# Unit – 1
# Introduction to Compiler

Mrs. Ruchi Sharma
ruchi.sharma@bkbiet.ac.in

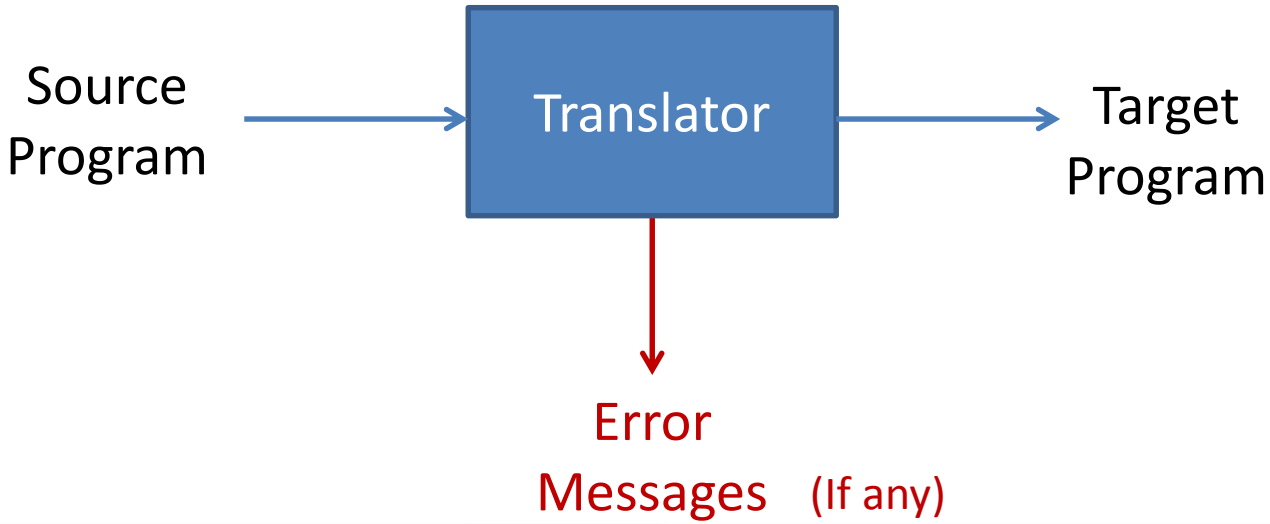
# Topics to be covered

- Translator
- Analysis synthesis model of compilation
- Phases of compiler
- Difference between compiler & interpreter
- Types of compiler
- Context of compiler (Cousins of compiler)
- Pass structure


# Translator

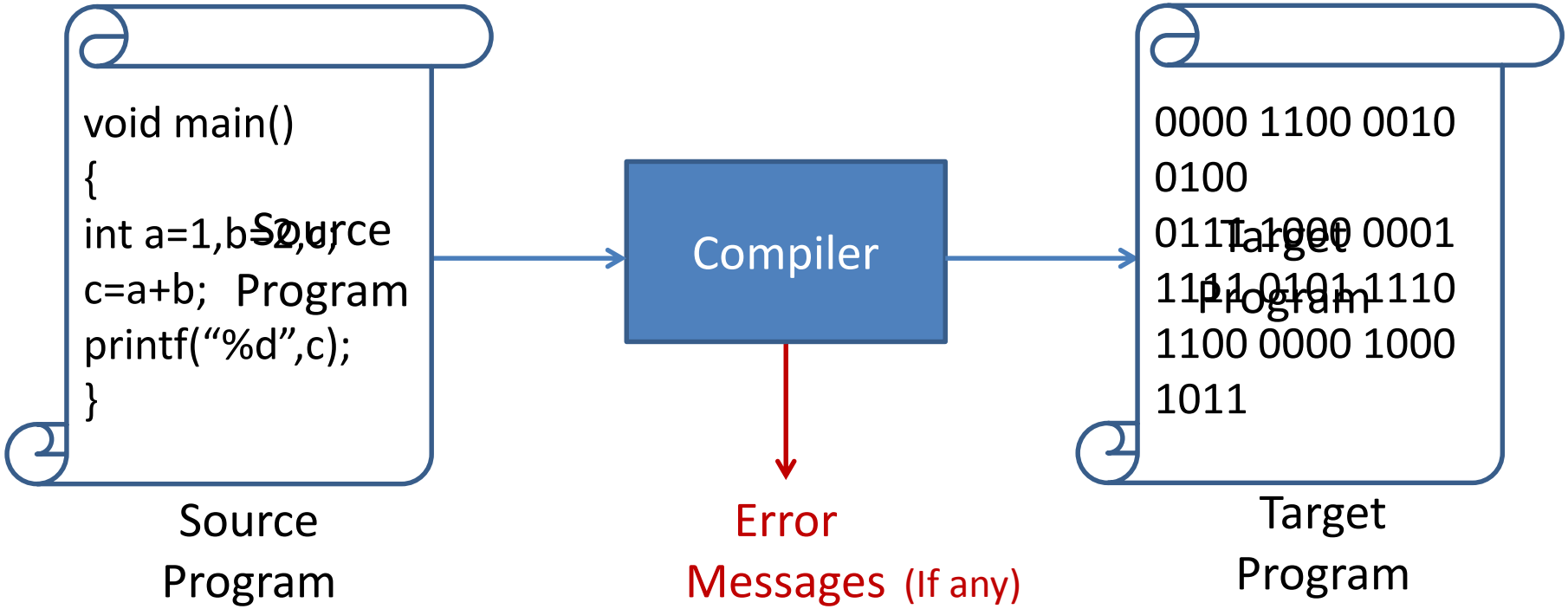
# Translator

- A translator is a program that takes one form of program as input and converts it into another form.
- Types of translators are:
  1. Compiler
  2. Interpreter
  3. Assembler



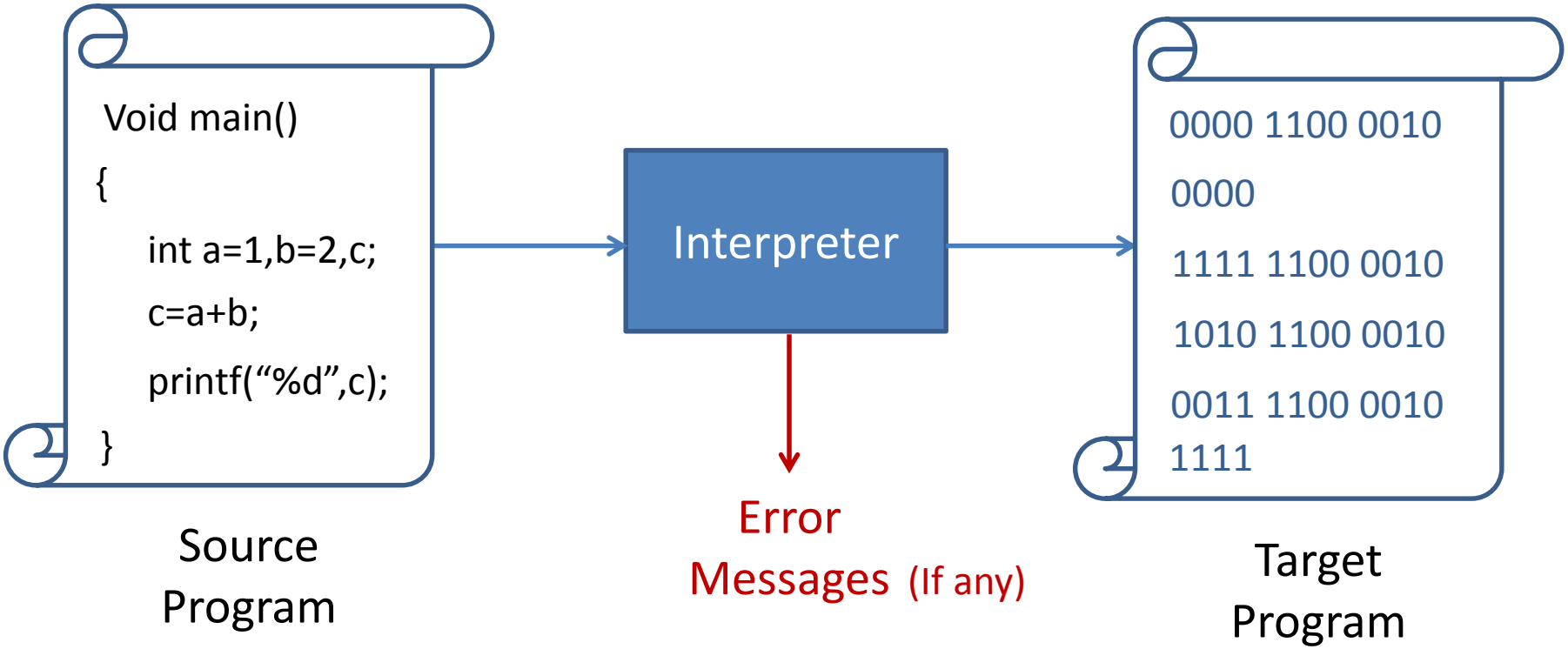
# Compiler

- A compiler is a program that reads a program written in source language and translates it into an equivalent program in target language.



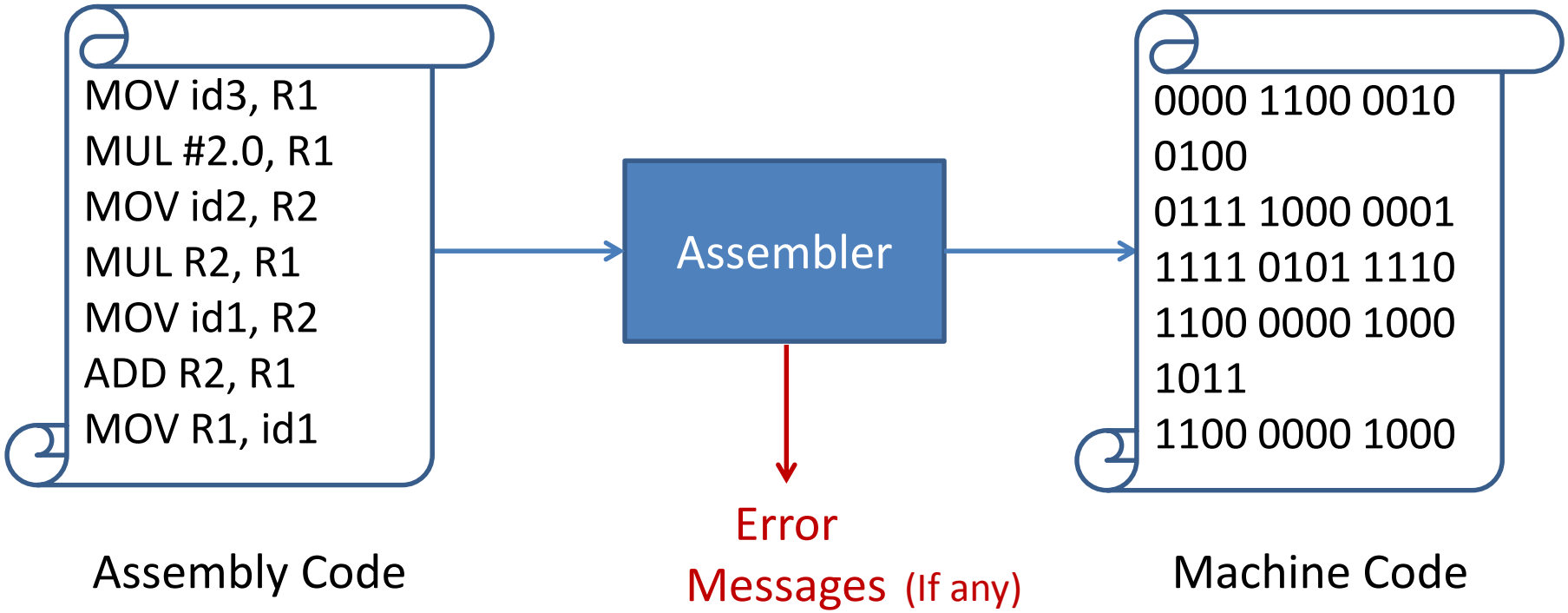
# Interpreter

- Interpreter is also program that reads a program written in source language and translates it into an equivalent program in target language line by line.



# Assembler

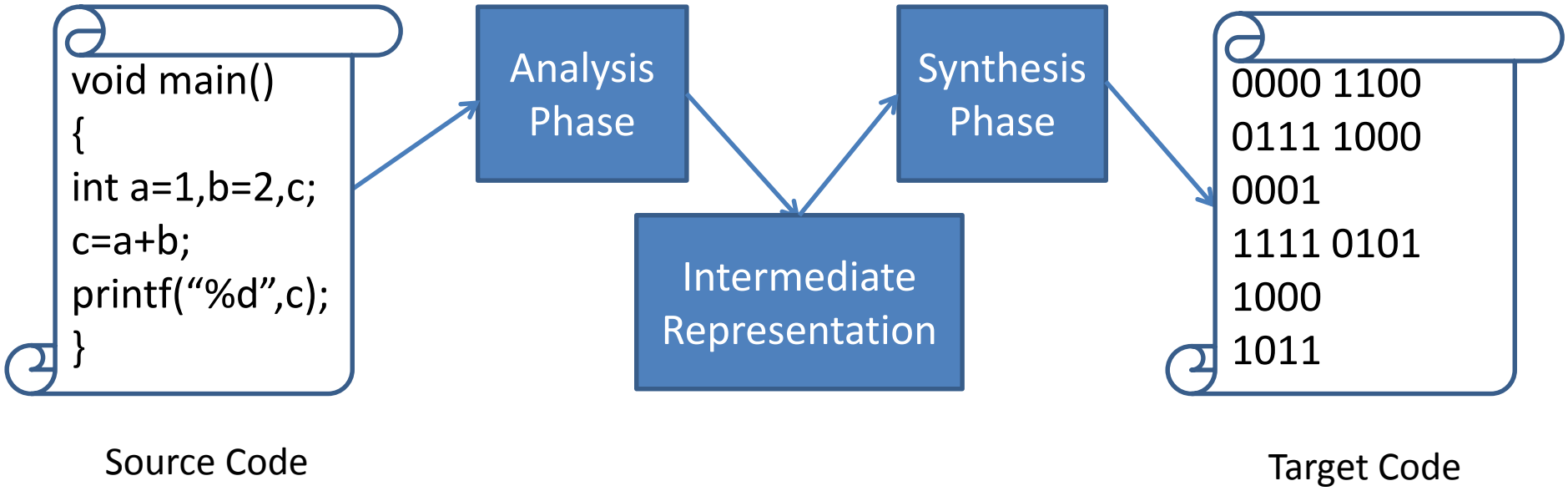
- Assembler is a translator which takes the assembly code as an input and generates the machine code as an output.



# Analysis synthesis model of compilation

# Analysis synthesis model of compilation

- There are two part of compilation.
  1. Analysis Phase
  2. Synthesis Phase



# Analysis phase & Synthesis phase

## Analysis Phase

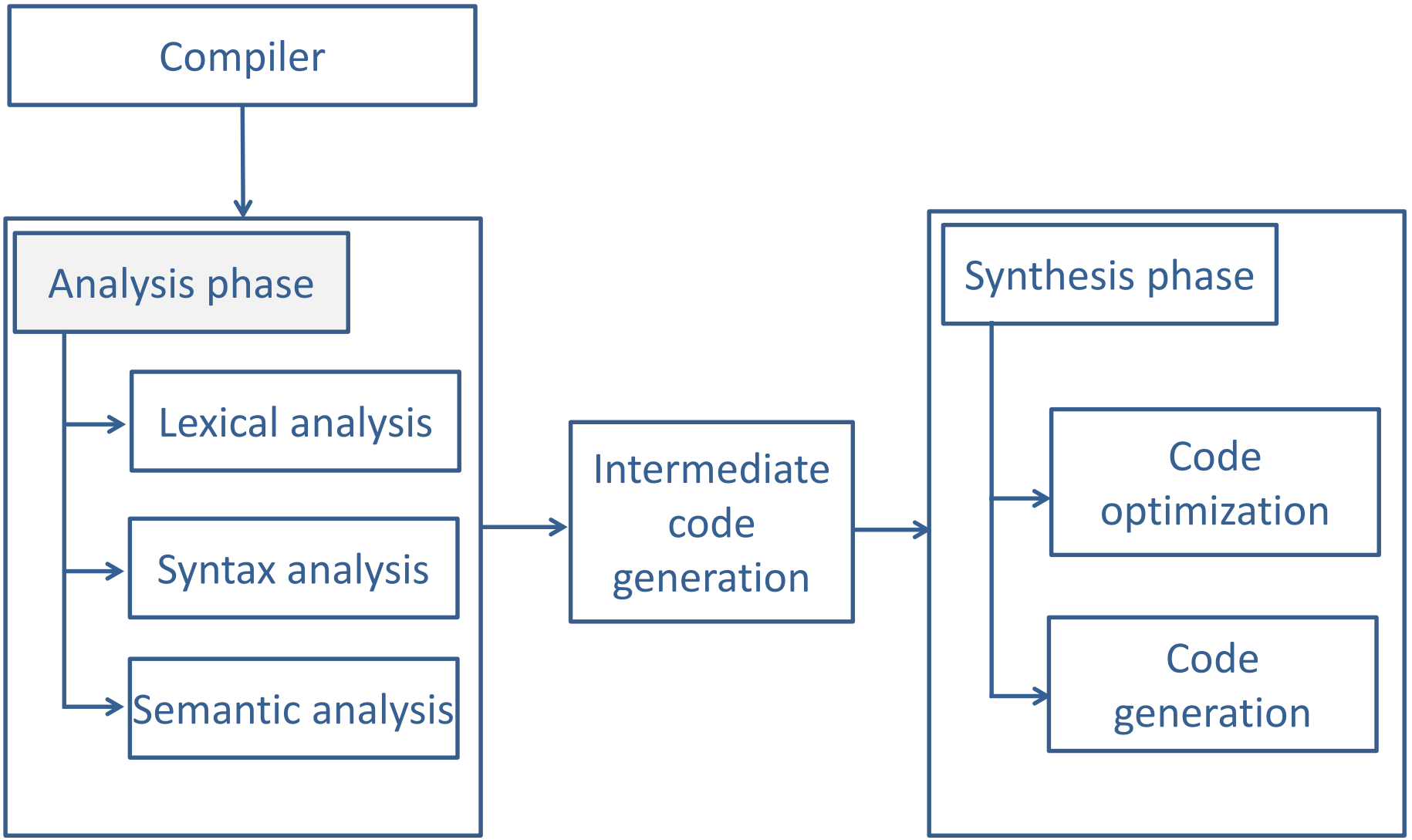
- Analysis part breaks up the source program into constituent pieces and creates an intermediate representation of the source program.
- Analysis phase consist of three sub phases:
  1. Lexical analysis
  2. Syntax analysis
  3. Semantic analysis

## Synthesis Phase

- The synthesis part constructs the desired target program from the intermediate representation.
- Synthesis phase consist of the following sub phases:
  1. Code optimization
  2. Code generation

# Phases of compiler

# Phases of compiler



# Lexical analysis

- Lexical Analysis is also called *linear analysis* or *scanning*.
- Lexical Analyzer divides the given source statement into the *tokens*.
- Ex: Position = initial + rate * 60 would be grouped into the following tokens:

  Position (identifier)

  = (Assignment symbol)

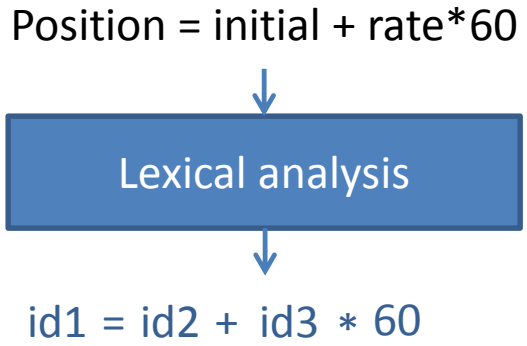
  initial (identifier)

  + (Plus symbol)

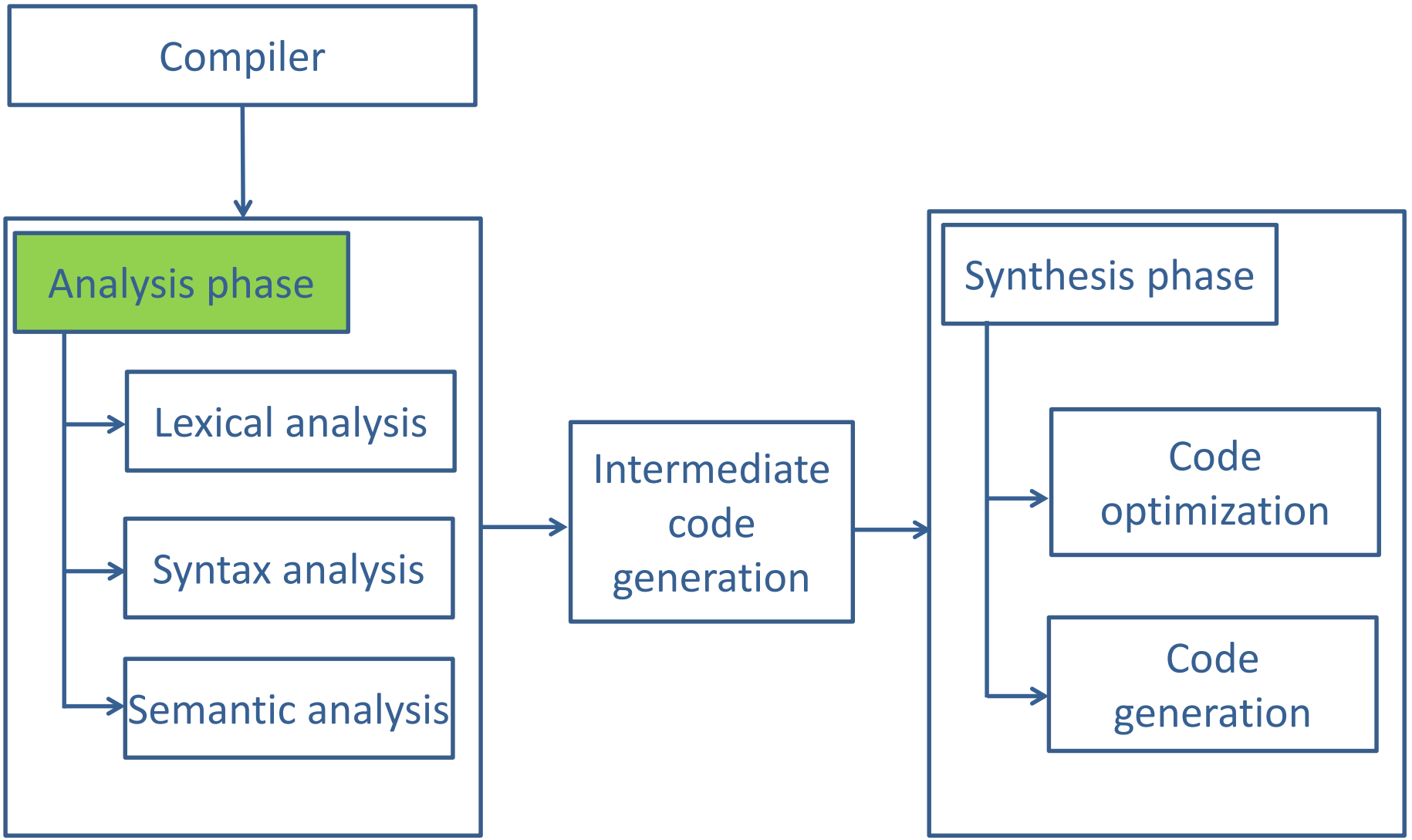
  rate (identifier)

  * (Multiplication symbol)

  60 (Number)

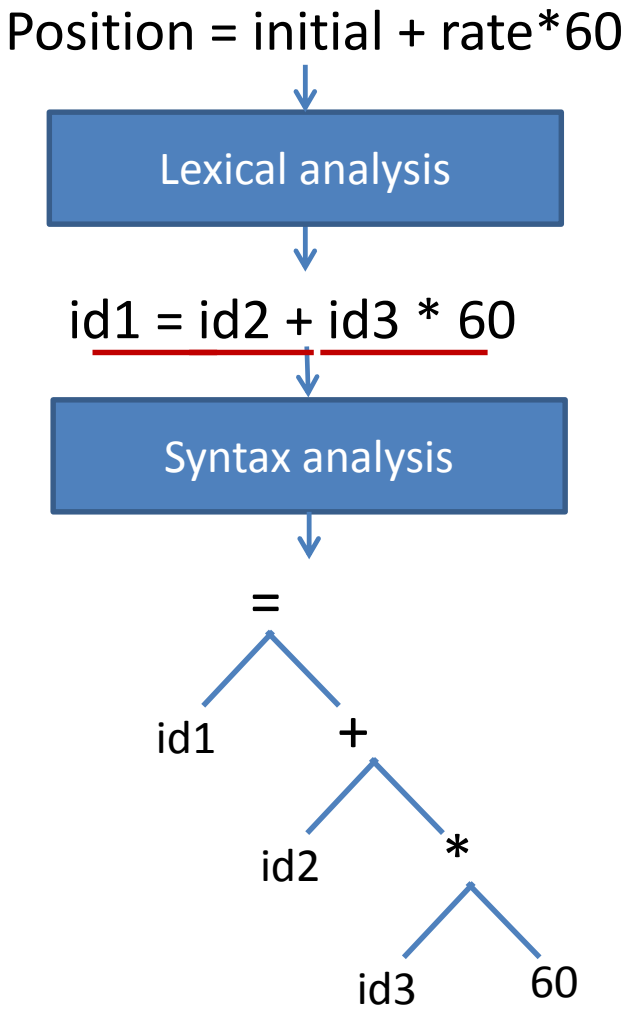


# Phases of compiler

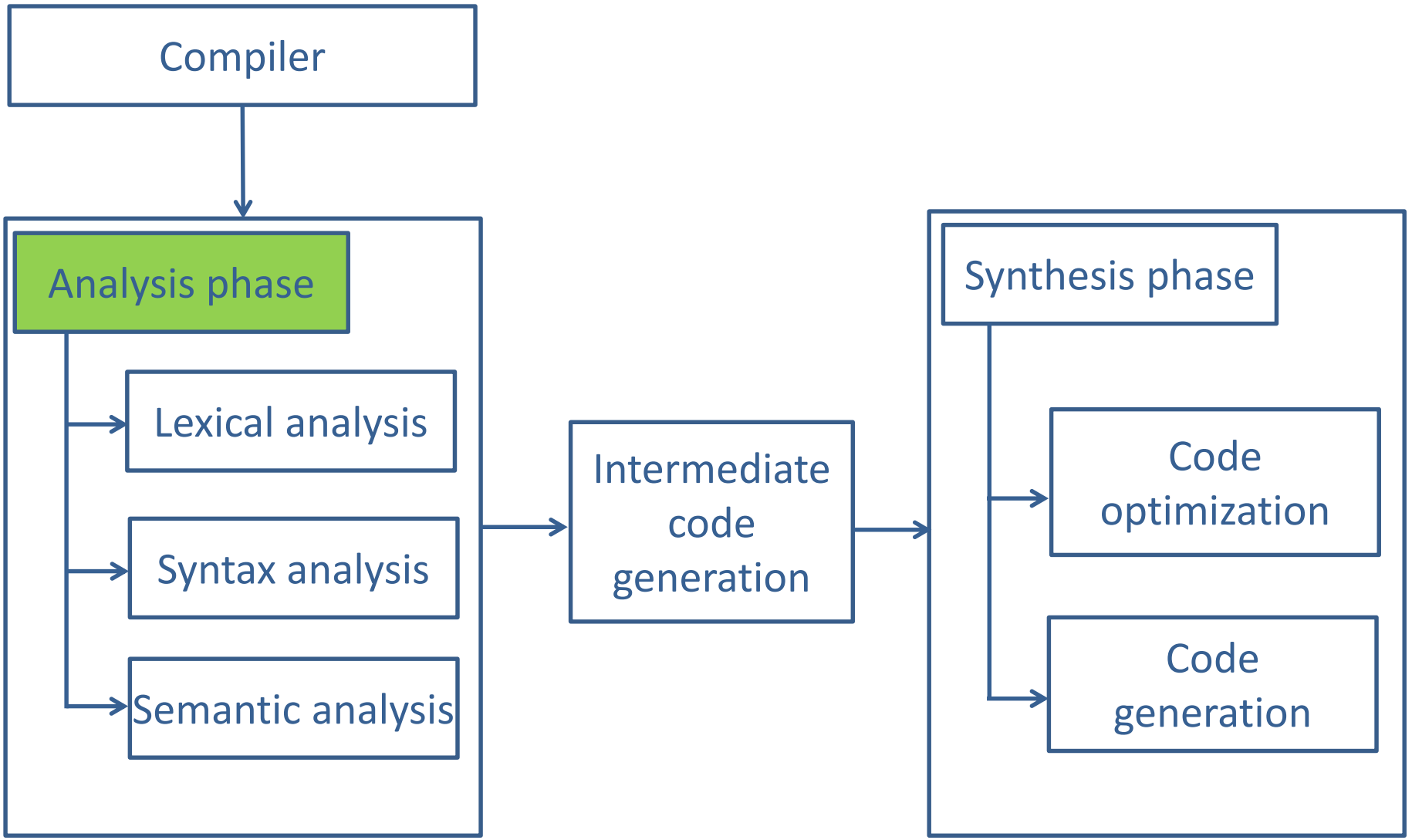


# Syntax analysis

- Syntax Analysis is also called *Parsing* or *Hierarchical Analysis*.
- The syntax analyzer checks each line of the code and spots every tiny mistake.
- If code is error free then syntax analyzer generates the tree.



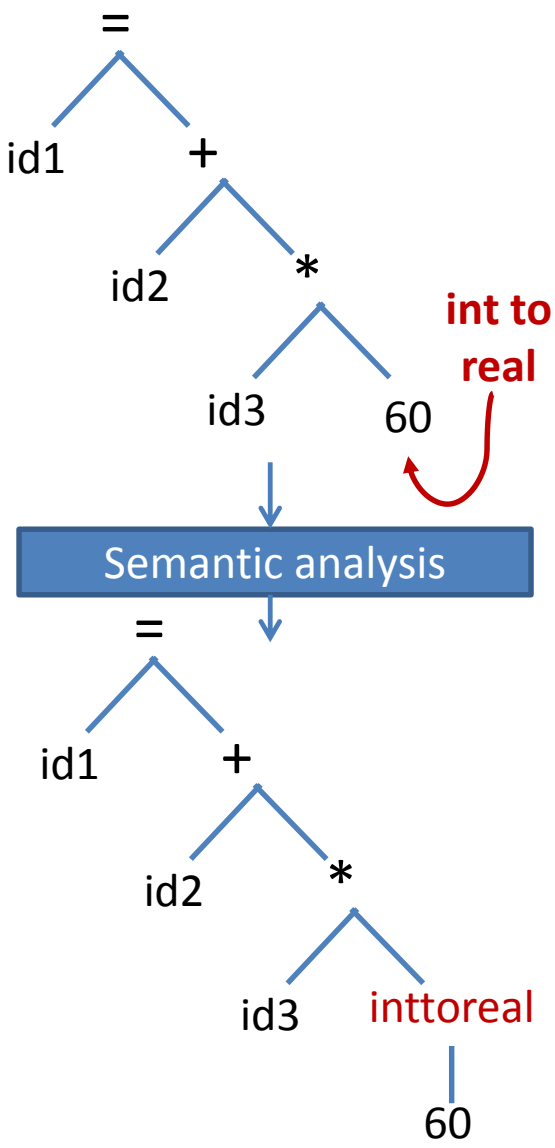
# Phases of compiler



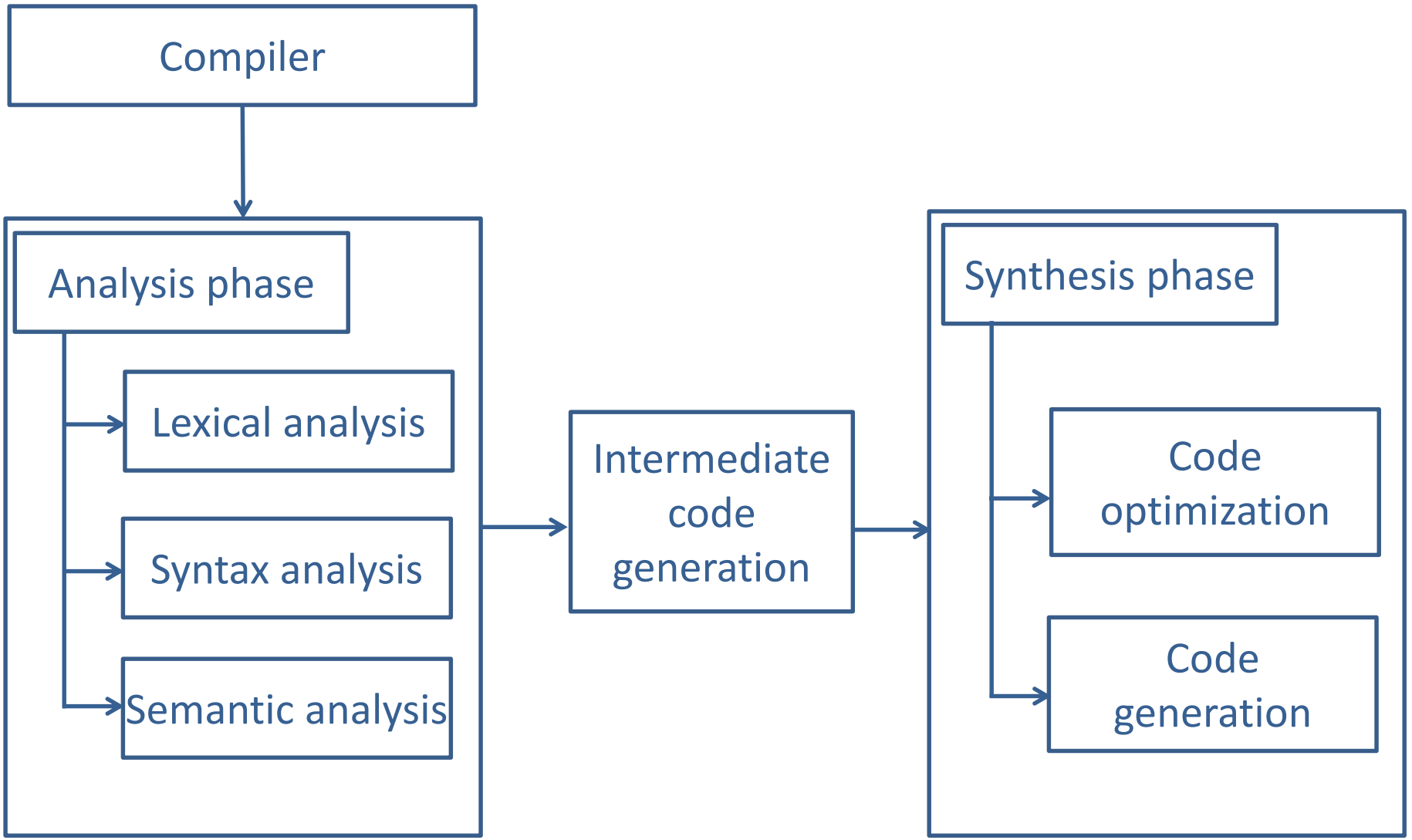
# Semantic analysis

- Semantic analyzer determines the meaning of a source string.
- It performs following operations:
  1. matching of parenthesis in the expression.
  2. Matching of if..else statement.
  3. Performing arithmetic operation that are type compatible.
  4. Checking the scope of operation.

*Note: Consider id1, id2 and id3 are real

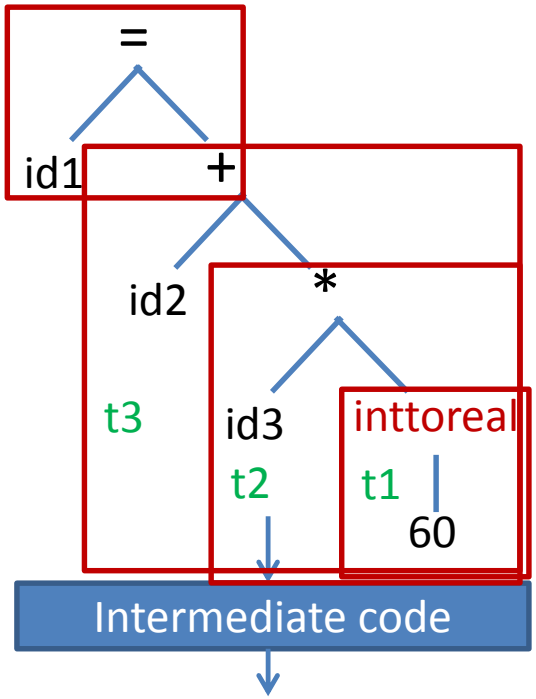


# Phases of compiler



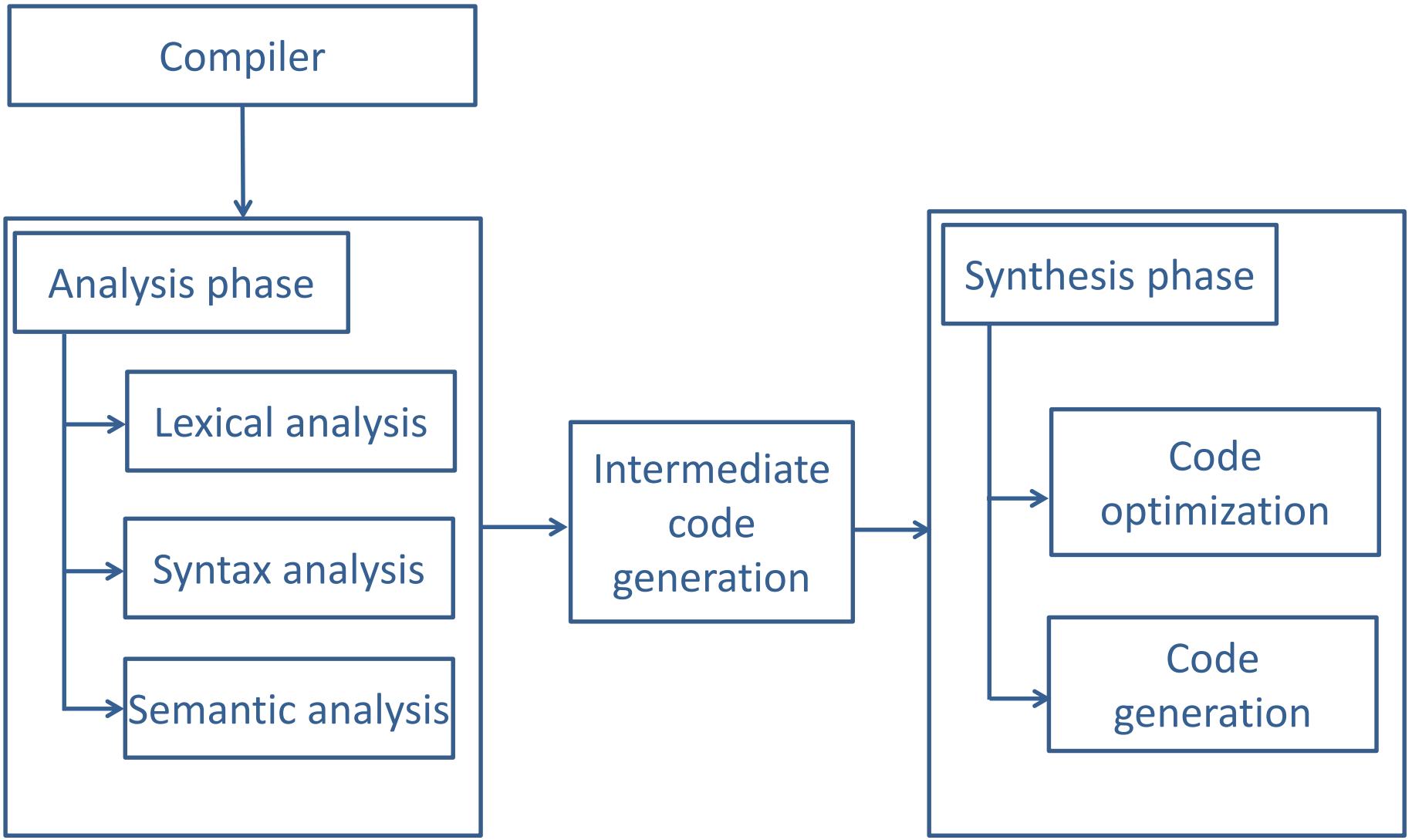
# Intermediate code generator

- Two important properties of intermediate code :
  1. It should be easy to produce.
  2. Easy to translate into target program.
- Intermediate form can be represented using "three address code".
- Three address code consist of a sequence of instruction, each of which has *at most three* operands.



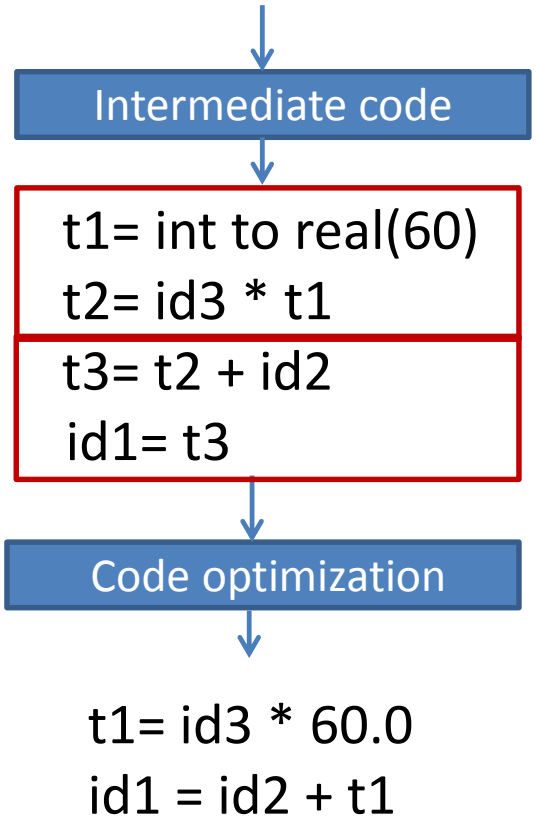

t1= int to real(60)
t2= id3 * t1
t3= t2 + id2
id1= t3

# Phases of compiler

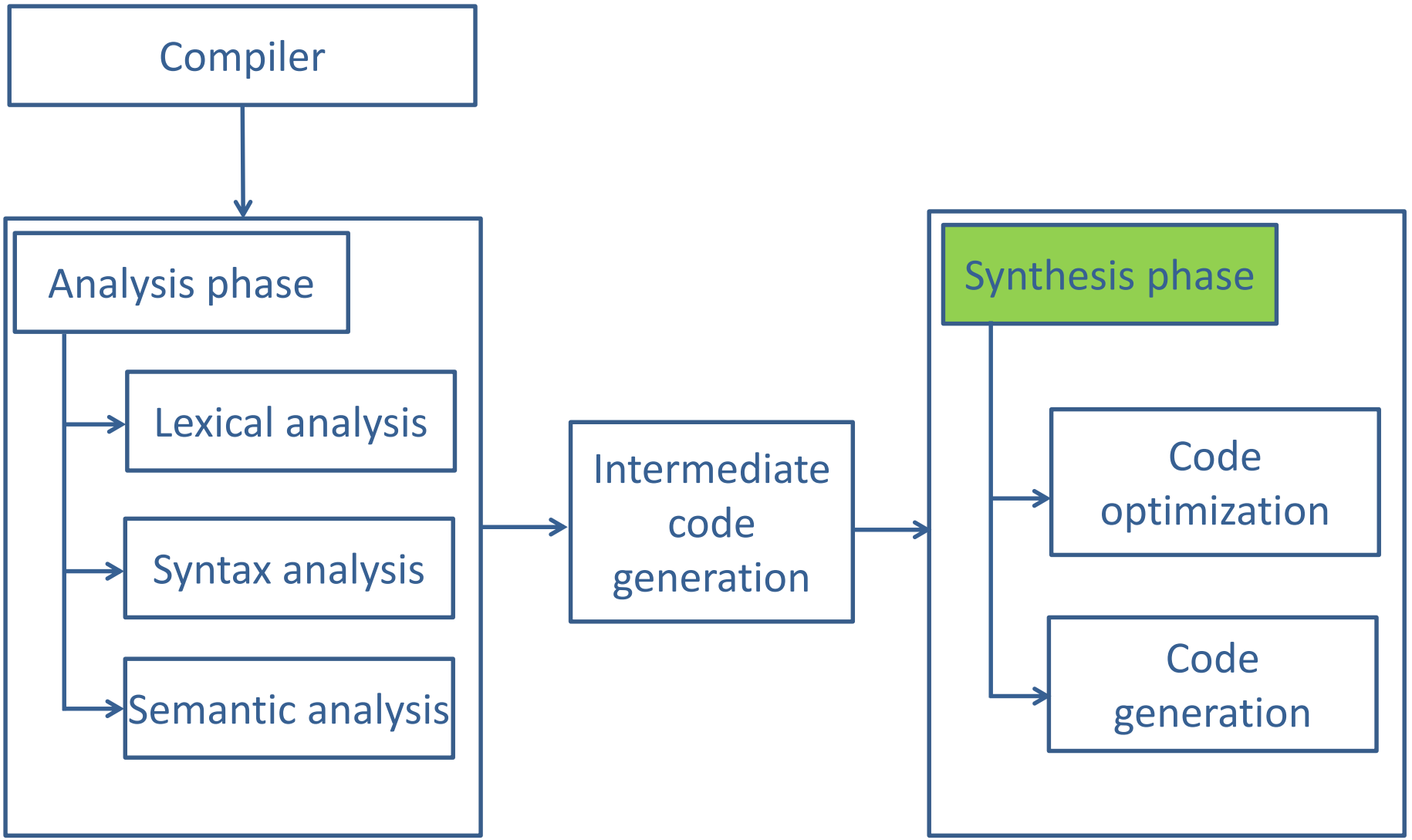


# Code optimization

- It improves the intermediate code.
- This is necessary to have a faster execution of code or less consumption of memory.

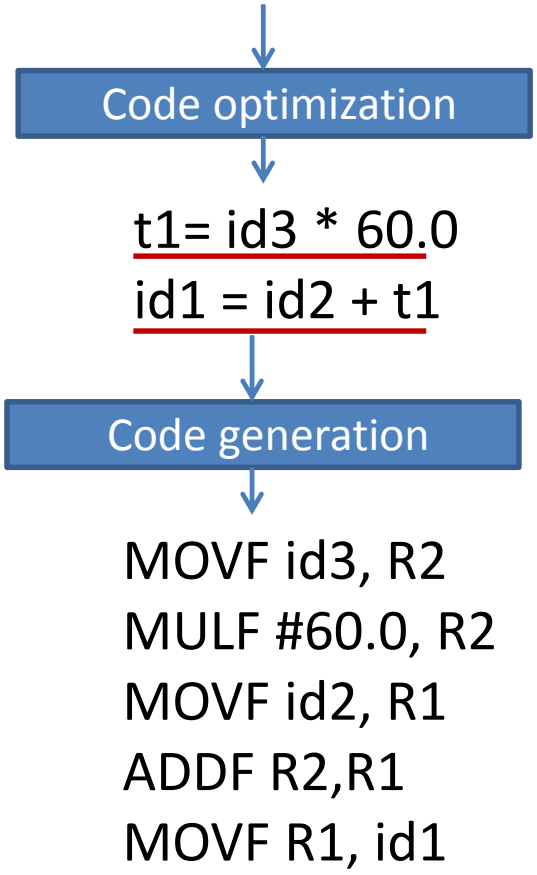


# Phases of compiler

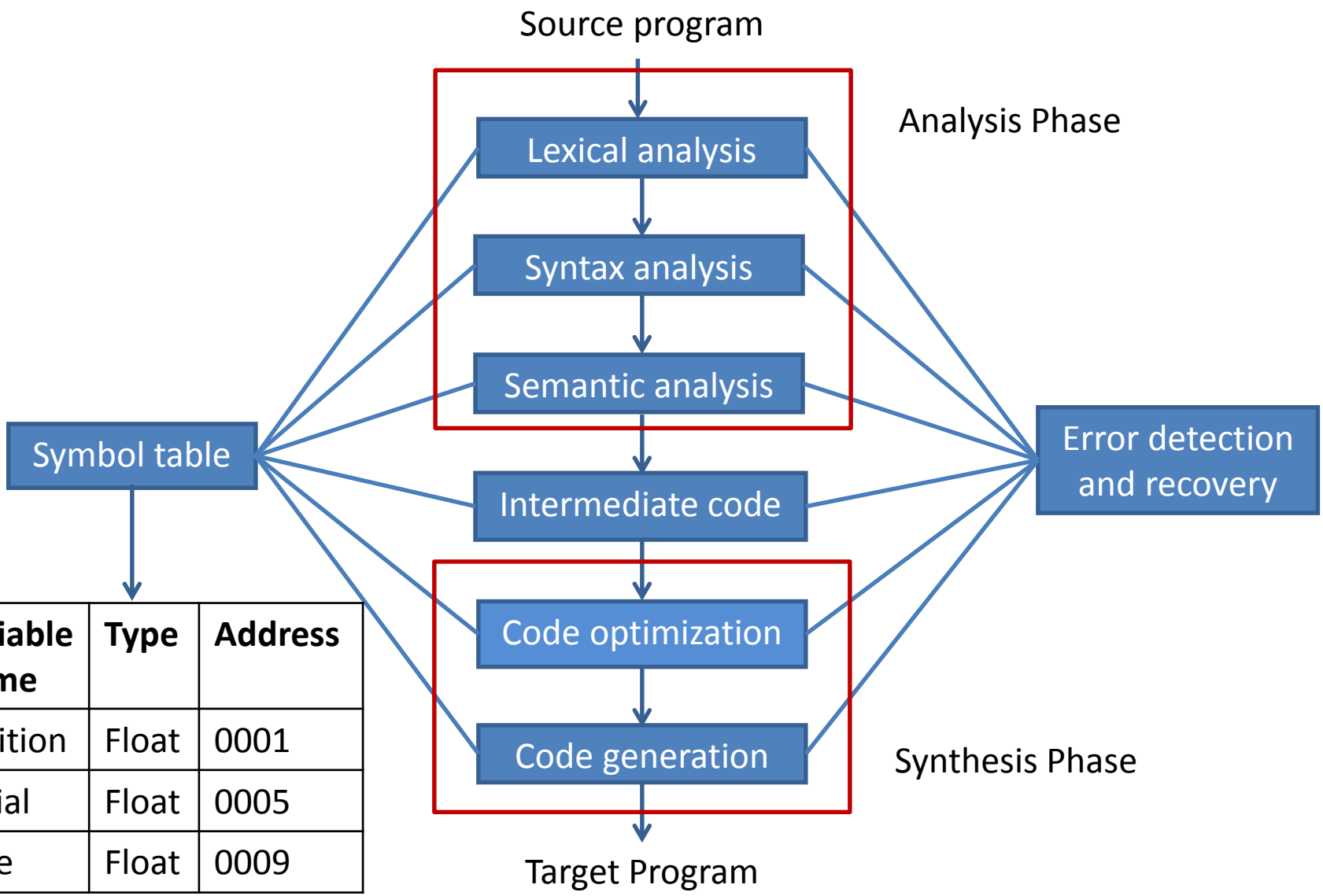


# Code generation

- The intermediate code instructions are translated into sequence of machine instruction.



# Phases of compiler




| Sr. | Variable Name | Type | Address |
|---|---|---|---|
| 1 | Position | Float | 0001 |
| 2 | Initial | Float | 0005 |
| 3 | Rate | Float | 0009 |

# Exercise

- Write output of all the phases of compiler for following statements:
  1. x = b-c*2
  2. I=p*n*r/100

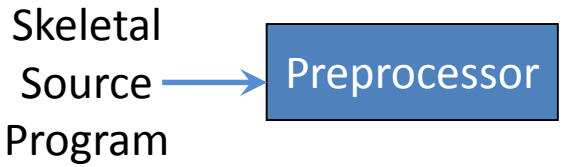
# Difference between compiler & interpreter

| Compiler | Interpreter |
|---|---|
| Scans the entire program and translates it as a whole into machine code. | It translates program's one statement at a time. |
| It generates intermediate code. | It does not generate intermediate code. |
| An error is displayed after entire program is checked. | An error is displayed for every instruction interpreted if any. |
| Memory requirement is more. | Memory requirement is less. |
| Example: C compiler | Example: Basic, Python, Ruby |

- The scanning/lexical analysis phase of a compiler performs the task of **reading the source program** as a file of characters and **dividing up into tokens**
- Usually implemented **as subroutine or co-routine** of parser.
- Front end of compiler.

# Context of Compiler
# (Cousins of compiler)

# Context of compiler (Cousins of compiler)

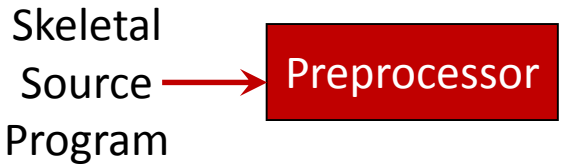



## Preprocessor

It performs the following functions:

1. Macro processing
2. File inclusion
3. Rational preprocessor
4. Language extensions

# Context of compiler (Cousins of compiler)




## Preprocessor

1. **Macro processing**: Allows user to define macros. Macro is shorthand for longer constructs.

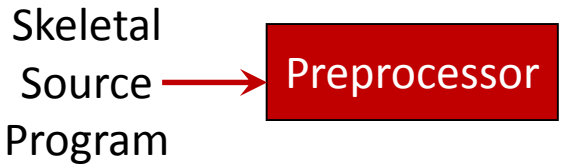
   Ex: #define PI 3.14159265358979323846

2. **File inclusion**: A preprocessor may include the header file into the program.

   Ex: #include<stdio.h>

3. **Rational preprocessor**: It provides built in macro for construct like while statement or if statement.

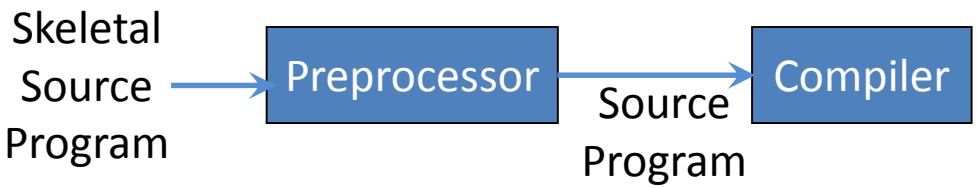
# Context of compiler (Cousins of compiler)




**Preprocessor**

4. **Language extensions**: Add capabilities to the language by using built-in macros.
   - Ex: the language equal is a database query language embedded in C.
   - Statement beginning with ## are taken by preprocessor to be database access statement unrelated to C and translated into procedure call on routines that perform the database access.

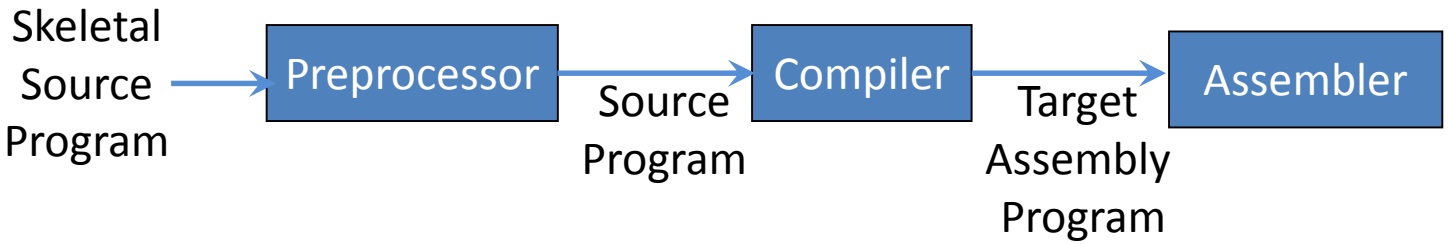
# Context of compiler (Cousins of compiler)




**Compiler**

- A compiler is a program that reads a program written in source language and translates it into an equivalent program in target language.

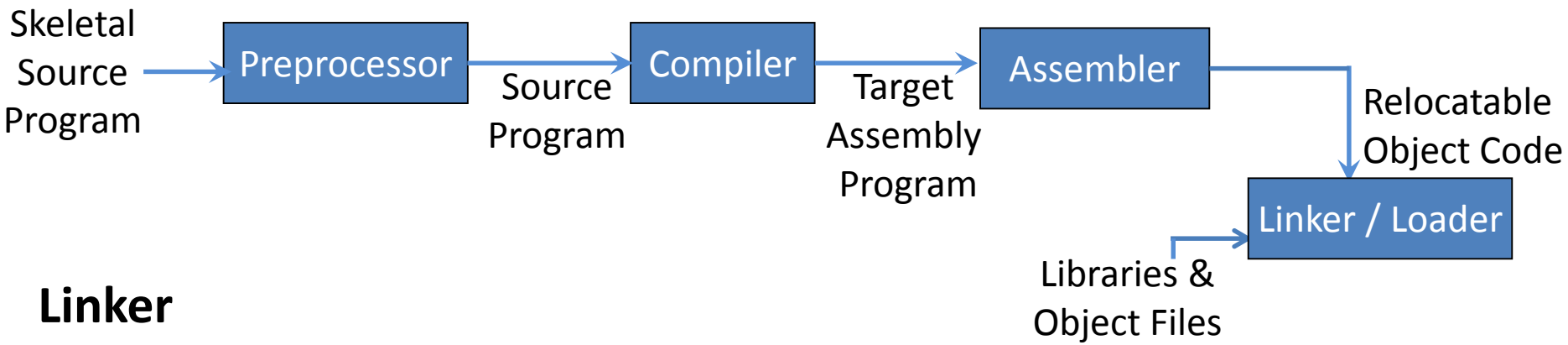
# Context of compiler (Cousins of compiler)




## Assembler

- Assembler is a translator which takes the assembly program (mnemonic) as an input and generates the machine code as an output.

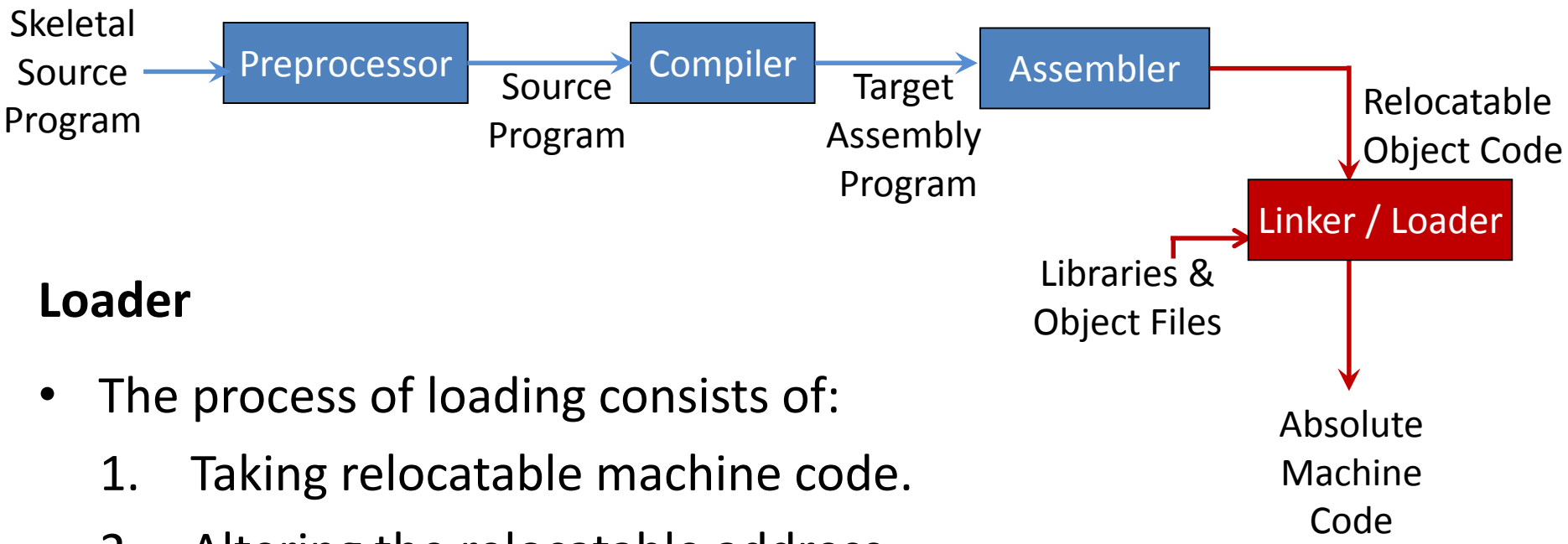
# Context of compiler (Cousins of compiler)




## Linker

- Linker makes a single program from a several files of relocatable machine code.
- These files may have been the result of several different compilation, and one or more library files.

# Context of compiler (Cousins of compiler)




## Loader

- The process of loading consists of:
  1. Taking relocatable machine code.
  2. Altering the relocatable address.
  3. Placing the altered instructions and data in memory at the proper location.

# Front end & back end (Grouping of phases)

**Front end:** Depends primarily on source language and largely independent of the target machine.

It includes following phases:

1. Lexical analysis
2. Syntax analysis
3. Semantic analysis
4. Intermediate code generation
5. Creation of symbol table

**Back end:** Depends on target machine and do not depend on source program.

It includes following phases:

1. Code optimization
2. Code generation phase
3. Error handling and symbol table operation

# Pass structure

# Pass structure

- One complete scan of a source program is called pass.
- Pass includes reading an input file and writing to the output file.
- In a single pass compiler analysis of source statement is immediately followed by synthesis of equivalent target statement.
- While in a two pass compiler intermediate code is generated between analysis and synthesis phase.
- It is difficult to compile the source program into single pass due to: *forward reference*

# Pass structure

**Forward reference:** A forward reference of a program entity is a reference to the entity which precedes its definition in the program.

- This problem can be solved by postponing the generation of target code until more information concerning the entity becomes available.
- It leads to multi pass model of compilation.
- In Pass I: Perform analysis of the source program and note relevant information.
- In Pass II: Generate target code using information noted in pass I.

# Effect of reducing the number of passes

- It is desirable to have a few passes, because it takes time to read and write intermediate file.
- If we group several phases into one pass then memory requirement may be large.

# Types of compiler

# Types of compiler

1. One pass compiler
   - It is a type of compiler that compiles whole process in one-pass.
2. Two pass compiler
   - It is a type of compiler that compiles whole process in two-pass.
   - It generates intermediate code.
3. Incremental compiler
   - The compiler which compiles only the changed line from the source code and update the object code.
4. Native code compiler
   - The compiler used to compile a source code for a same type of platform only.
5. Cross compiler
   - The compiler used to compile a source code for a different kinds platform.

# End of PART-1(Unit-I)