Unit: 1

Overview of the Compiler & it’s Structure

Topics to be covered

- Language Processor
- Translator
- Analysis synthesis model of compilation
- Phases of compiler
- Grouping of the Phases
- Difference between compiler & interpreter
- Context of compiler (Cousins of compiler)
- Pass structure
- Types of compiler
- The Science of building a compiler

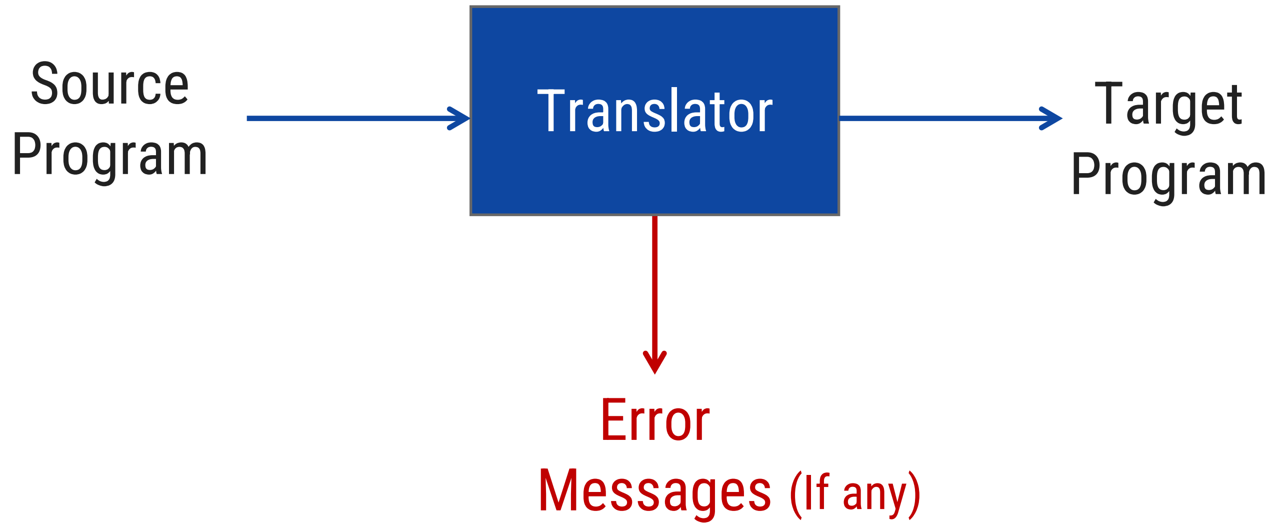
Language processor

- Language processor is a software which bridges semantic gap.
- A **language processor** is a software program designed or used to perform tasks such as processing program code to machine code.

Translator

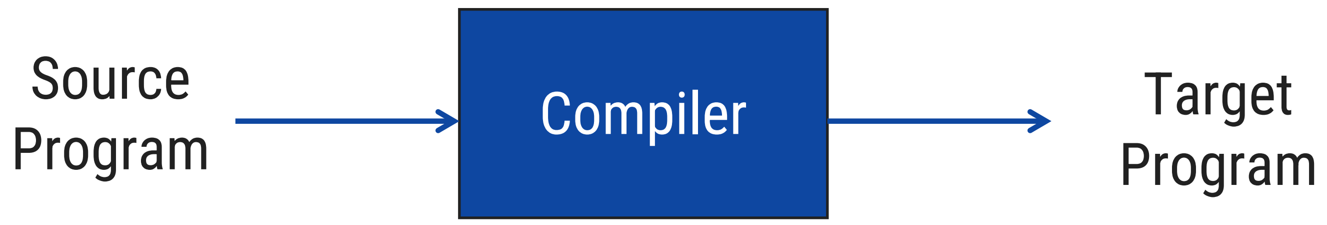
Translator

- 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



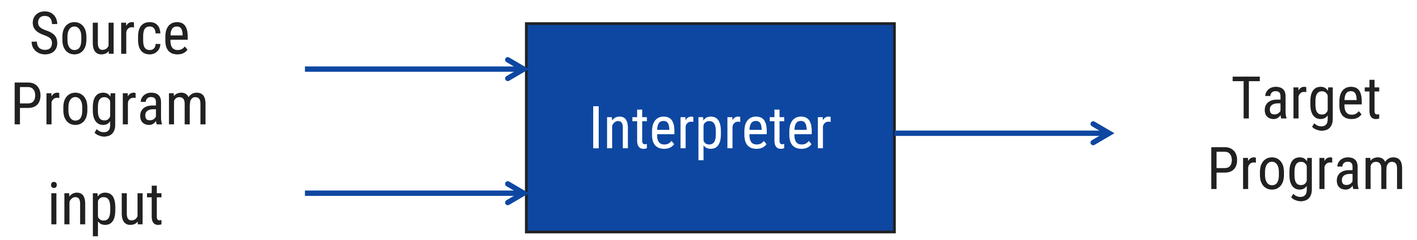
Compiler

- A compiler is a program that read a program written in one language – the Source language and translates it into an equivalent program in another language – the target language.
- An important role of the compiler is to report any errors in the source program that it detects during the translation process.



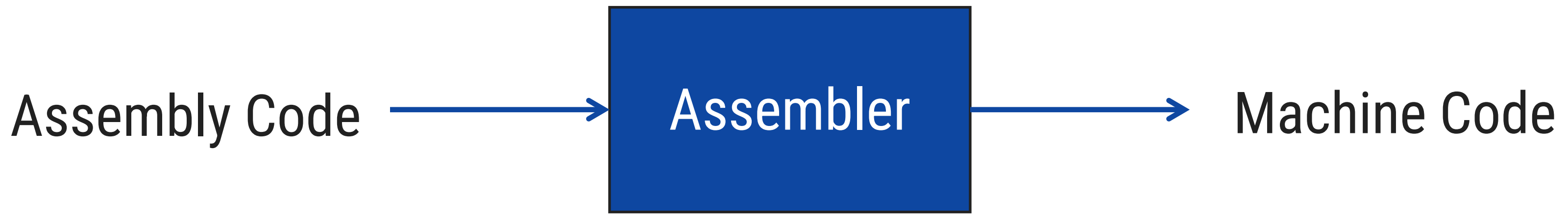
Interpreter

- 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.
- The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs .
- An interpreter, however, can usually give better error diagnostics than a compiler, because it executes the source program statement by statement.



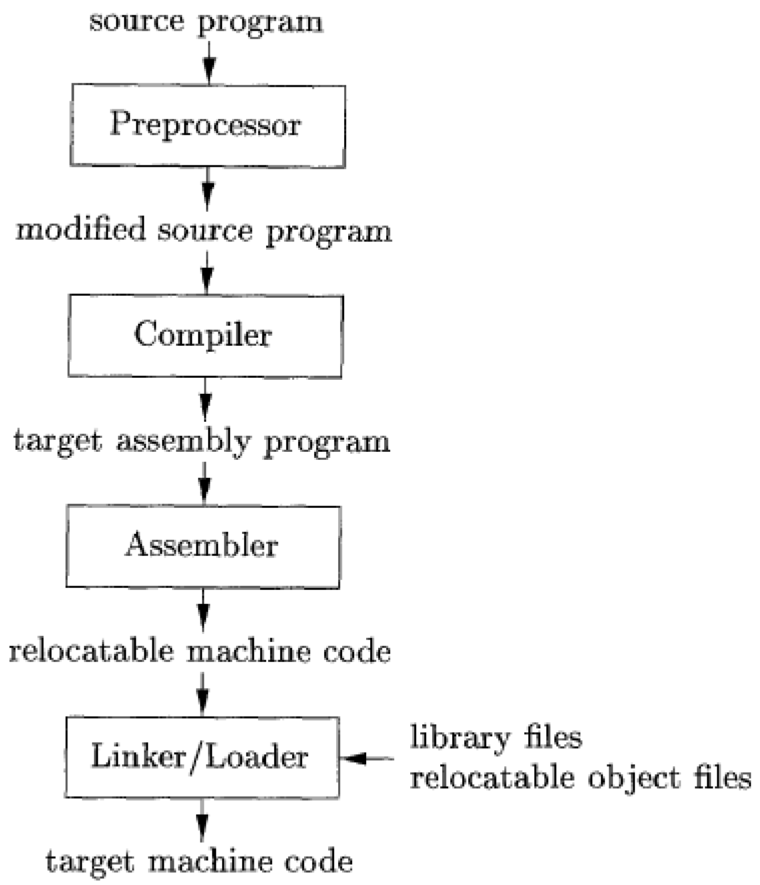
Assembler

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



Context of Compiler (Cousins of compiler)

A language – processing system

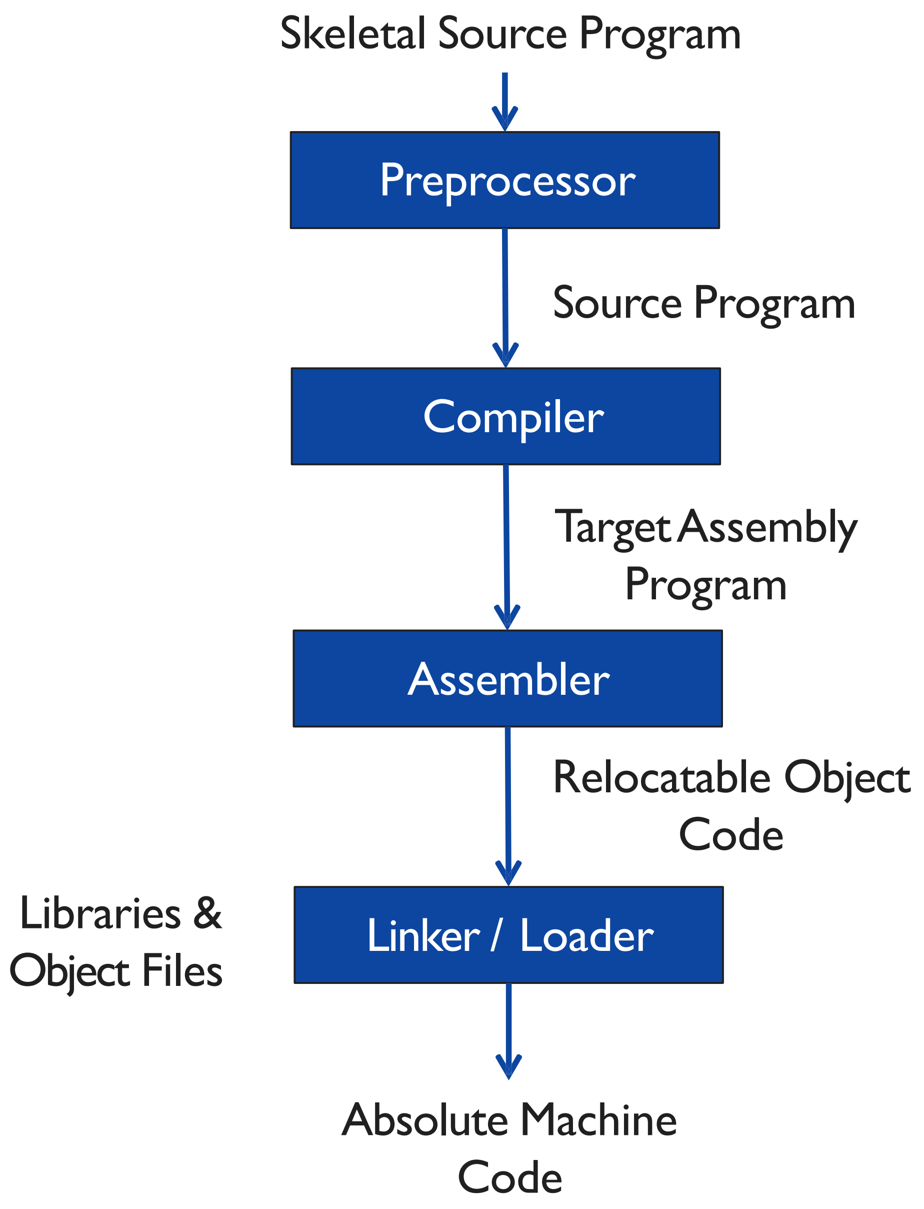


A language – processing system

- 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**.
- The preprocessor may also expand shorthands, called macros, into source language statements.
- The modified source program is then fed to a compiler.
- The **compiler** may produce an assembly-language program as its output, because assembly language is easier to produce as output and is easier to debug.
- The assembly language is then processed by a program called an **assembler** that produces relocatable machine code as its output.
- Large programs are often compiled in pieces, so the relocatable machine code may have to be linked together with other relocatable object files and library files into the code that actually runs on the machine.
- The **linker** resolves external memory addresses, where the code in one file may refer to a location in another file.
- The **loader** then puts together all of the executable object files into memory for execution.

Context of compiler (Cousins of compiler)

- In addition to compiler, many other system programs are required to generate absolute machine code.
- These system programs are:
    - ➥ Preprocessor
    - ➥ Assembler
    - ➥ Linker
    - ➥ Loader

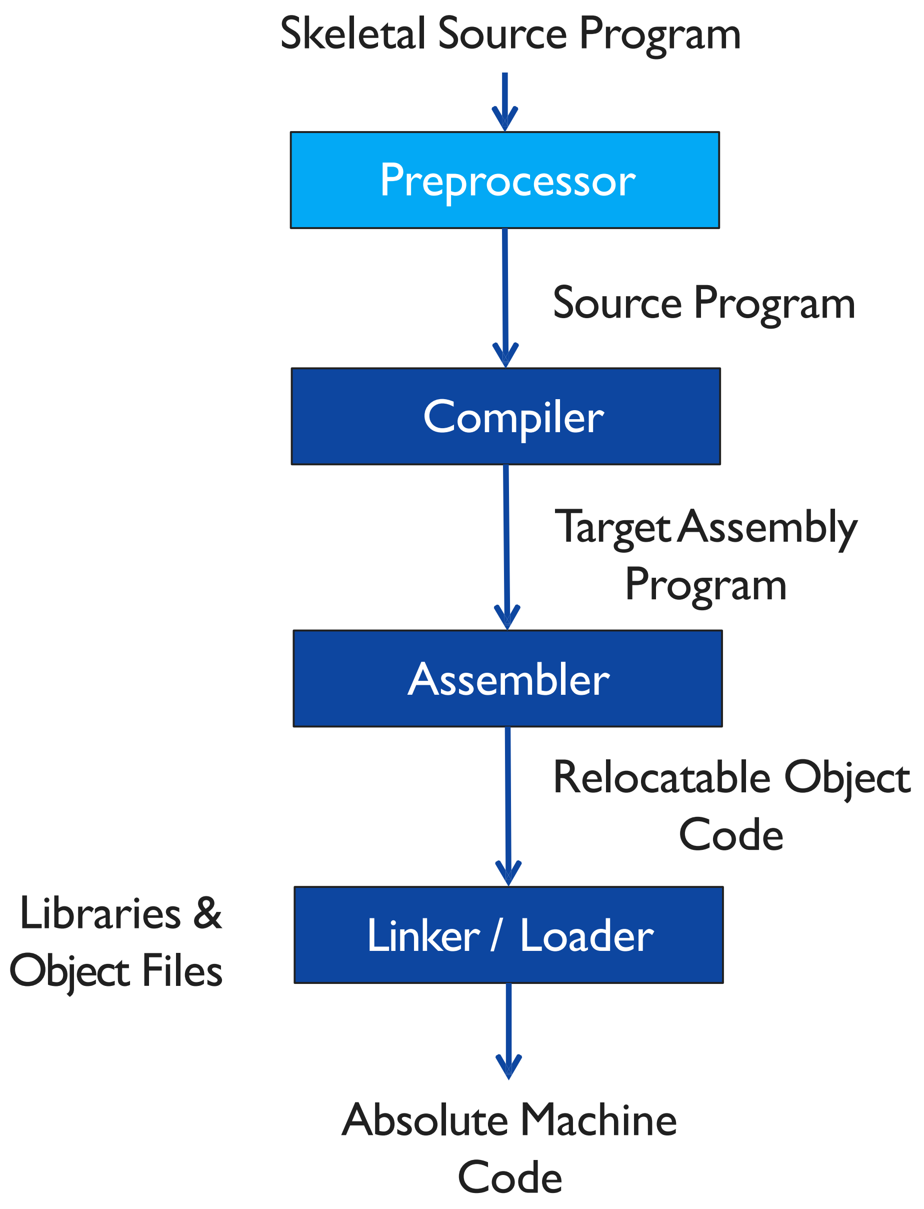


Context of compiler (Cousins of compiler)

Preprocessor

- **Some of the task performed by preprocessor:**

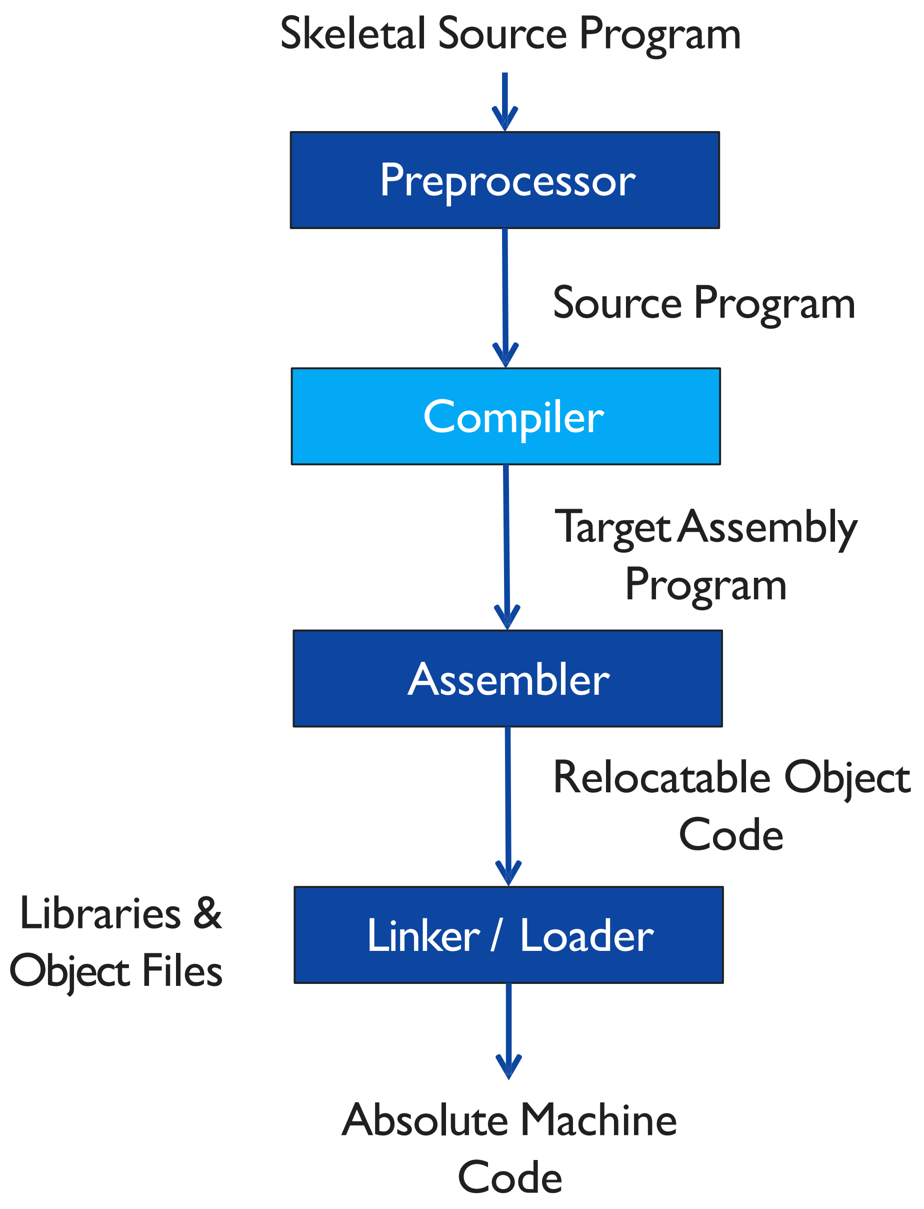
1. **Macro processing:** Allows user to define macros. Ex: #define PI 3.14159265358979323846
2. **File inclusion:** A preprocessor may include the header file into the program. Ex: #include<stdio.h>
3. **Rational preprocessor:** It provides built in macro for construct like while statement or if statement.
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

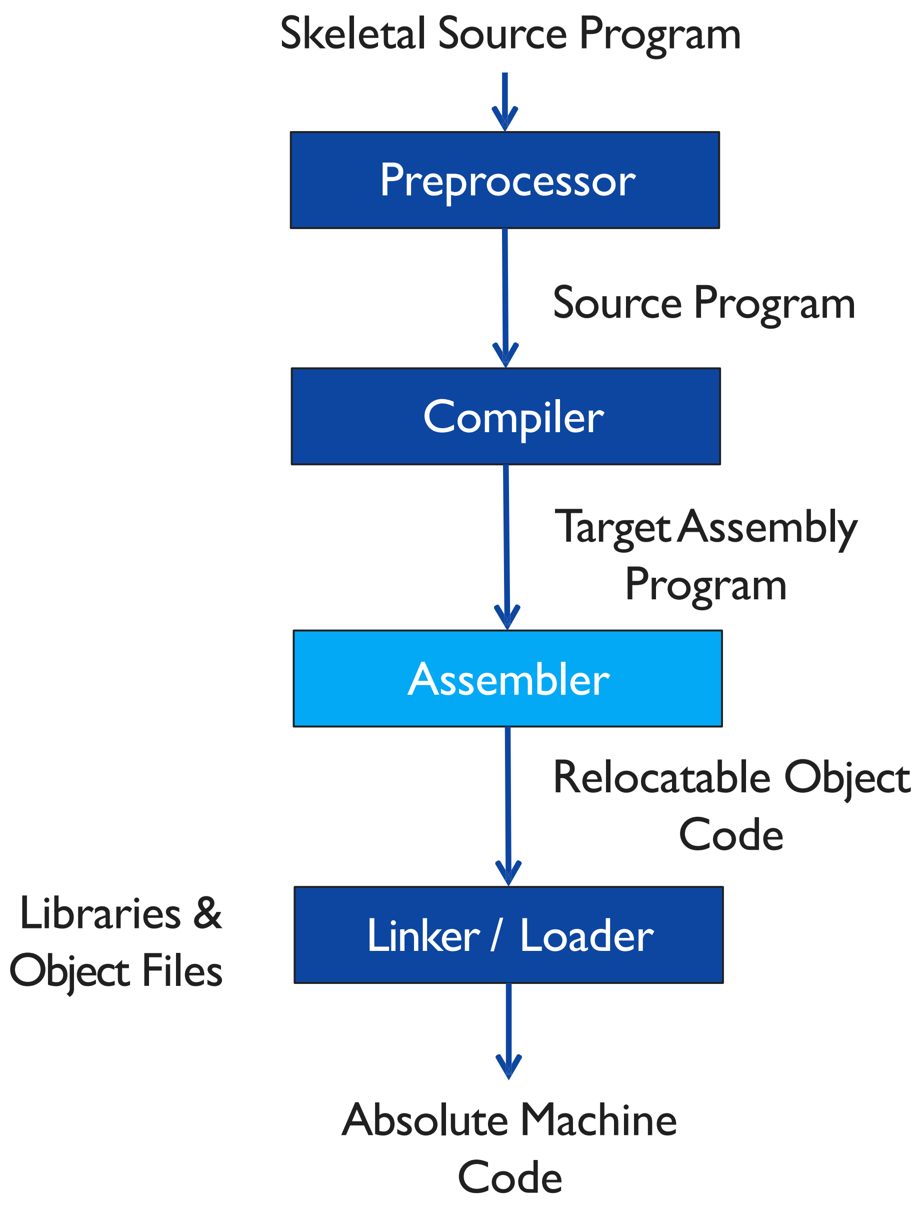
- A compiler is a program that reads a program written in source language and translates it into an equivalent program in target language.



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.



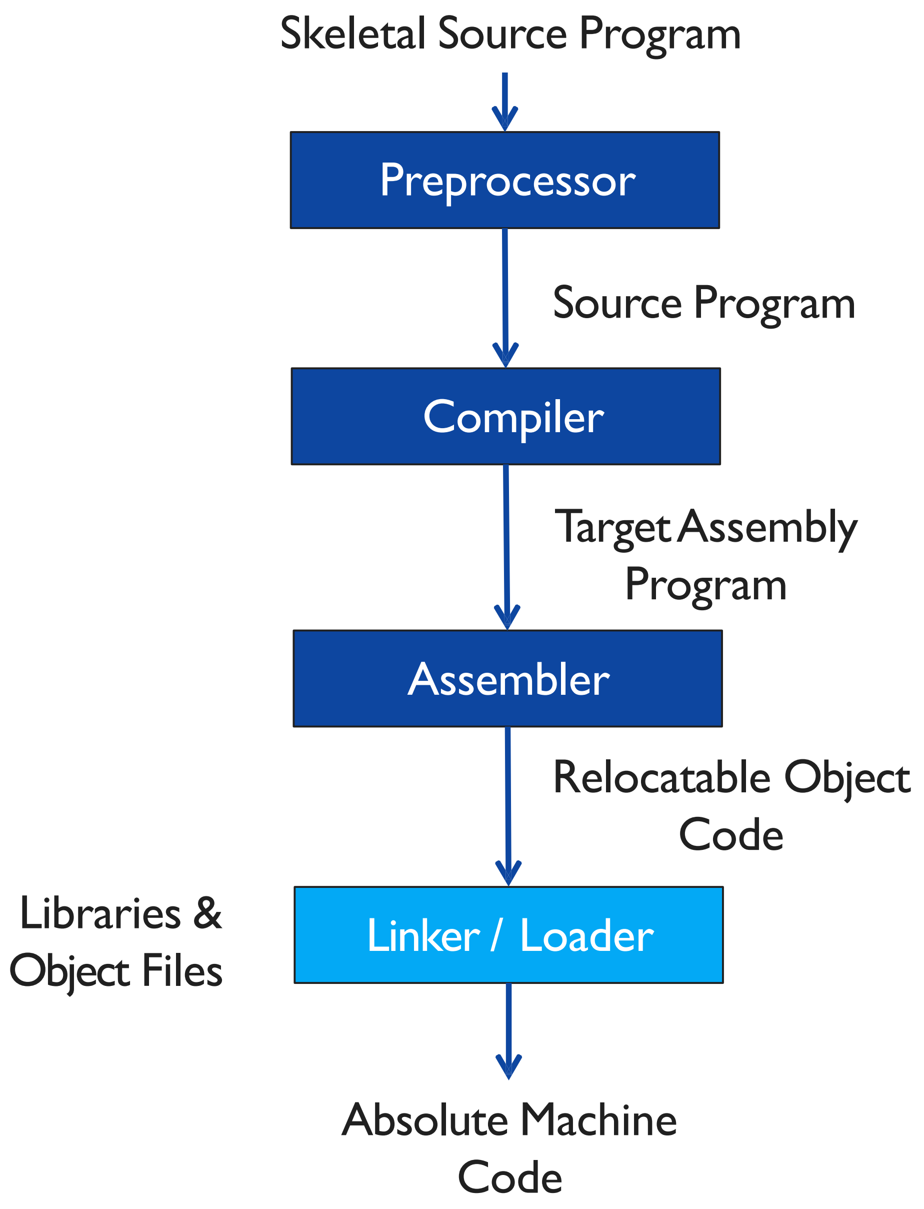
Context of compiler (Cousins of compiler)

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.

Loader

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



Analysis Synthesis model of compilation

Analysis synthesis model of compilation

- **There are two parts of compilation.**

1. Analysis Phase
2. Synthesis Phase

Analysis phase & Synthesis phase

Analysis Phase

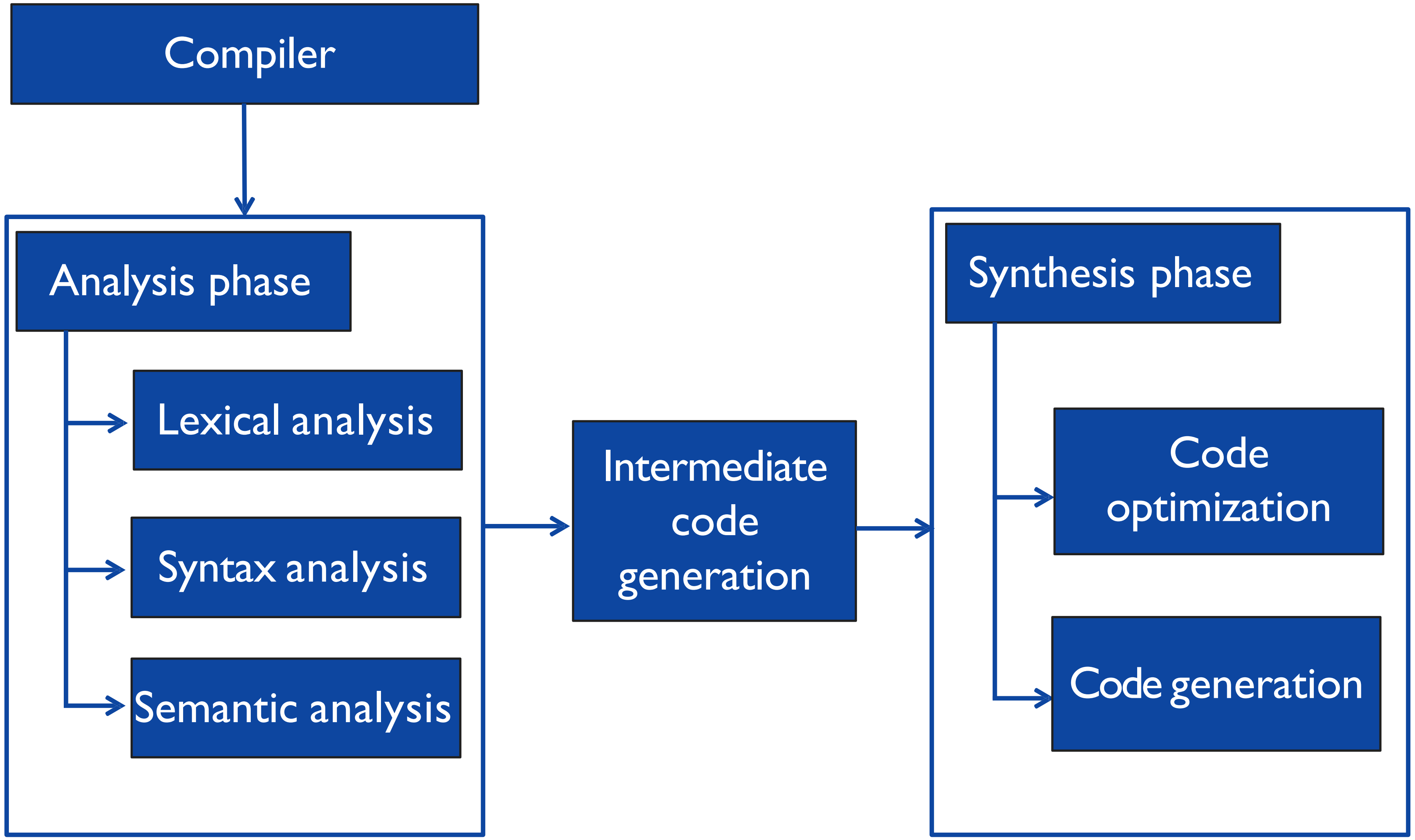
- Analysis part breaks up the source program into constituent pieces and creates an intermediate representation of the source program.
- Analysis phase consists of three sub phases:
  1. Lexical analysis
  2. Syntax analysis
  3. Semantic analysis

Synthesis Phase

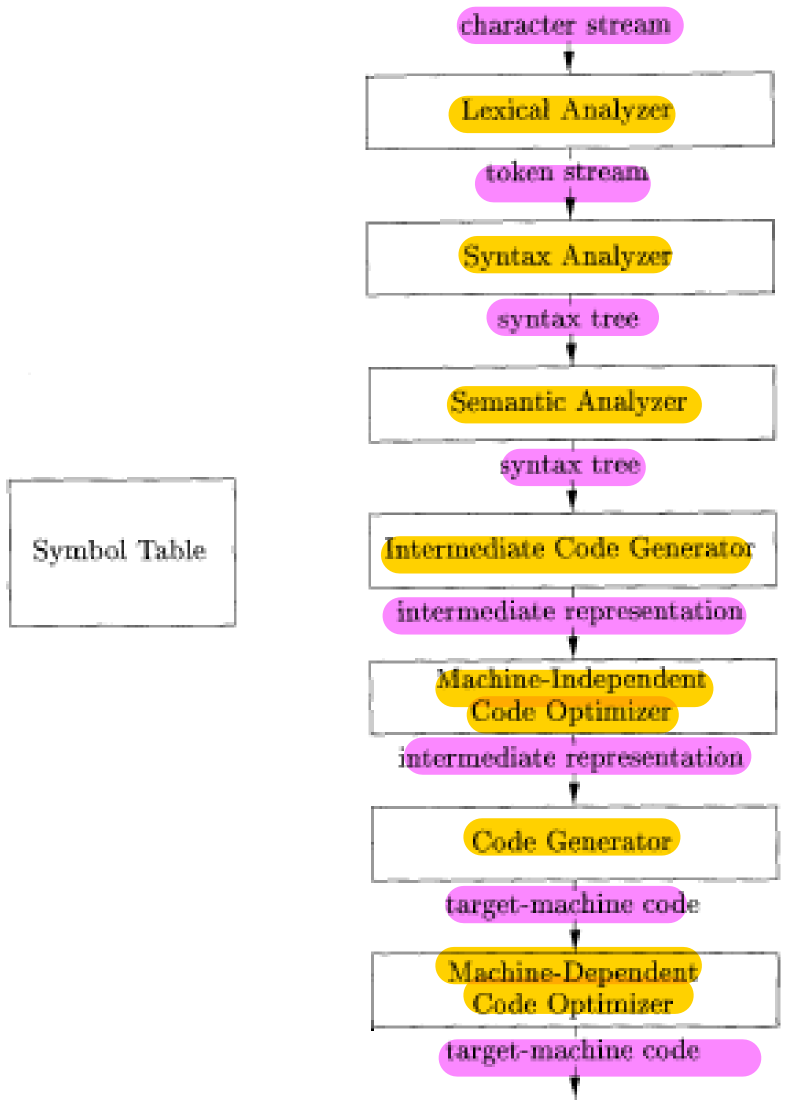
- The synthesis part constructs the desired target program from the intermediate representation.
- Synthesis phase consist of the following sub phases:
  1. Code optimization
  2. Code generation

Phases of compiler

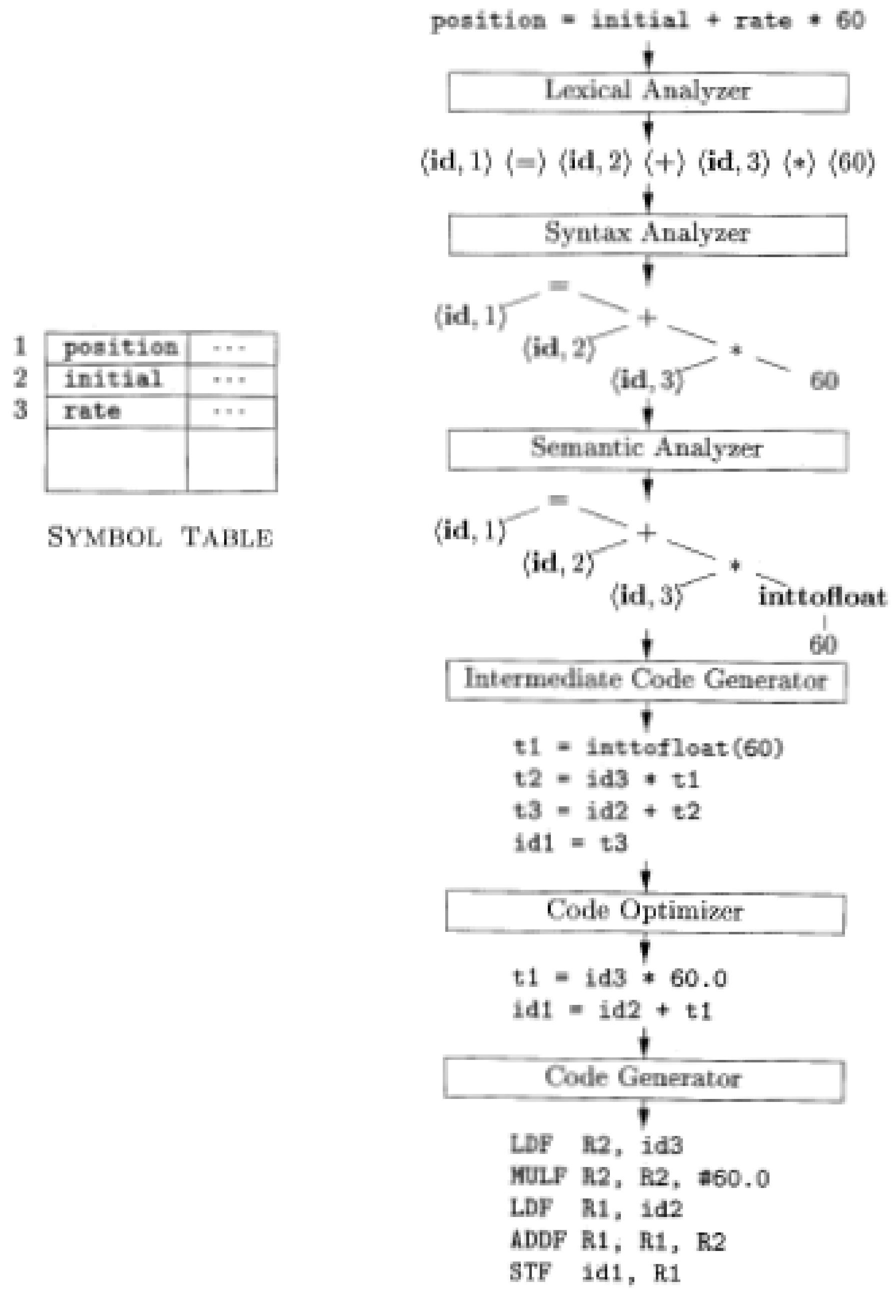
Phases of compiler



Phases of compiler



Phases of compiler



1	position	...
2	initial	...
3	rate	...

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

```
t1 = id3 * 60.0
id1 = id2 + t1
```

```
LDF  R2, id3
MULF R2, R2, #60.0
LDF  R1, id2
ADDF R1, R1, R2
STF  id1, R1
```

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)

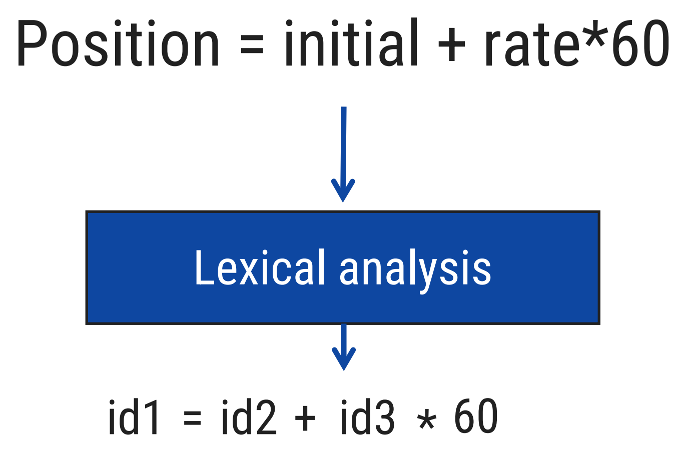
  initial (identifier)

  + (Plus symbol)

  rate (identifier)

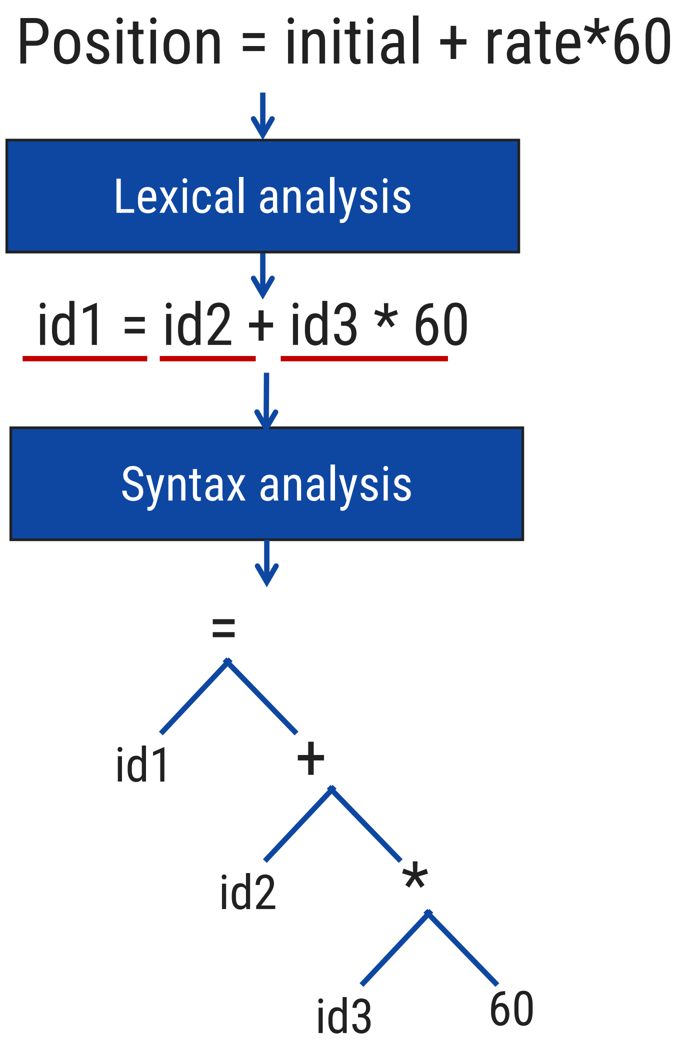
  * (Multiplication symbol)

  60 (Number)



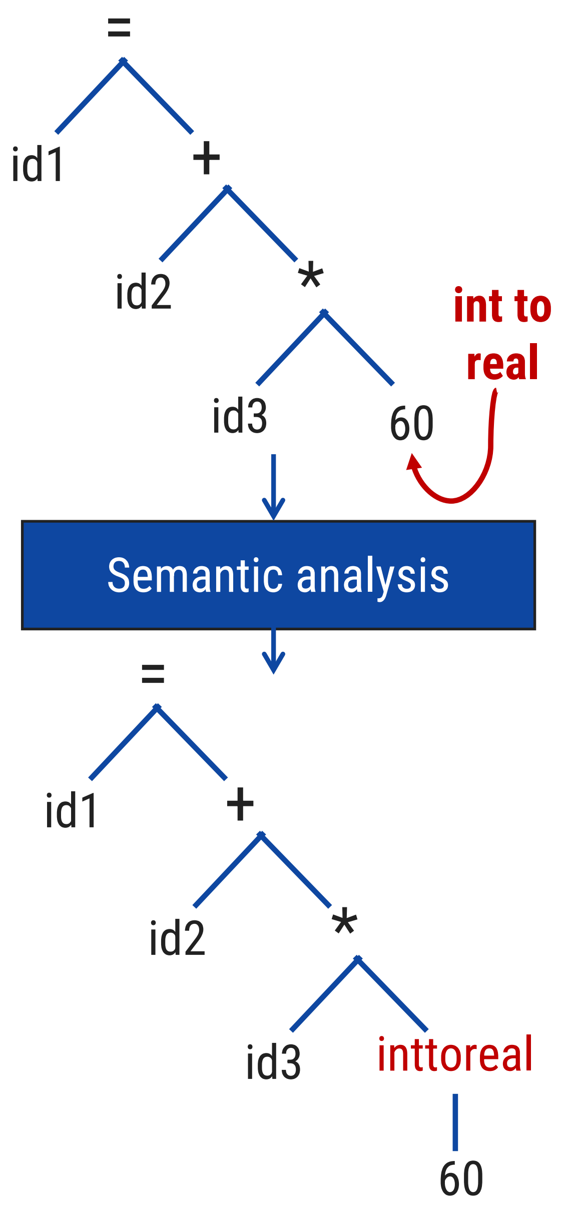
Syntax analysis

- 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 parse tree.



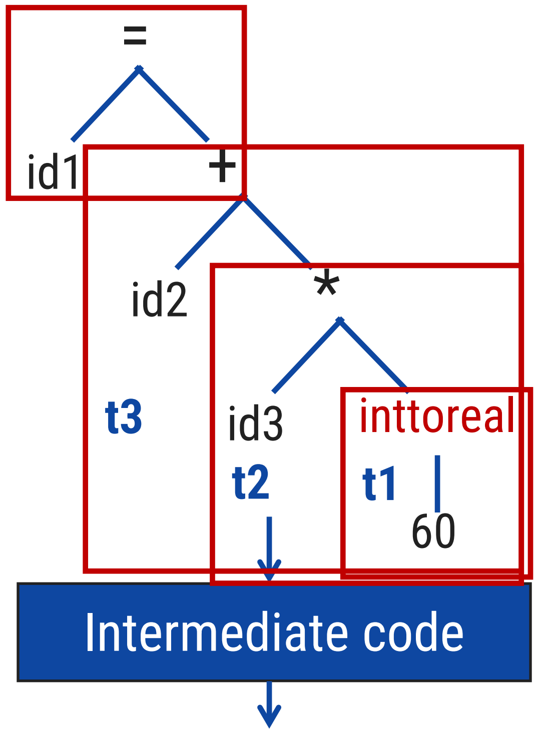
Semantic analysis

- 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.



Intermediate code generator

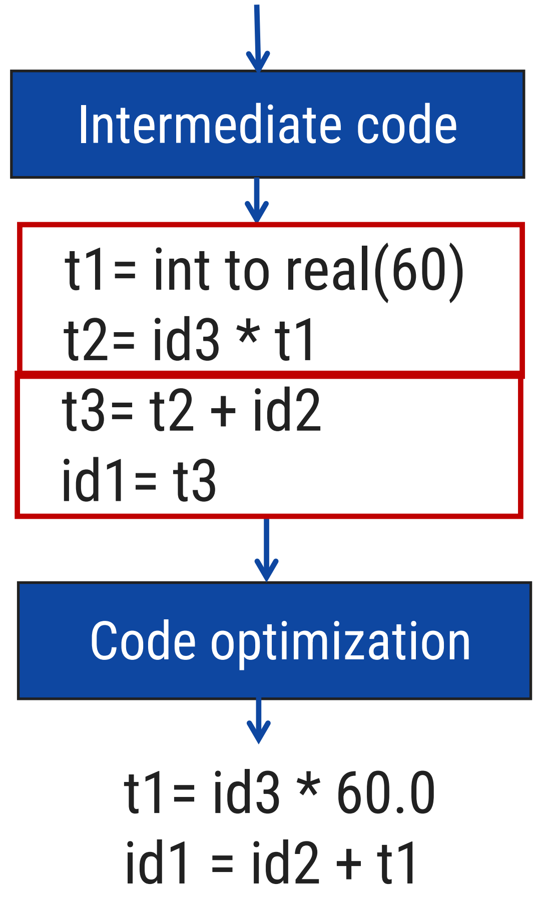
- Two important properties of intermediate code :
  1. It should be **easy to produce**.
  2. **Easy to translate** into target program.
- Intermediate form can be represented using **"three address code"**.
- Three address code consist of a sequence of instruction, each of which has **at most three**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



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

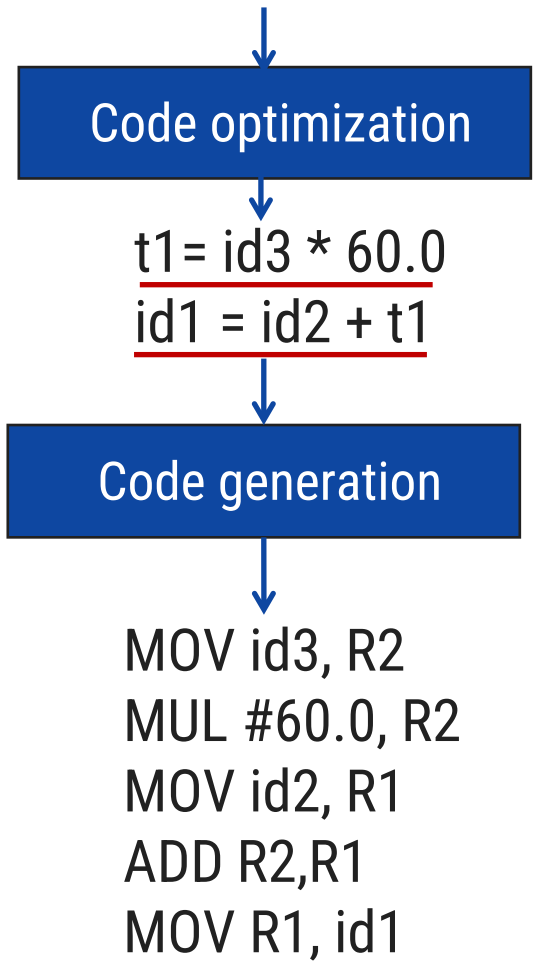
Code optimization

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



Code generation

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

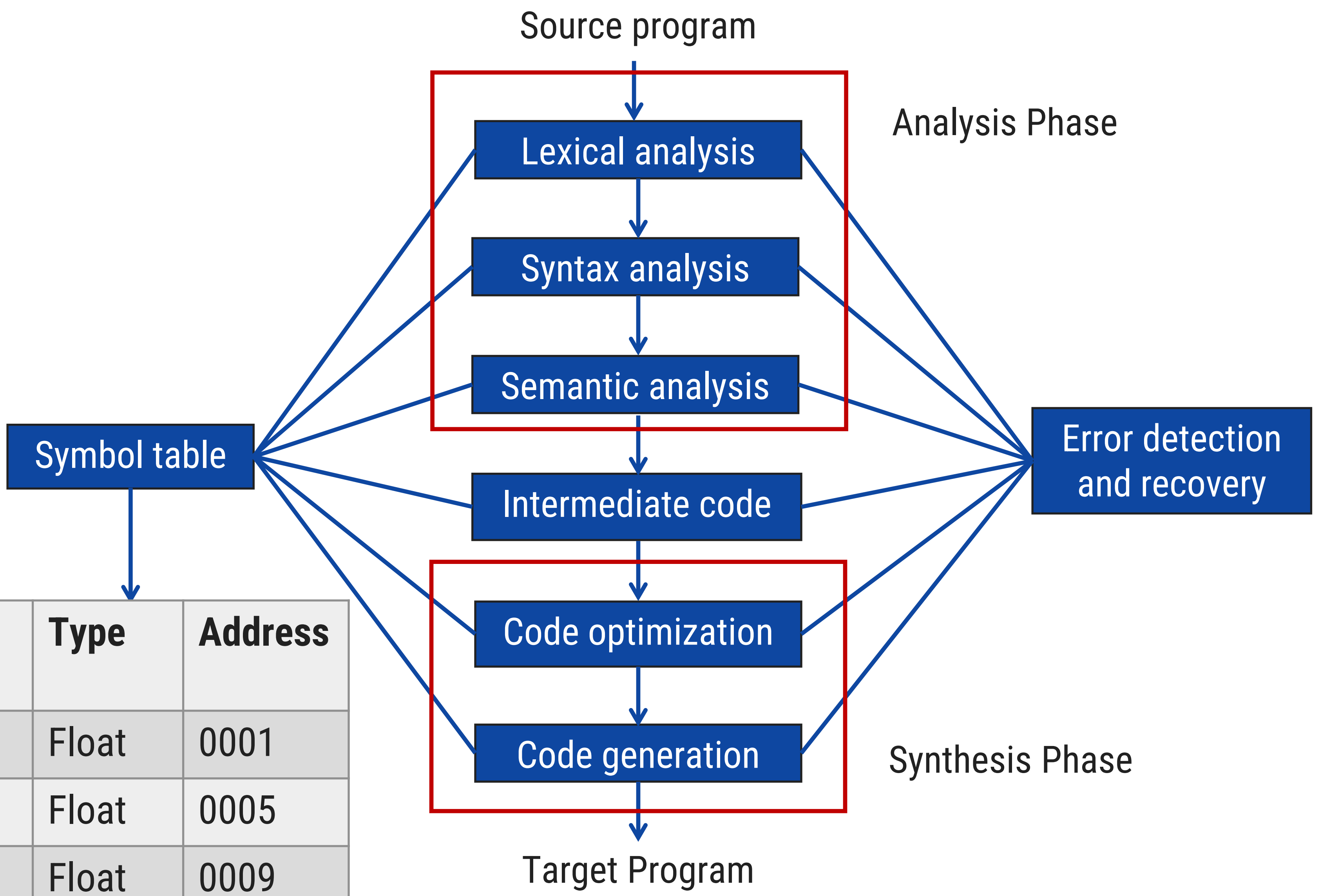


```
t1= id3 * 60.0
id1 = id2 + t1
```

```
MOV id3, R2
MUL #60.0, R2
MOV id2, R1
ADD R2,R1
MOV R1, id1
```

Id3→R2
Id2→R1

Phases of compiler



Variable Name	Type	Address
Position	Float	0001
Initial	Float	0005
Rate	Float	0009

Grouping of Phases

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 & Error handling

Back end

- Depends on target machine and do not depends on source program.
- It includes following phases:
    1. Code optimization
    2. Code generation phase
    3. Error handling and symbol table operation

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

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.

Pass I:

- Perform analysis of the source program and note relevant information.

Pass II:

- 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.

1.2.9 Compiler-Construction Tools

The compiler writer, like any software developer, can profitably use modern software development environments containing tools such as language editors, debuggers, version managers, profilers, test harnesses, and so on. In addition to these general software-development tools, other more specialized tools have been created to help implement various phases of a compiler.

These tools use specialized languages for specifying and implementing specific components, and many use quite sophisticated algorithms. The most successful tools are those that hide the details of the generation algorithm and produce components that can be easily integrated into the remainder of the compiler. Some commonly used compiler-construction tools include

1. *Parser generators* that automatically produce syntax analyzers from a grammatical description of a programming language.

2. *Scanner generators* that produce lexical analyzers from a regular-expression description of the tokens of a language.

3. *Syntax-directed translation engines* that produce collections of routines for walking a parse tree and generating intermediate code.

4. *Code-generator generators* that produce a code generator from a collection of rules for translating each operation of the intermediate language into the machine language for a target machine.

5. *Data-flow analysis engines* that facilitate the gathering of information about how values are transmitted from one part of a program to each other part. Data-flow analysis is a key part of code optimization.

6. *Compiler-construction toolkits* that provide an integrated set of routines for constructing various phases of a compiler.

We shall describe many of these tools throughout this book.