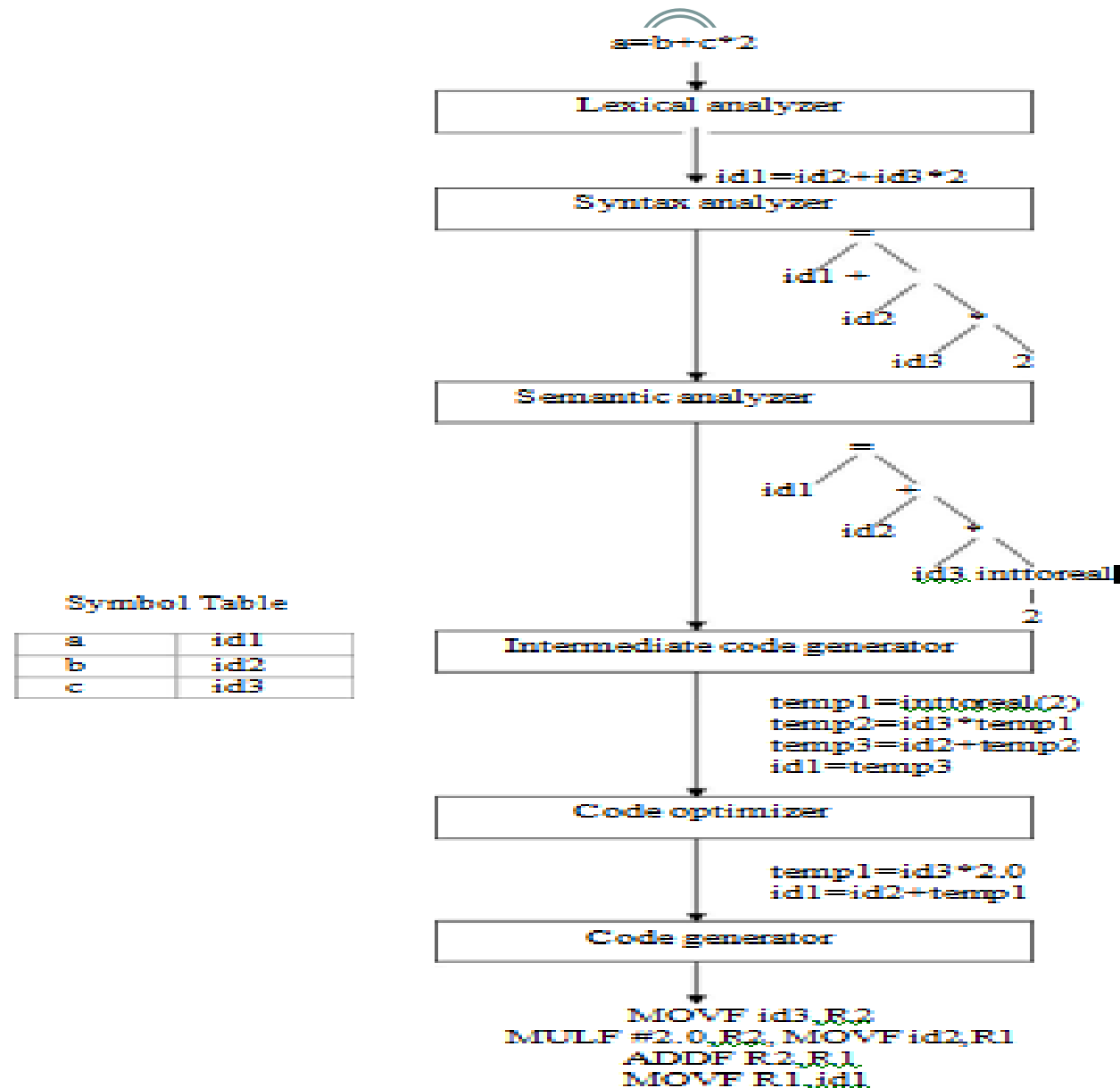


# Code Generation

41

- ✓ A compiler may generate pure machine codes (machine dependent assembly language) directly.
- ✓ Generates object code from (optimized) intermediate code
- The code generation involves
  - ▷ Allocation of register and memory
  - ▷ Generation of correct references
  - ▷ Generation of correct data types
  - ▷ Generation of missing code

# Example $a=b+c*2$



# Symbol Table

43

Symbol table management is a part of the compiler that interacts with several of the phases

- ▶ Identifiers and their values are found in lexical analysis and placed in the symbol table
- ▶ During syntactical and semantic analysis, type and scope information is added
- ▶ During code generation, type information is used to determine what instructions to use
- ▶ During optimization, the “live analysis” may be kept in the symbol table

# Handling Errors

44

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, and the like
- ✓ The compiler may be able to continue with some errors, but other errors may stop the process

# Compiler Construction Tools

45

➤ **Scanner Generators** : Produce Lexical Analyzers

- Example. Lex (Flex)

➤ **Parser Generators** : Produce Syntax Analyzers

Example-YACC (Yet Another Compiler-Compiler).

➤ **Syntax-directed Translation Engines** : Generate intermediate Code

Example.YACC (Bison)

# Cont...

46

- **Automatic Code Generators** : Generate Actual Code
  - i.e. It takes a collection of rules to translate intermediate language into machine language.
  - **Data-Flow Engines** : Support Optimization
- Means: It does code optimization using data-flow analysis, that is, the gathering of information about how values are transmitted from one part of a program to each other part .

# Types of compiler

47

## ❑ Native code compiler

- ✓ A compiler may produce binary output to run /execute on the same computer and operating system. This type of compiler is called as native code compiler.

## ❑ Cross Compiler

- A cross compiler is a compiler that runs on one machine and produce object code for another machine.

## ❑ Bootstrap compiler

- If a compiler has been implemented in its own language. self-hosting compiler.

# Cont...

48

## ❑ **One pass compiler**

- The compilation is done in one pass over the source program, hence the compilation is completed very quickly.
- This is used for the programming language PASCAL, COBOL, FORTAN.

## ❑ **Multi-pass compiler(2 or 3 pass compiler)**

- In this compiler , the compilation is done step by step .
- Each step uses the result of the previous step and it creates another intermediate result.

Example:- gcc , Turbo C++

## ❑ **JIT Compiler**

- This compiler is used for JAVA programming language and Microsoft .NET



# Cont...

49

## ❑ **Source to source compiler**

- ✓ It is a type of compiler that takes a high level language as a input and its output as high level language. Example Open MP(**Open Multi-Processing**)

## ❑ **Parallelizing Compiler**

- ✓ A Compiler capable of compiling a code in parallel computer architecture.

## ❑ **Threaded Code Compiler**

- ✓ The compiler which will simply replace a string (e.g., name of subroutine) by an appropriate binary code.

## ❑ **Incremental Compiler**

- ✓ The compiler which compiles only the changed lines from the source code and update the object code accordingly.

# Cont...

50

## ❑ **Stage Compiler**

A compiler which converts the code into assembly code only.

## ❑ **Just-in-time Compiler**








A compiler which converts the code into machine code after the program starts execution.

## ❑ **Retargetable Compiler**

✓ A compiler that can be easily modified to compile a source code for different CPU architectures

# Operations of Compiler

51

-  These are some operations that are done by the compiler.
-  It breaks source programs into smaller parts.
-  It enables the creation of symbol tables and intermediate representations.
-  It helps in code compilation and error detection.
-  it saves all codes and variables.
-  It analyses the full program and translates it.
-  Convert source code to machine code.

# Advantages of Compiler Design

52

- ✓ **Efficiency:** Compiled programs are generally more efficient than interpreted programs.
- ✓ **Portability:** Once a program is compiled, the resulting machine code can be run on any computer or device that has the appropriate hardware and operating system, making it highly portable.
- ✓ **Error Checking:** Compilers perform comprehensive error checking during the compilation process, which can help catch syntax, semantic, and logical errors in the code before it is run.
- ✓ **Optimizations:** Compilers can make various optimizations to the generated machine code, such as eliminating redundant instructions or rearranging code for better performance.

# Disadvantages of Compiler Design

53

- ✗ **Longer Development Time:** Developing a compiler is a complex and time-consuming process that requires a deep understanding of both the programming language and the target hardware platform.
- ✗ **Debugging Difficulties:** Debugging compiled code can be more difficult than debugging interpreted code because the generated machine code may not be easy to read or understand.
- ✗ **Lack of Interactivity:** Compiled programs are typically less interactive than interpreted programs because they must be compiled before they can be run, which can slow down the development and testing process.
- ✗ **Platform-Specific Code:** If the compiler is designed to generate machine code for a specific hardware platform, the resulting code may not be portable to other platforms.

A spiral-bound notebook with a light beige, textured cover. The notebook is open, showing a blank page with the text "End of Chapter 1" written in a dark brown, serif font. The spiral binding is visible on the left side. The notebook is set against a dark brown background.

**End of Chapter 1**