B.SC. CSIT
6Th Semester
Compiler Design and ConstructionCSC-365

Unit 1
Introduction to Compiler - 3hrs

Instructor

Tekendra Nath Yogi

Tekendranath@gmail.com

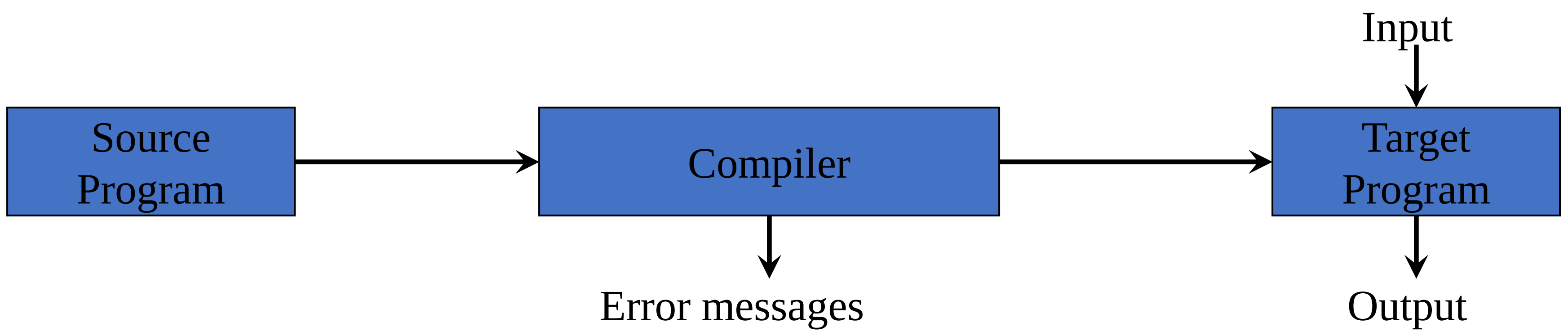
Contents

Compiler

- As we know, humans are good at giving instructions in English like language, whereas computers can only process binary language.
  - It is cumbersome for human to write program in binary.
  - Computer cannot execute even a single instruction given in any other form.
- So, there is a need of a translator that translates the computer instructions given in English like language to binary language.
- Two main types of translators in computer:
  - Compiler and
  - Interpreter

Compiler

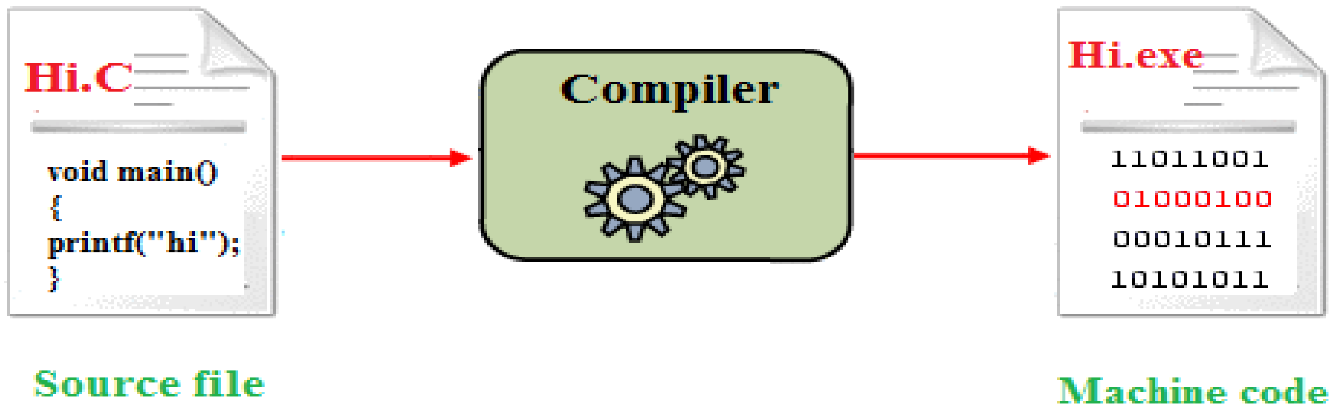
- Compiler is a program that can read a program in one language - the source language - and translate it into an equivalent program in another language - the target language.



- **Source program:** May be program written in a high-level programming language like C code.
- **Target program:** May be some other programming languages program, or machine readable code.
- **Error Messages:** Grammatical errors, errors due to the undefined meanings, etc.

Compiler

- Apart from translating source code from high level language to low level language, compiler has other responsibilities too.
  - The compiler also makes the end code efficient which is optimized for execution time and memory space.
  - It also detects and reports the error during translation.
- **Example:**

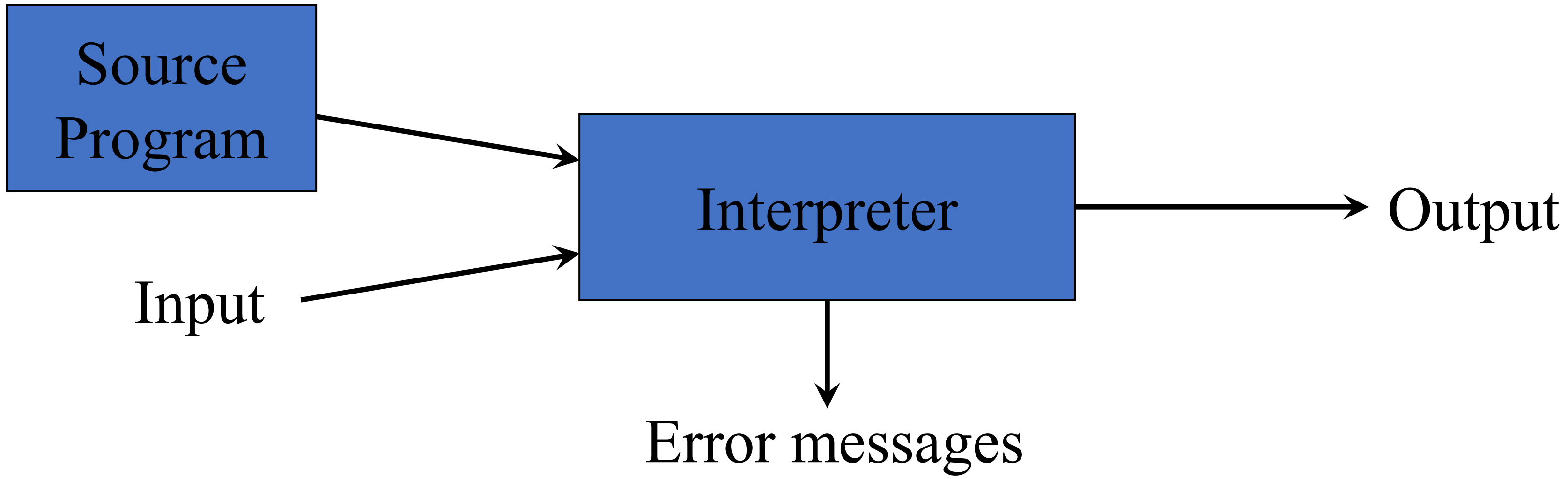


```
void main()
{
printf("hi");
}
```

```
11011001
01000100
00010111
10101011
```

Interpreter

- An interpreter converts high-level language into low-level machine language, just like a compiler.
  - Compiler transforms code written in a high-level programming language into the machine code, at once, before program runs, whereas an Interpreter converts each high-level program statement, one by one, into the machine code, during program run.



Language processing system

- In addition to a compiler, several other programs may be required to create an executable target program, as shown in Figure below:

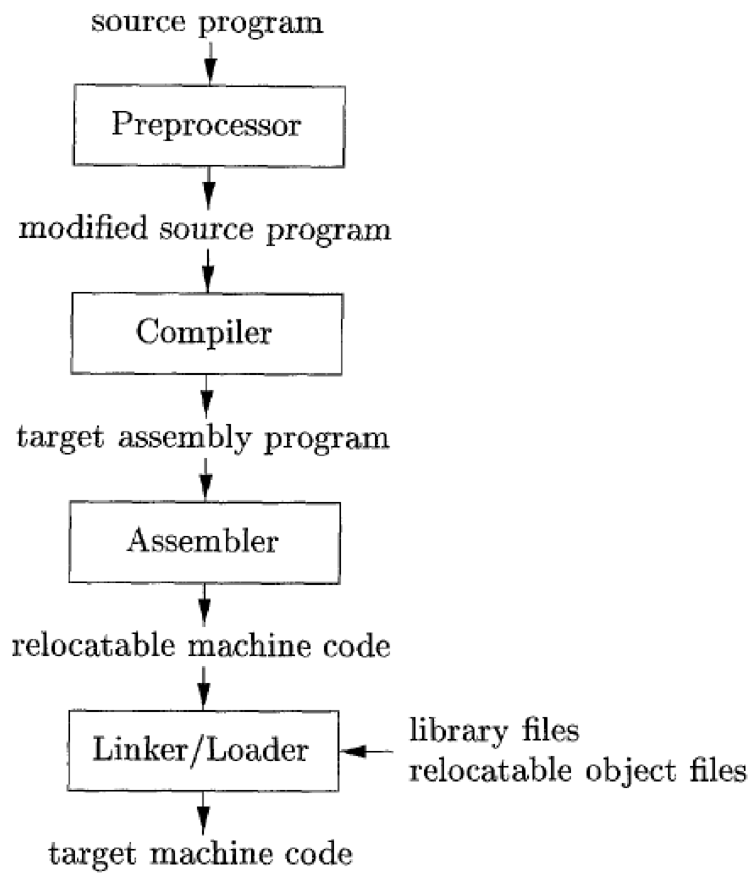


Figure: A Language processing system

Language processing system

- Preprocessor:
  - 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.
- Compiler:
  - 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.

Language processing system

- Assembler:
  - The assembly language is then processed by a program called an assembler that produces re-locatable machine code as its output.
- Linker/ Loader:
  - Large programs are often compiled in pieces, so the re-locatable machine code may have to be linked together with other re-locatable 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 the entire executable object files into memory for execution.

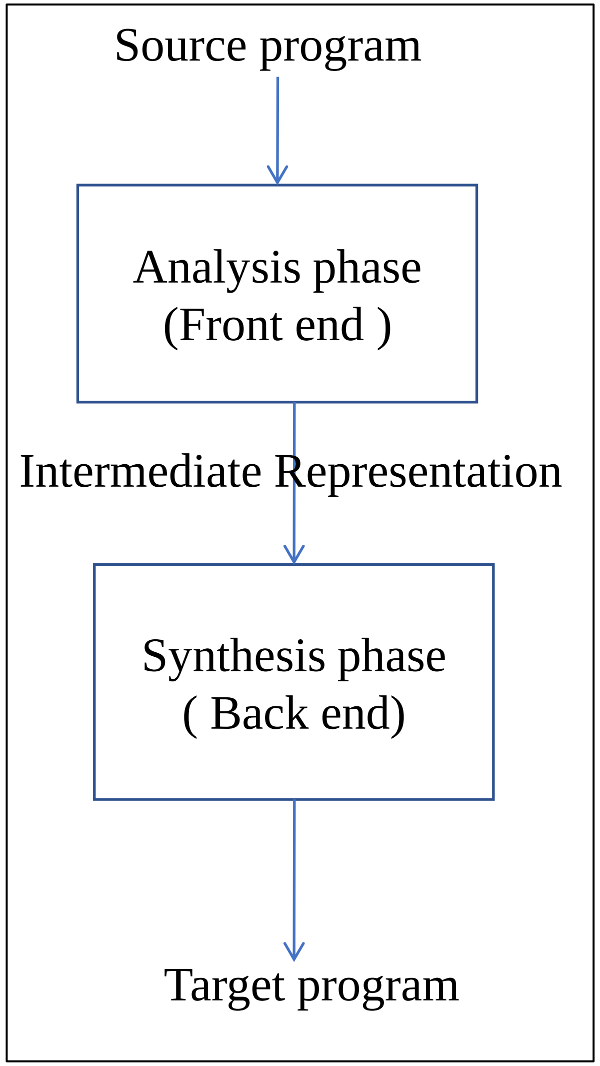
Structure of a compiler

Analysis and Synthesis Model of Compilation

- High level examination of Analysis and synthesis model of compilation consists of two major phases. They are:

1. In analysis phase,
   - ✓ an intermediate representation is created from the given source program.
   - ✓ Lexical Analysis, Syntax Analysis and Semantic Analysis are the parts of this phase.

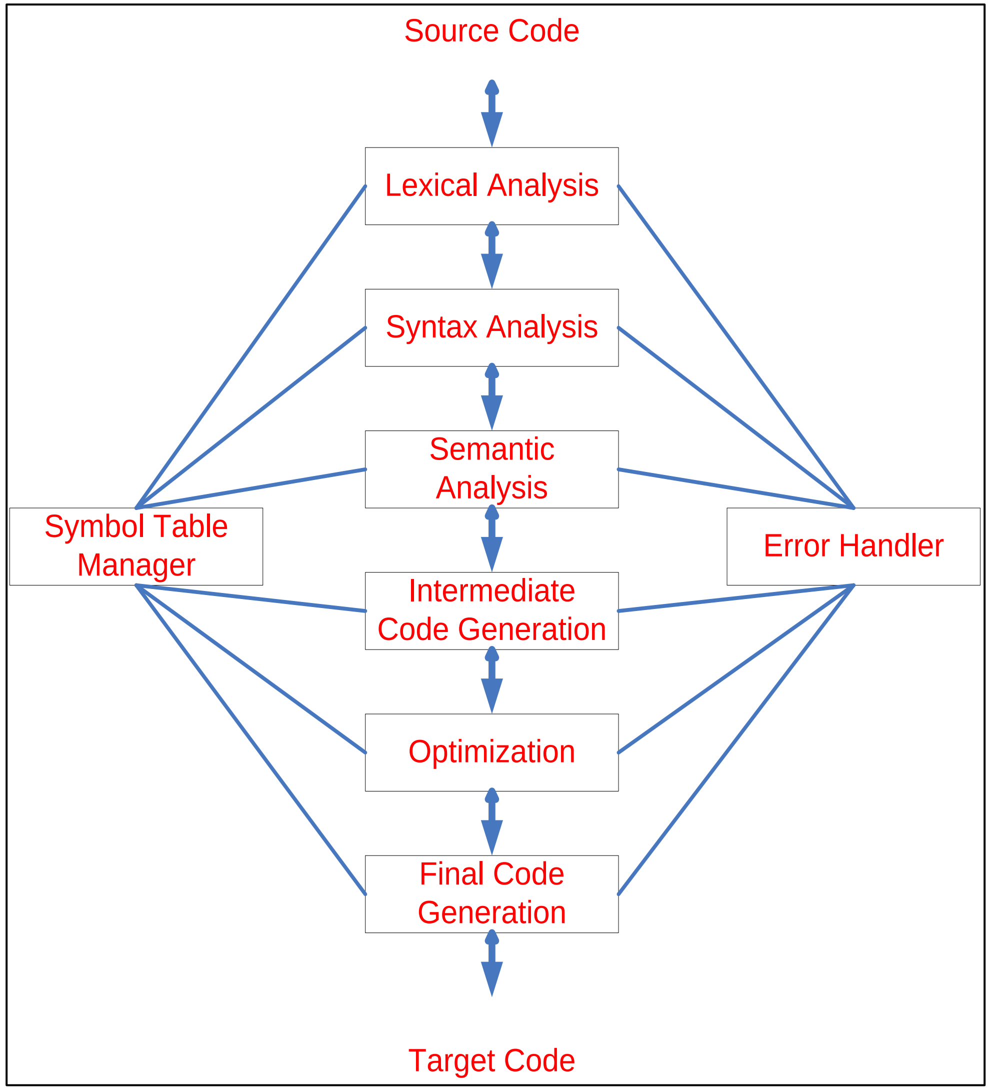
- 2. In synthesis phase,
   - ✓ an equivalent target program is created from this intermediate representation.
   - ✓ Intermediate Code Generation, Code Optimization, and Code Generation are the parts of this phase.


Source program
Analysis phase (Front end)
Intermediate Representation
Synthesis phase (Back end)
Target program

Structure of a compiler

Analysis and Synthesis Model of Compilation – Sub-phases

- A more detail examination of the compilation process contains the sequence of various sub-phases.
- Each sub-phase takes source program in one representation and produces output in another representation.
    - Each sub-phase takes input from its previous stage and gives output to next phase.
- They communicate with **error handlers.**
- They also communicate with the **symbol table manager.**



Structure of a compiler

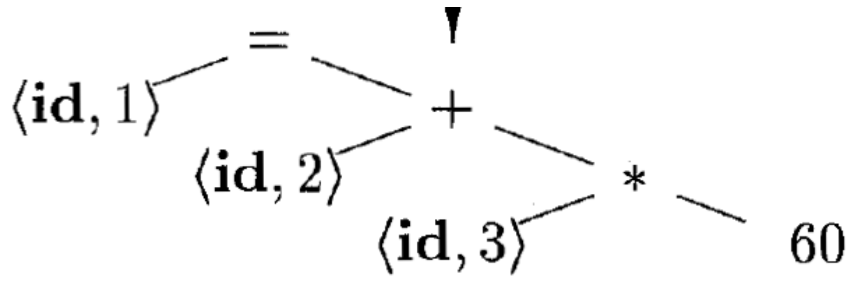
Analysis and Synthesis Model of Compilation – Analysis

- **Lexical Analysis (Scanning) :**
  - The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes.
  - For each lexeme, the lexical analyzer produces as output a token of the form <token-name, attribute-value> that it passes on to the subsequent phase, syntax analysis.
  - For example, suppose a source program contains the assignment statement: position = i n i t i a l + r a t e * 60
  - For the above statements following sequence of tokens are generated after lexical analysis: <id, 1> < => <id, 2> <+> <id, 3> <*> <60>.
  - Blanks separating the lexemes would be discarded by the lexical analyzer.

Structure of a compiler

Analysis and Synthesis Model of Compilation – Analysis

- **Syntax analysis (Parser):**
  - The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream.
  - A syntax tree for the token stream <id, 1> <=> <id, 2> <+> <id, 3> <*> <60> is shown as the output of the syntactic analyzer in Figure below:

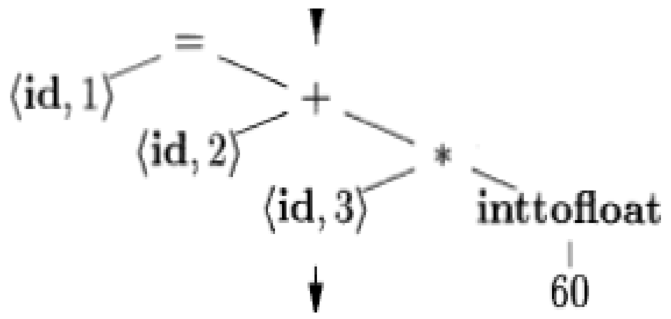


  - The subsequent phases of the compiler use the grammatical structure to help analyze the source program and generate the target program.

Structure of a compiler

Analysis and Synthesis Model of Compilation-- Analysis

- **Semantic analysis:**
  - The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition.
  - It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.
  - For Example:



Structure of a compiler

Analysis and Synthesis Model of Compilation – Synthesis

- **Intermediate code generation:**
  - An intermediate language is often used by many compiler for analyzing and optimizing the source program.
  - The intermediate language should have two important properties:
    - It should be simple and easy to produce.
    - It should be easy to translate to the target program
  - A compiler may produce an explicit intermediate codes representing the source program. These intermediate codes are generally machine (architecture) independent. But the level of intermediate codes is close to the level of machine codes.
  - For example:

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

Structure of a compiler
Analysis and Synthesis Model of Compilation – Synthesis

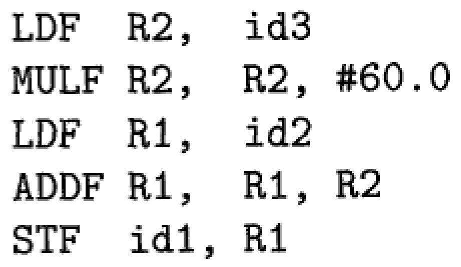
- **Code Optimization:**
  - The compile time code optimization involves static analysis of the intermediate code to remove extraneous operation in of time and space. i.e.,
    - Detection of redundant function calls
    - Detection of loop invariants
    - Common sub-expression elimination
    - Dead code detection and elimination
  - For example:

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

Structure of a compiler

Analysis and Synthesis Model of Compilation – Synthesis

- **Code Generation:**
  - The code generator takes as input an intermediate representation of the source program and maps it into the target language.
  - If the target language is machine code, registers or memory locations are selected for each of the variables used by the program. Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task.
  - For example, using registers R1 and R2, the intermediate code in above might get translated into the machine code:



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

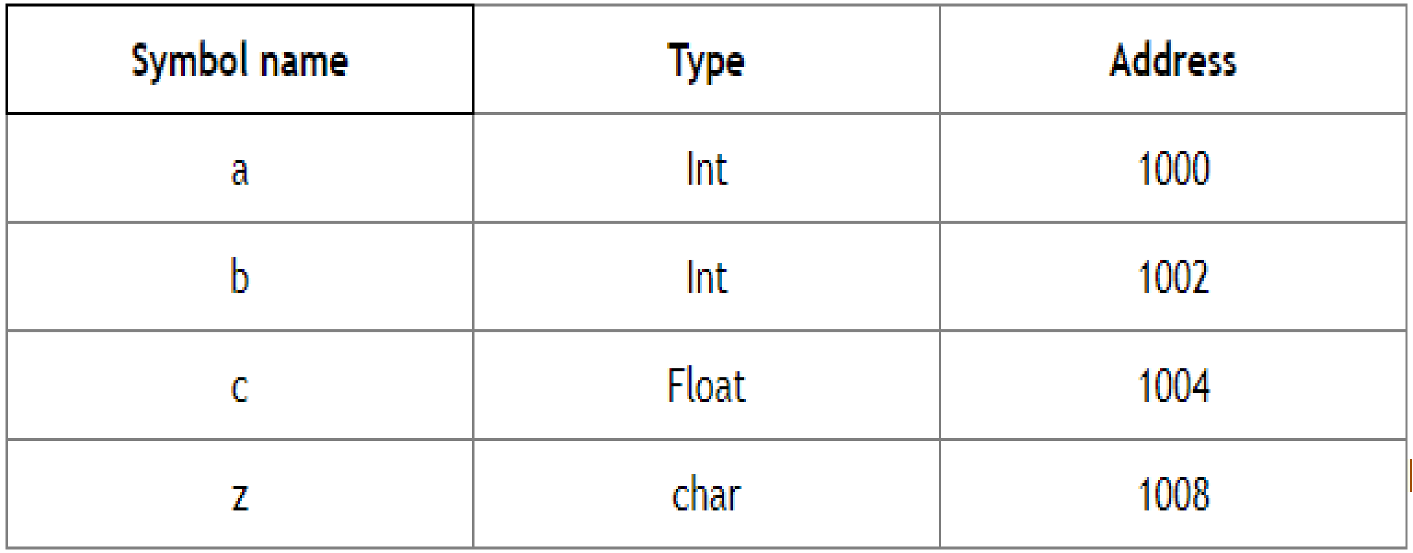
Symbol table

- Symbol table is a data structure that is used by the compiler to stores identifier and its related attributes like storage allocated, type, scope, number and types of the parameters, type returned (in case of functions).
- This data structure is maintained and accessed through out the phases of the compiler.
- The information in the symbol table is collected incrementally by the analysis phase of a compiler and used by the synthesis phases to generate the target code.
- **Operations:**
    - allocate( ): to allocate a new empty symbol table
    - free( ): to remove all entries and free the storage of a symbol table
    - set_attribute: to associate an attribute with a given entry

Symbol table

- get_attribute: to get an attribute associated with a given entry
- insert(id, “x”): inserts the token id named x in the symbol table.
- Lookup(“z”): looks up for the occurrence of the string z in the symbol table and returns value accordingly , error if not found.

- **Example:** int a, b; float c; char z;



Symbol name	Type	Address
a	Int	1000
b	Int	1002
c	Float	1004
z	char	1008

Error Handler

- Errors can be encountered at any phase of the compiler during compilation of the source program for several reasons such as:
    - In lexical analysis phase, errors can occur due to misspelled tokens, unrecognized characters, etc. These errors are mostly the typing errors.
    - In syntax analysis phase, errors can occur due to the syntactic violation of the language.
    - In Semantic analysis and intermediate code generation phase, errors can occur due to incompatibility of operands type for an operator.
    - In code optimization phase, errors can occur during the control flow analysis due to some unreachable statements.
    - In code generation phase, errors can occurs due to the incompatibility with the computer architecture during the generation of machine code. For example, a constant created by compiler may be too large to fit in the word of the target machine.
    - In symbol table, errors can occur during the bookkeeping routine, due to the multiple declaration of an identifier with ambiguous attributes.

Error Handler

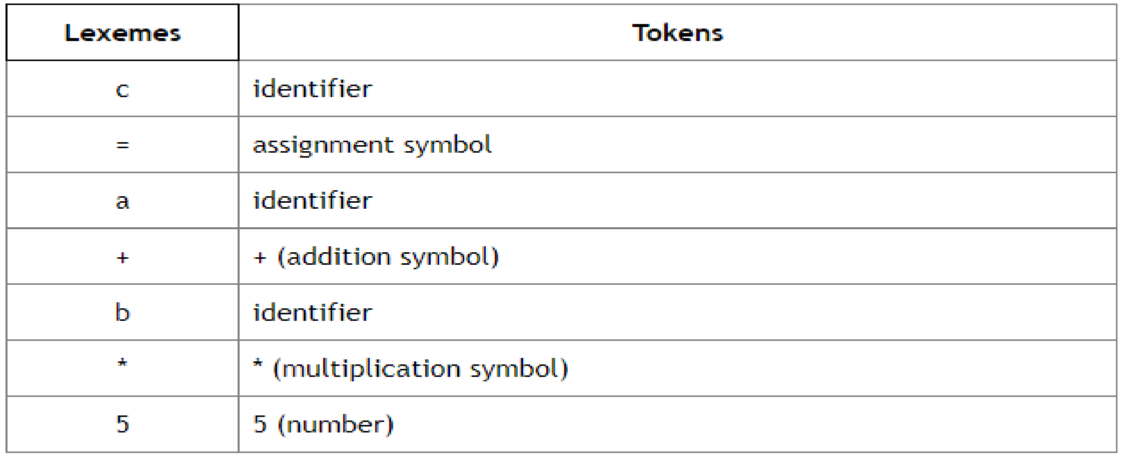
- Error detection and reporting/handling of errors are important functions of the compiler.
- Whenever an error is encountered during the compilation of the source program, an error handler is invoked.
- If possible Error handler handles the error such as, Possibly error handler may add some characters to make correct tokens, automatic type conversion, etc.
- Otherwise, generates a suitable error reporting message regarding the error encountered.
- The error reporting message allows the programmer to find out the exact location of the error.

Error Handler

- Error detection and reporting/handling of errors are important functions of the compiler.
- Whenever an error is encountered during the compilation of the source program, an error handler is invoked.
- If possible Error handler handles the error such as, Possibly error handler may add some characters to make correct tokens, automatic type conversion, etc.
- Otherwise, generates a suitable error reporting message regarding the error encountered.
- The error reporting message allows the programmer to find out the exact location of the error.

Translation for simple expression: Example

- **Lexical analysis:**
  - Input: stream of characters
  - Output: Token
  - Token Template: <token-name, attribute-value>
  - For example, c=a+b*5;
  - Lexemes and tokens

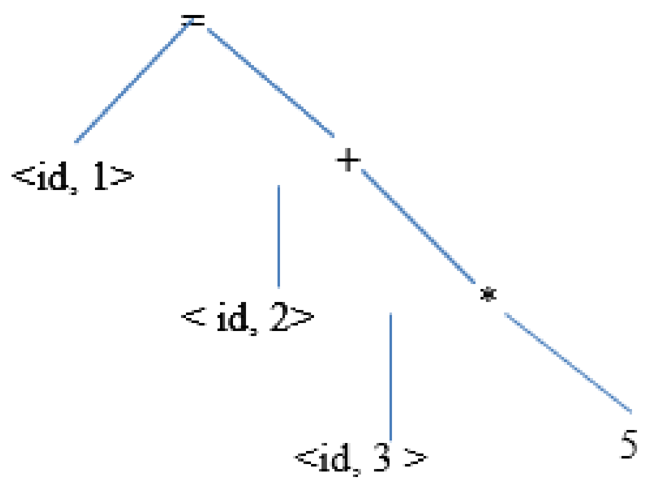


Lexemes	Tokens
c	identifier
=	assignment symbol
a	identifier
+	+ (addition symbol)
b	identifier
*	* (multiplication symbol)
5	5 (number)

  - Hence, <id, 1><=>< id, 2>< +><id, 3 >< * >< 5>

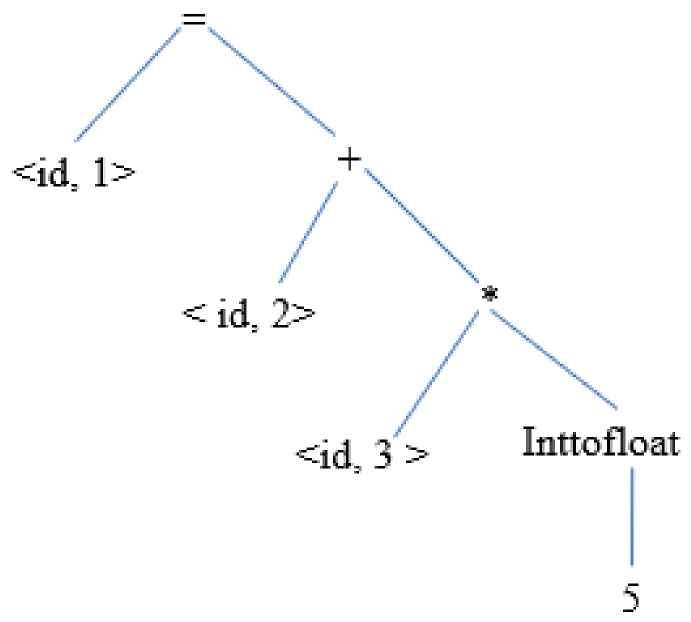
Translation for simple expression: Example

- **Syntax Analysis:**
  - Input: Tokens
  - Output: Syntax tree
  - For example: <id, 1><=>< id, 2>< +><id, 3 >< * >< 5>
  - Syntax tree:



Translation for simple expression: Example

- **Semantic Analysis:**
  - Input: Syntax tree
  - Output: Syntax tree with type information
  - Example:



Translation for simple expression: Example

- **Intermediate code generation:**
  - t_1 = inttofloat (5)
  - $t_2 = id_3$* tl
  - $t_3 = id_2 + t_2$
  - $id_1 = t_3$
- **Code Optimization:**

  $t_1 = id_3$* 5.0
  $id_1 = id_2 + t_1$
- **Code Generation:**

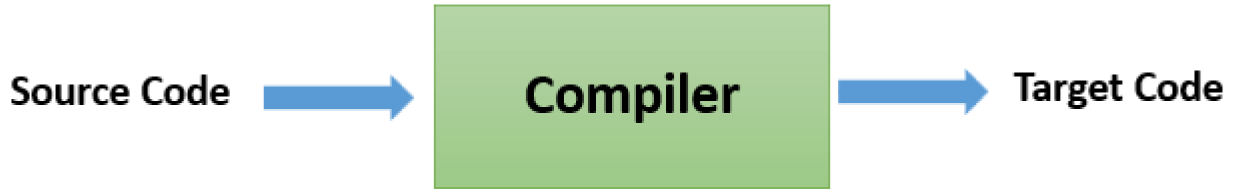
  LDF R_2, id_3
  MULF R_2, R2, # 5.0
  LDF R_1, id_2
  ADDF R_1, R1, R_2
  STF id_1, R_1

Types of Compiler

- Several phases are generally implemented in single pass, reading an input and writing an output.
- Few passes speed up the processing, more number of passes though decrease the compilation speed, increases the enhancement capability.
- Types of compilers based on passes:
  - Single Pass Compilers
  - Two Pass Compilers
  - Multi-pass Compilers

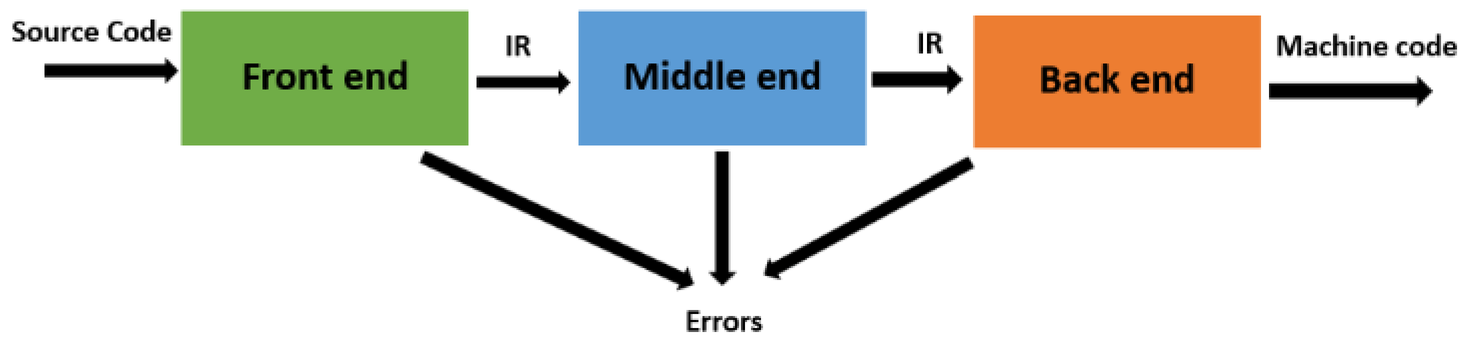
Types of Compiler

- **Single Pass Compiler:**
  - In single pass Compiler source code directly transforms into machine code.
  - For example, Pascal language.



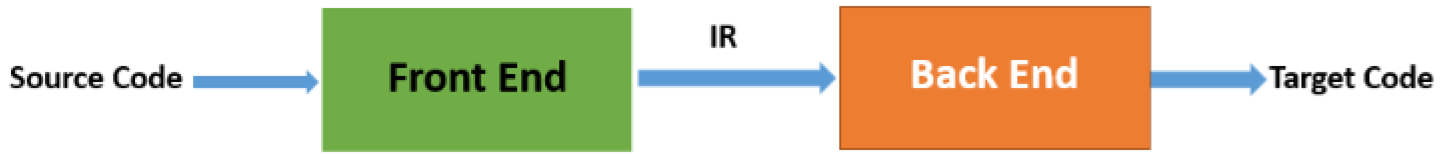
Types of Compiler

- **Multi-pass Compilers (Wider compiler)**
- The multi-pass compiler processes the source code of a program several times.
- It develops multiple intermediate codes.
- All of these multi-pass take the output of the previous pass as an input.



Types of Compiler

- **Two Pass Compiler:**
  - Two pass Compiler is divided into two sections, viz.
    - **Front end:** It maps legal code into Intermediate Representation (IR).
    - **Back end:** It maps IR onto the target machine



Compiler construction tools (Compiler-Compilers)

- Scanner Generators:
    - Generates lexers automatically i.e. tokenizes the source code. E.g. flex, lex, etc.
- Parser Generators:
    - Produces syntax analyzers automatically. E.g. bison, yacc, JavaCC, etc.
- Syntax Directed Translation Engines:
    - Walks through parse tree and generates intermediate code. E.g. bison, yacc, JavaCC, etc.
- Automatic Code Generators:
    - Generates target code automatically from intermediate code.
- Data Flow Engines:
    - Optimizers using data flow analysis.

Preprocessor

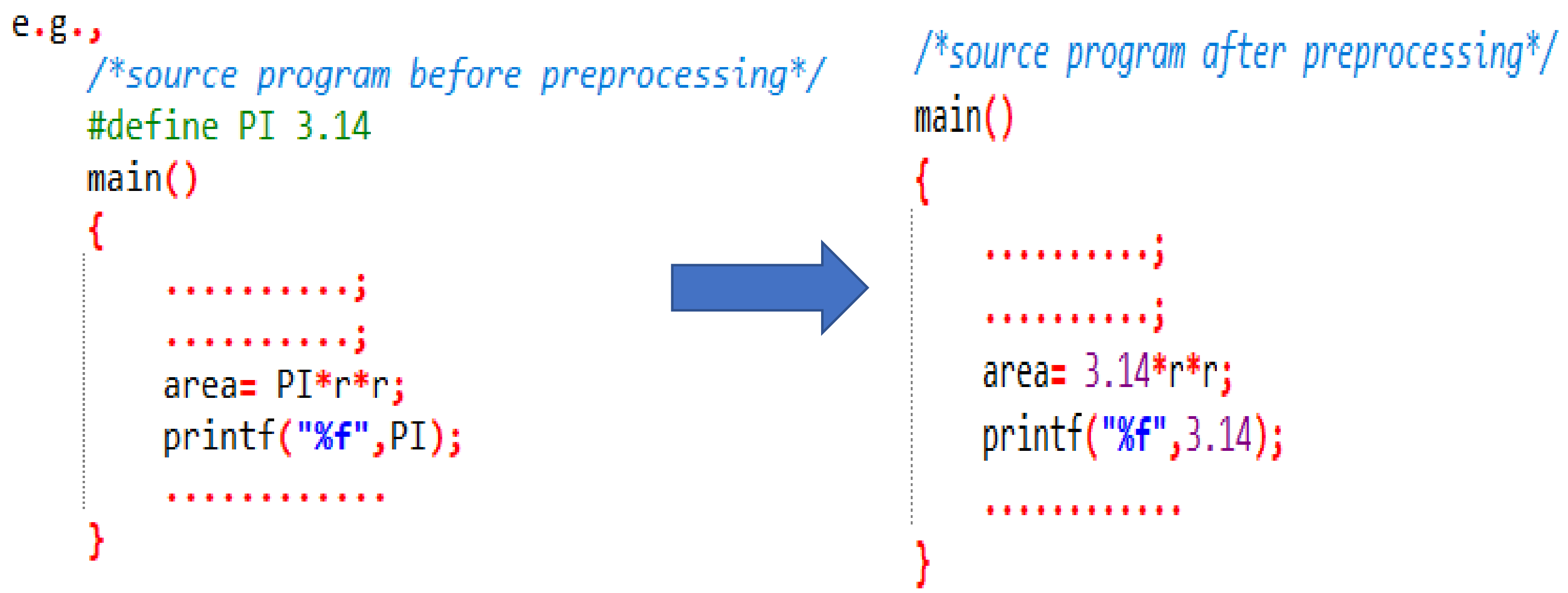
- Compiler component responsible for processing of preprocessor directives present in the source program
- Such processing includes:
  - Inclusion of header file into source program file
  - Macro expansion

Macro

- A macro is an abbreviation, which stands for some related lines of code. Macros are useful for the following purposes:
  - To simplify and reduce the amount of repetitive coding
  - To reduce errors caused by repetitive coding
  - To make an assembly program more readable.
- A macro consists of name, set of formal parameters and body of code. The use of macro name with set of actual parameters is replaced by some code generated by its body. This is called macro expansion.
- Macros allow a programmer to define pseudo operations, typically operations that are generally desirable, are not implemented as part of the processor instruction, and can be implemented as a sequence of instructions. Each use of a macro generates new program instructions, the macro has the effect of automating writing of the program.

Macro

- **Example1** : Syntax: #define IDENTIFIER string
  - If this statement is included in the program at the beginning, then the preprocessor replaces every occurrence of the IDENTIFIER in the source by the string



```
e.g.,
    /*source program before preprocessing*/
    #define PI 3.14
    main()
    {
        ..........;
        ..........;
        area= PI*r*r;
        printf("%f",PI);
        ...........
    }
```

```
/*source program after preprocessing*/
main()
{
    .........;
    .........;
    area= 3.14*r*r;
    printf("%f",3.14);
    ...........
}
```

Macro

Before macro expansion:

```
#define max(a,b) a>b?a:b

Main ()

{

        int x , y;

        x=4; y=6;

        z = max(x, y);

}
```

After macro expansion:

```
#define max(a,b) a>b?a:b
main()
{
        int x , y;
        x=4; y=6;
        z = x>y?x:y;
}
```

- The above program was written using C programming statements. Defines the macro max, taking two arguments a and b. This macro may be called like any C function, using identical syntax. Therefore, after pre-processing Becomes z = x>y ? x: y; After macro expansion, the whole code would appear like this.

Homework

- What is compiler? How it differ from interpreter? Explain.
- What are the phases of a compiling a program? Explain the function of each phase in brief.
- Symbol table is necessary for compiler construction. Justify your statement with example.
- What is translator? Discuss the role of various phases of the compiler in the translation of source program to object code.
- Discuss the action taken by every phase of the compiler on the following string: A=B* C+D/E.
- Write the difference between syntax and semantics of the languages, also give the example in C language.
- What are various types of compiler? Analyze the advantages and disadvantages for single and multi-pass compilers.
- Discuss the role of machine architecture in compiler design, also give the pro and cons for virtual machine and real machine architecture.

Thank You !