CS4031
Compiler Construction
Lecture 1

Mahzaib Younas

Lecturer, Department of Computer Science

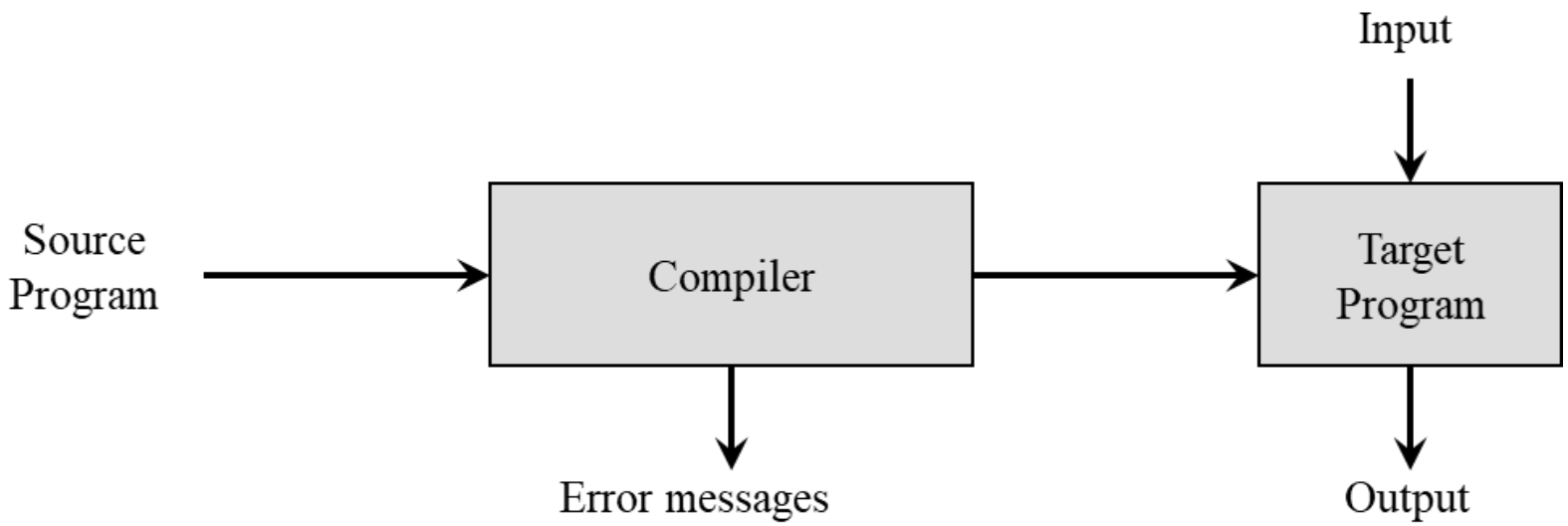
FAST NUCES CFD

Outlines

1. Compilers and Interpreters
2. The structure of a compiler
3. Why learn about compilers?
4. The Evolution of Programming Language
5. Summary

Compiler

- “Compilation”
- Translation of a program **written in a source language** into a semantically equivalent **program written in a target language**



Definition

- A compiler is a program translates (or compiles) a program written in a **high-level programming language** (the source language) that is suitable for human programmers into the low-level machine language (target language) that is required by computers.

- During this process, the compiler will also attempt to spot and report obvious programmer mistakes that detect during the translation process.

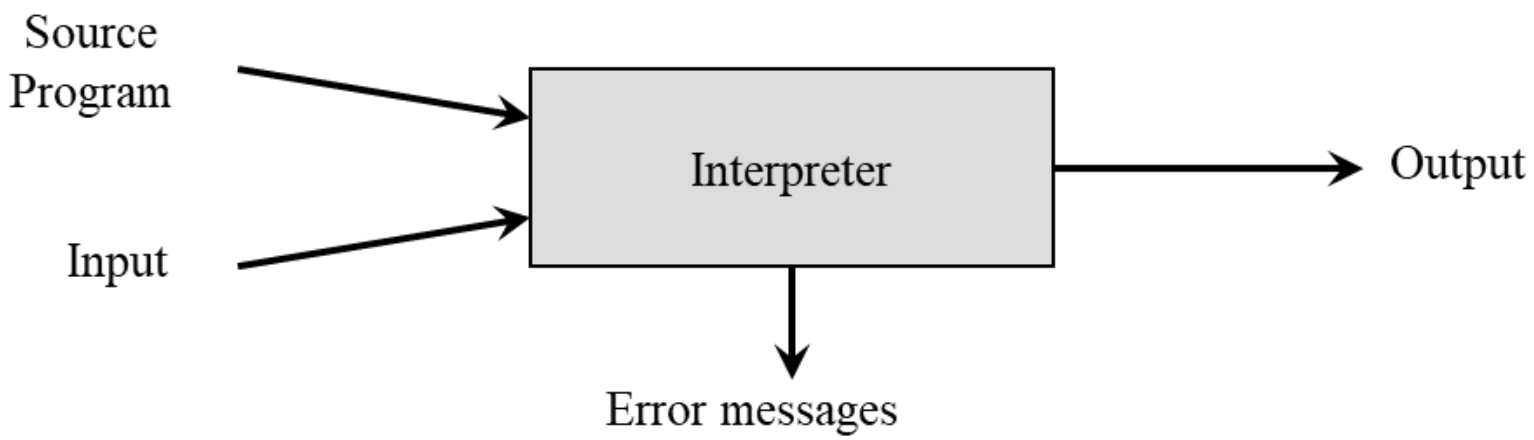
Why we use High Level Language for Programming?

Using a high-level language for programming has a large impact on how fast programs can be developed. The main reasons for this are:

1. Compared to machine language, the notation used by programming languages is closer to the way humans think about problems.
2. The compiler can spot some obvious programming mistakes.
3. Programs written in a high-level language tend to be shorter than equivalent programs written in machine language.
4. The same program can be compiled to many different machine languages and, hence, be brought to run on many different machines.

Interpreter

- Performing the operations implied by the source program



Compiler vs. Interpreter

Compiler	Interpreter
Takes Entire program as **input**	**Take single instruction as input**
It is **Faster**	It is **Slower**
intermediate object code is generated.	**No** intermediate code is **generated**
Required **more memory Due to intermediate object** code	Required **less** memory As no intermediate code is generated
Program not need compile **every time**	**Every time** higher level program is converted into lower level program.
Errors are displayed after entire program is checked.	**Errors** are displayed **for every instruction** interpreted.
Debugging is comparatively hard.	**Debugging** is **easy**.
Ex: C, C++.	Ex: python, Ruby, basic.

Knowledge Require to build Compiler

- Programming Languages
  - Parameter passing techniques
  - Scope of variables
  - Memory allocation techniques
- Theory
  - Automata
  - Context free languages
  - Grammers

Algorithms and data structure
- Hashing
- Graphs
- dynamic programming

Computer Architecture
- Assembly Language

Software Engineering

The Analysis-Synthesis Model of Compilation

There are two parts to compilation:

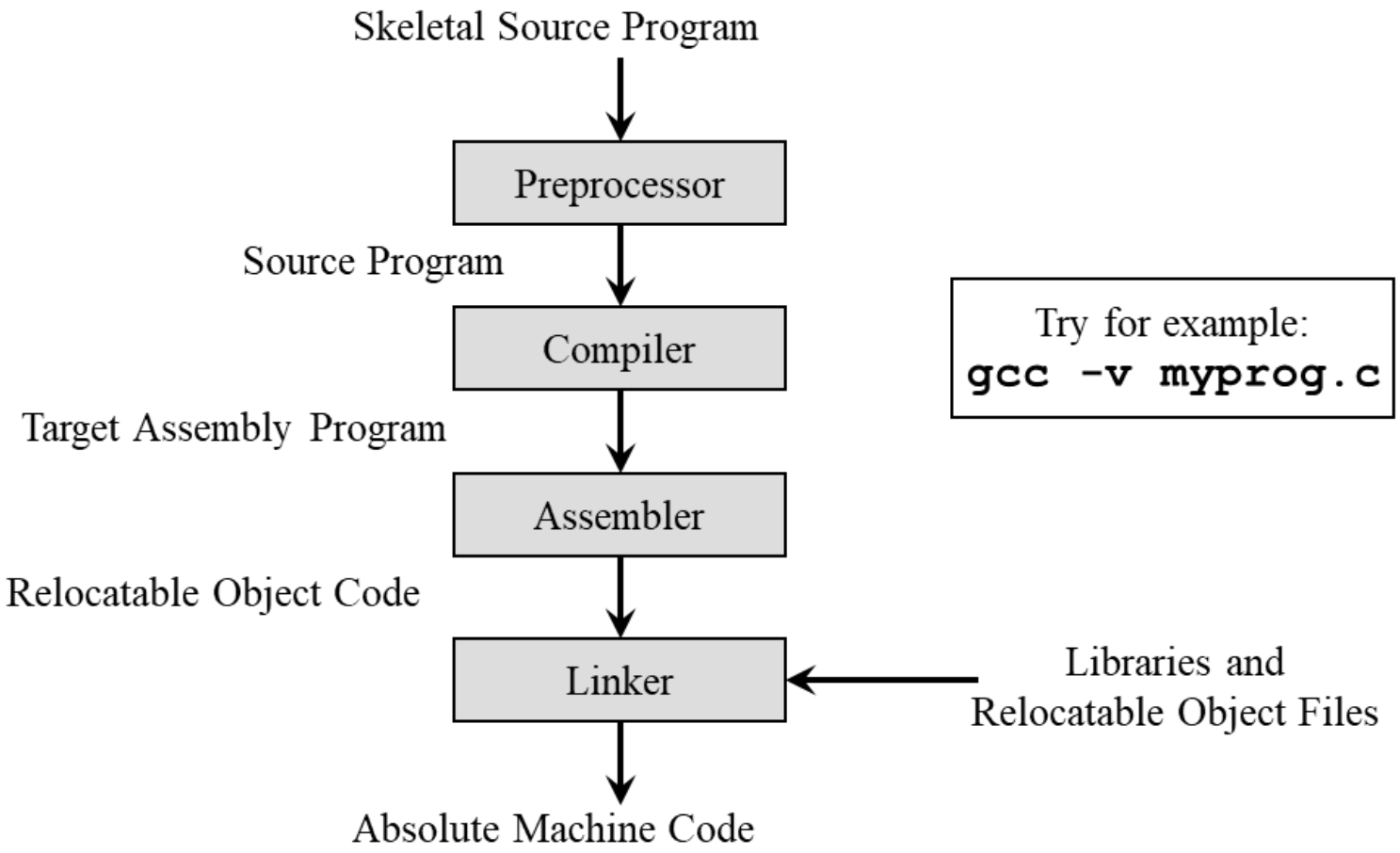
- Analysis determines the operations implied by the source program which are recorded in a tree structure

- Synthesis takes the tree structure and translates the operations therein into the target program

Why Learn About Compiler?

- It is considered a topic that you should know in order to be “well-cultured” in computer science.
- A good craftsman should know his tools, and compilers are important tools for programmers and computer scientists.
- The techniques used for constructing a compiler are useful for other purposes as well.
- There is a good chance that a programmer or computer scientist will need to write a compiler or interpreter for a domain-specific language.

Preprocessors, Compilers, Assemblers, and Linkers



Preprocessor

- A source program may be divided into modules stored in separate files and may consist of macros.
- Preprocessor produce input to the compiler.
- It process the source code before the compilation and produce a code that can be more efficiently used by the compiler.
- It cannot be include as the part of single pass compiler.

Example

Sample C Program

```
#include<stdio.h>
#define min(x,y)

Void fun()
{
Int a = 1;
Int b =2;
Int c;
c = min(a,b)
}
```

Preprocessor output

```
Void fun()
{
Int a = 1;
Int b =2;
Int c;
c = if((a)<(b))
Print a;
else
Print b;
}
```

If the C language program is an input for a preprocessor, then it produces the output as a C program where there are no #include and macors, that is C program only C statement.

Compiler

- To convert any high level language to machine code, one translator is mandatory that is nothing but a compiler.

A compiler is translator that convert the high level language into the machine level language,.

Assembler

- The third stage is the assembly stage, this stage is to convert the assembly code generated in the second stage into our executable file, which is to convert our assembly language into a machine language that can be executed by our machine.

- after opening it through vim, we will find that this is completely confusing to our eyes, but this is a binary language that the machine can understand

Linker

- The linker combines all external programs (such as libraries and other shared components) with our program to create a final executable.

See Step by step process

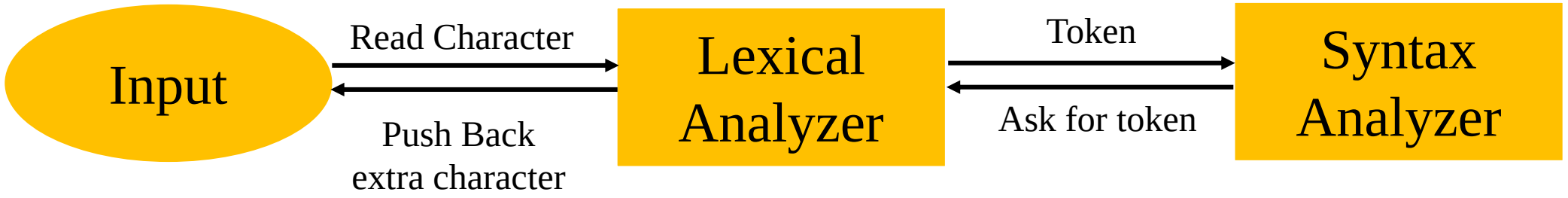
- To see what happens after each stage, we will have the process with the following flags
- **-E** : Stop after the preprocessing stage; do not run the compiler proper.
- **-S** : Stop after the stage of compilation proper; do not assemble.
- **-c** : Compile or assemble the source files, but do not link.

Phases of Compiler

- Lexical Analysis (Make token Stream)
- Syntax Analysis (Syntax Tree)
- Semantic Analyzer (syntax Tree)
- Intermediate code generator (Intermediate Representation)
- Code Optimization (Target Machine Code)
- Machine Dependent code optimizer (Target machine code)

Lexical Analysis

- Lexical analysis is the process of breaking down the source code of the program into smaller parts, called **tokens.**



- **Remove comments**
- **Remove white spaces**
- **Also known as Scanner or tokenizers**

Example

- A simple statement of C or C++ code

sum = 2 + c;

- Create the token of above line

sum	**=**	**2**	**+**	**C**
identifier	**Assignment operator**	**Constant**	**operator**	**Identifier**

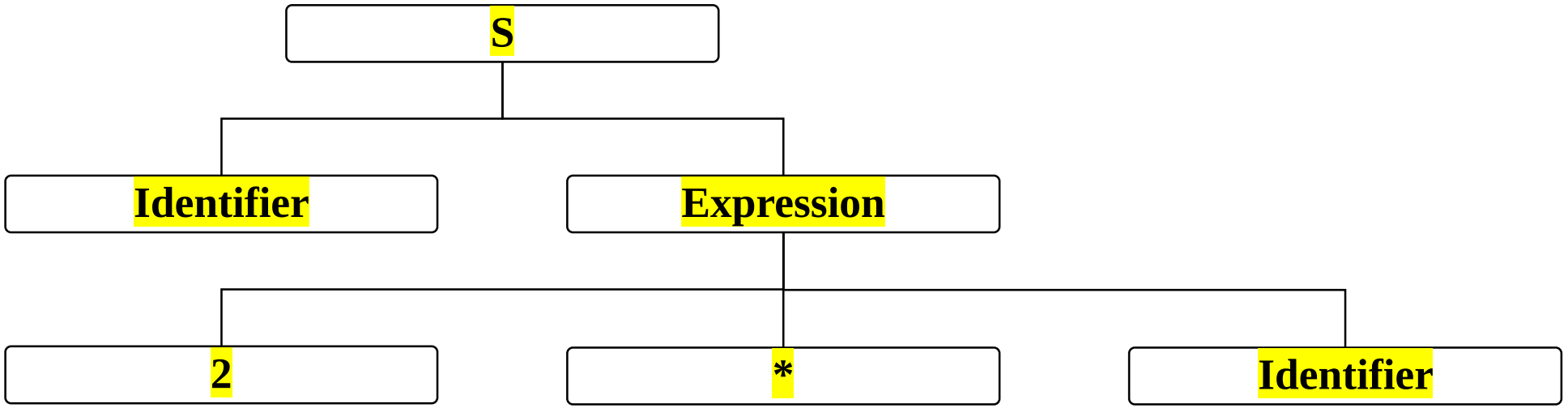
Syntax Analysis (Parser)

- The second phase of a compiler is syntax analysis, also known as parsing. This phase takes the stream of tokens generated by the lexical analysis phase and checks whether they conform to the grammar of the programming language. The output of this phase is usually an Abstract Syntax Tree.

AST, is a representation of the structure of a programming language. It is a tree-like structure that is composed of nodes, each of which represents a language element like a variable, operator, or keyword.

Example

- sum = 2 * c
- Th AST of the given C code will be



Semantic Analysis

In semantic analysis, compiler check the logical type error

- It checks whether the parse tree follows the rules of language. Semantic analyzer keeps track of identifiers, their types and expressions. The output of semantic analysis phase is the annotated tree syntax.

Intermediate Code Generator

- compiler generates the source code into the intermediate code. Intermediate code is generated between the high-level language and the machine language.
- Convert the code into three address code

Example

sum = 2 * c

- Conversion of above statement in intermediate code generator

Temp = 2 *c

Sum = temp;

Code Optimization

- Code optimization is an optional phase. It is used to improve the intermediate code so that the output of the program could run faster and take less space.
- It removes the unnecessary lines of the code and arranges the sequence of statements in order to speed up the program execution.

Machine Dependent code optimizer

- Code generation is the final stage of the compilation process. It takes the optimized intermediate code as input and maps it to the target machine language.
- Code generator translates the intermediate code into the machine code of the specified computer.

Example

- Example: Register based system
- Lets says

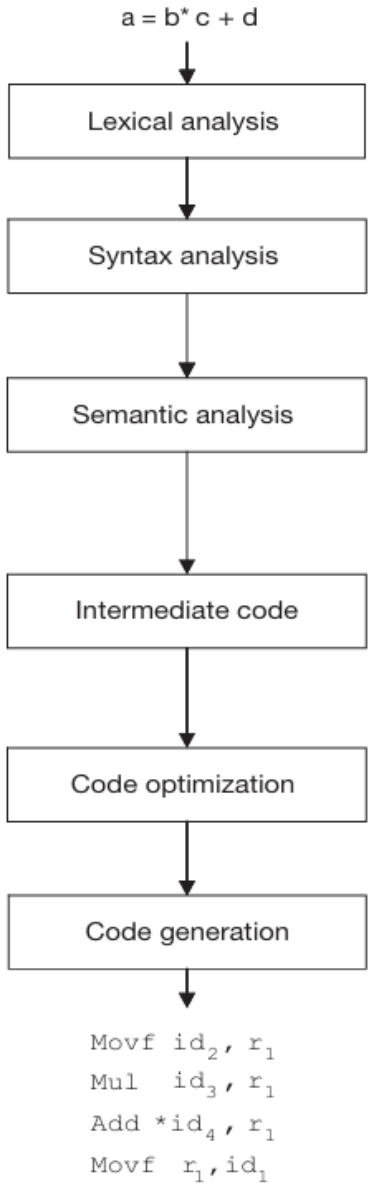
R1 = 2

R2 = x

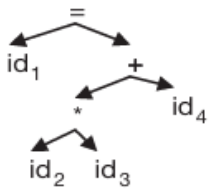
MUL R1 R2

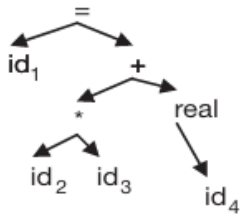
Store sum R1

Example:
a = b* c + d



$id_1 = id_2 * id_3 + id_4$





$t_1 = real(id_4)$
$t_2 = id_2 * id_3$
$t_3 = t_2 + t_1$
$id_1 = t_3$

$t_2 = id_2 * id_3$
$id_1 = t_2 + real(id_4)$

Example 2:

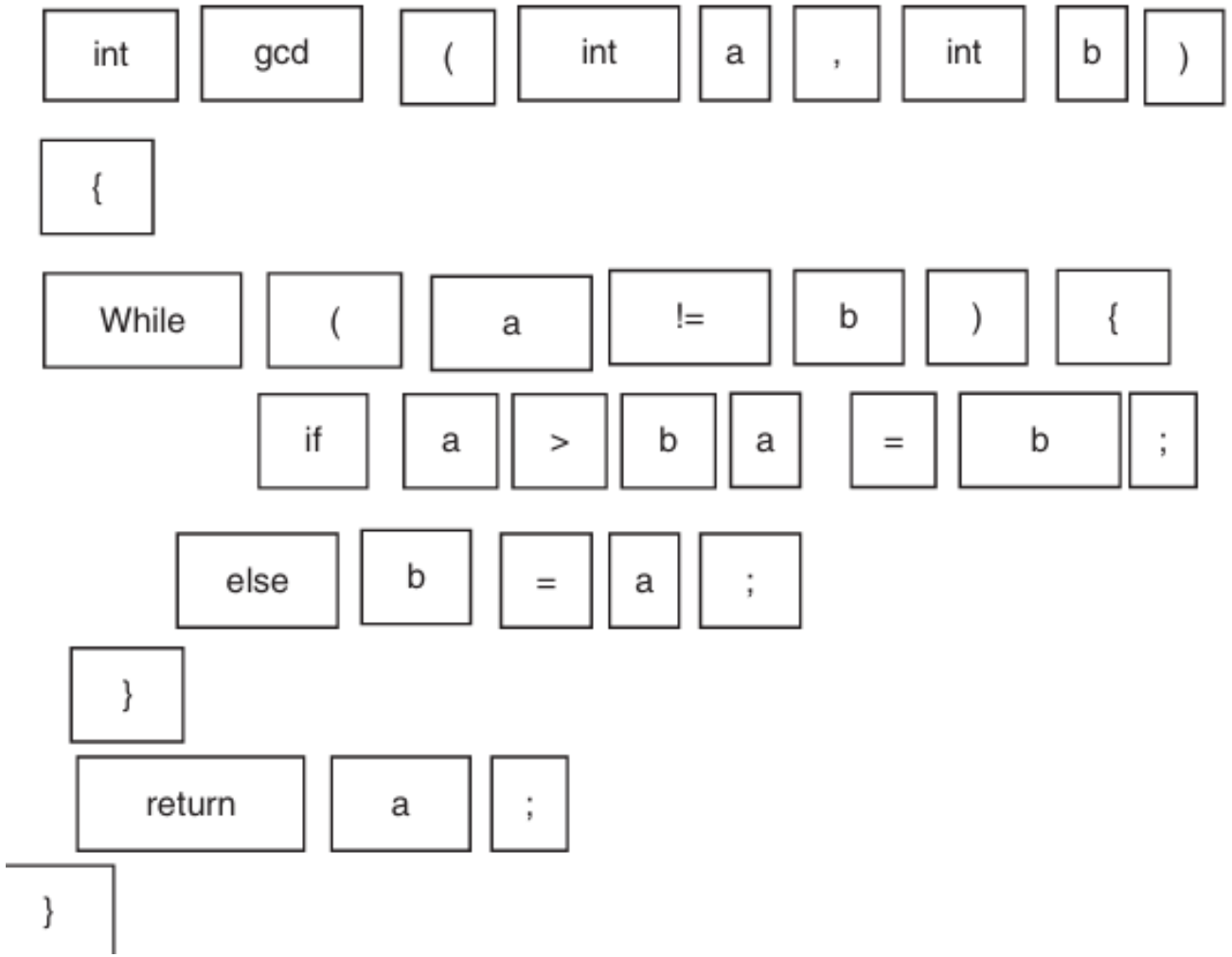
- Consider the following simple C statement

1. Price = old + Rate * 50
2. Grade = Mid1 + Mid2 + Final;

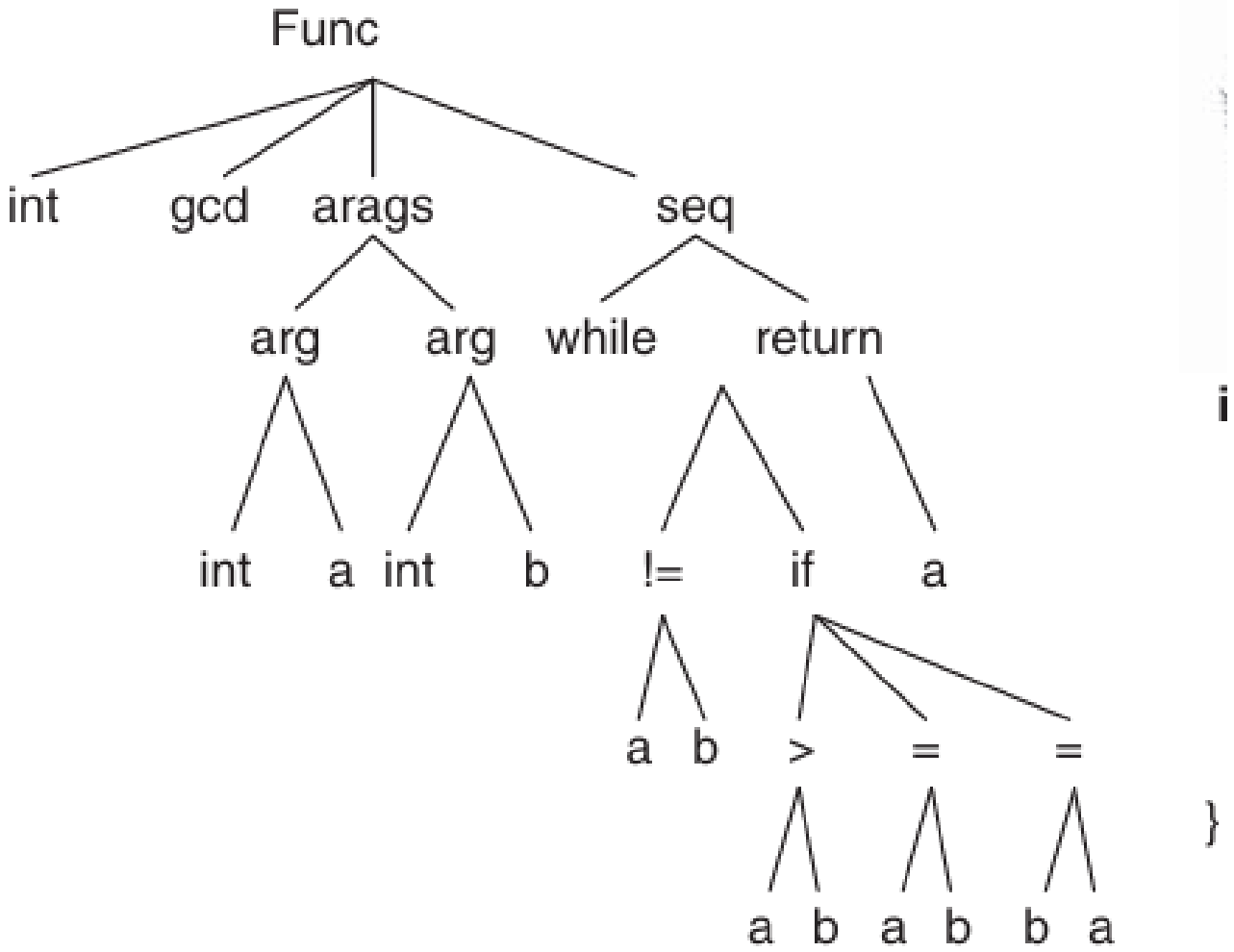
Show the output of each phase of compiler on the following input

```
int gcd(int a, int b)
{
while (a != b) {
    if (a > b) a =b;
    else b =a;
}
return a;
}
```

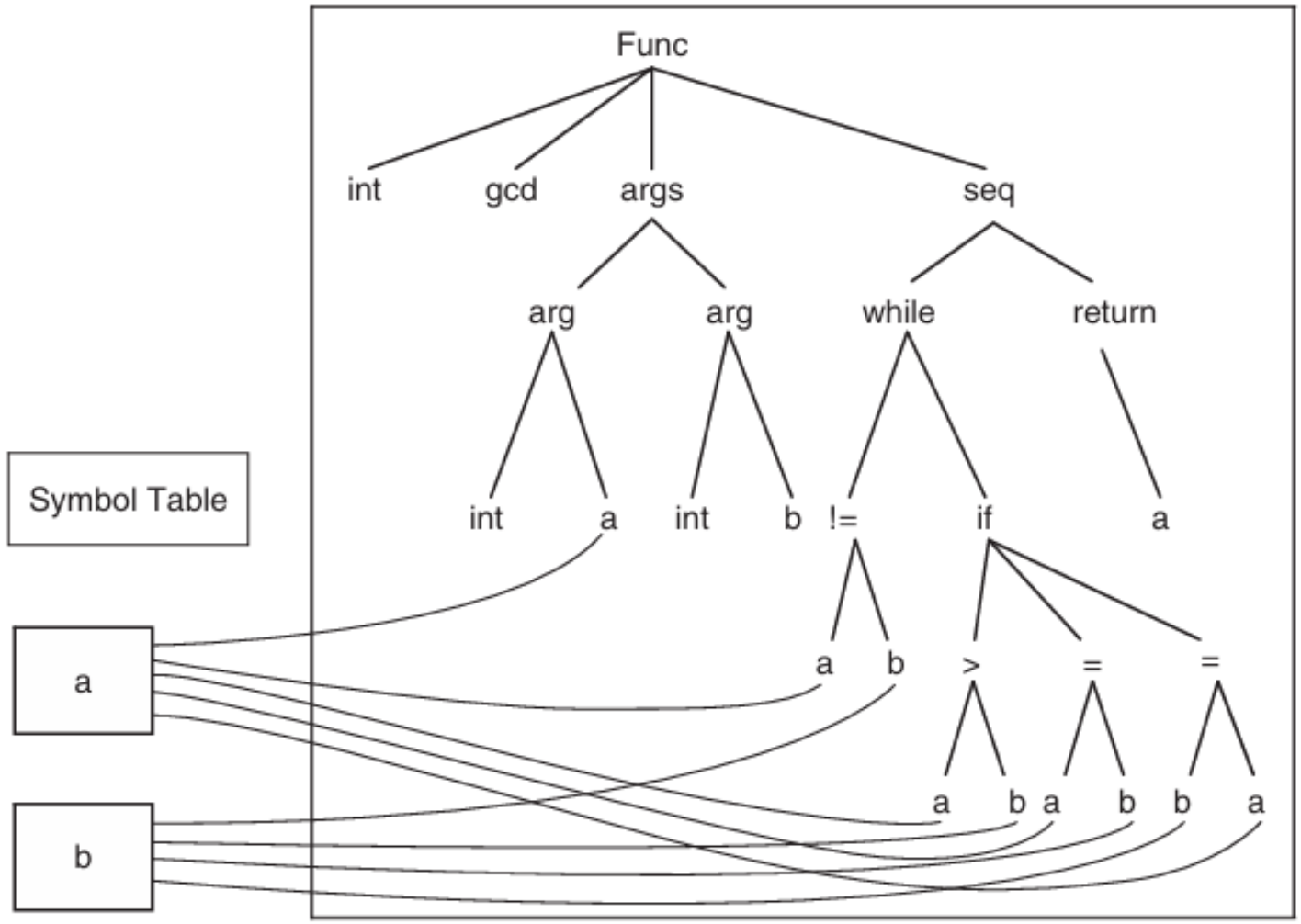
Lexical Analysis



Syntax Analysis



Semantic Analysis



Intermediate code Generator

If (a !=b) go to L1

go to Last

L1: if a > b goto L2

goto L3

L2: a=b

goto Last

L3: b=a

Last: goto calling program

Target Code

```
mov a, R0                % Load a from stack
mov b, R1                % Load b from stack
cmp, !=, R0,R1
jmp .L1                  % while (a != b)
jne .L5                  % if (a < b)
subl %edx,%eax           % a =b
jmp .L8
L5: subl %eax,%edx       % b =a
jmp .L8
L3: leave                % Restore SP, BP
ret
```

Classifications of Compilers

- There are three types of compilers

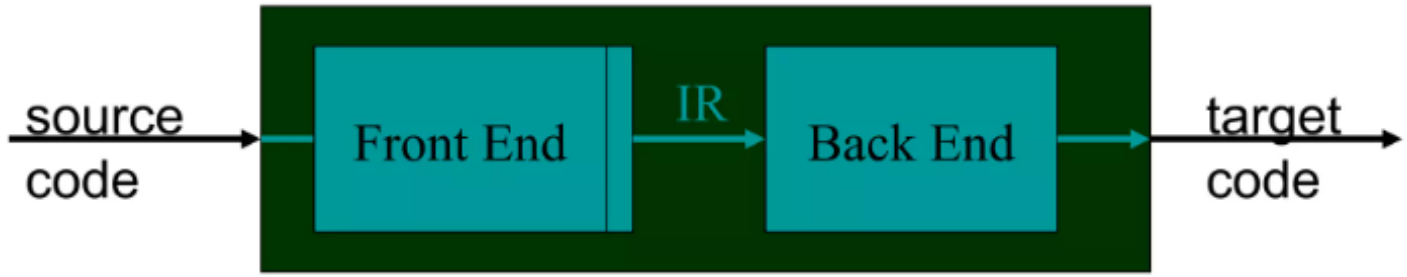
1. Single pass Compiler
2. Two Pass Compiler
3. Multi Pass Compiler

Single Pass Compiler

- If we combine or group all the phases of compiler design in a **single** module known as a single pass compiler.
- Source Code Directly transfer to the object code
- Example: Pascal

Two-Pass Compiler

- Its based on two parts
- **First Pass** is referred as
  - Front end
  - Analytic part
  - Platform independent
- **Second Pass** is referred as
  - Back end
  - Synthesis Part
  - Platform Dependent



Two Pass Compiler

Front End

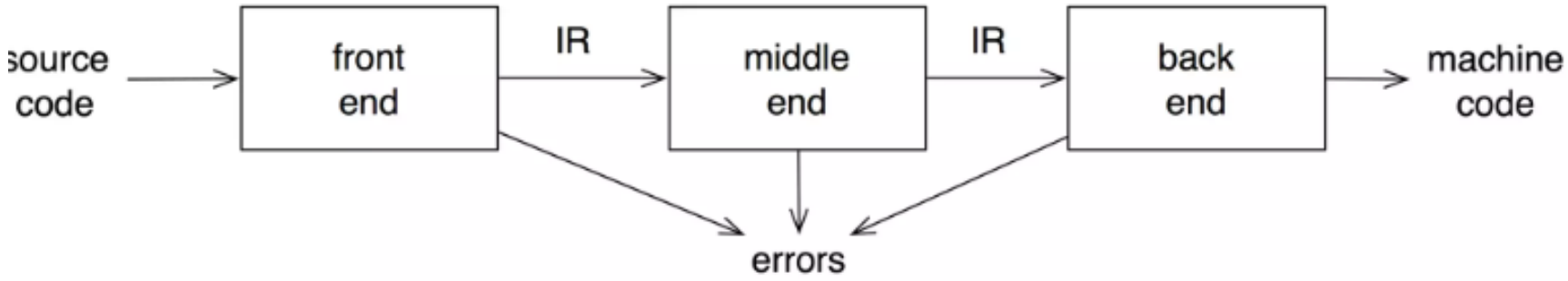
- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- Intermediate code generation

Back End

- Code Optimization
- Target Code Generation

Multi Pass Compiler

- A multi-pass compiler can process the source code of a program multiple times.
- In the first pass
  - the compiler can read the source code, scan it, extract the tokens and save the result in an output file.
- In the second pass
  - the compiler can read the output file produced by the first pass, build the syntactic tree and implement the syntactical analysis. The output of this phase is a file that includes the syntactical tree.
- In the third pass,
  - the compiler can read the output file produced by the second pass and check that the tree follows the rules of language or not. The output of the semantic analysis phase is the annotated tree syntax. This pass continues until the target output is produced.



Analyzes and changes IR
Goal is to reduce runtime
must preserve values