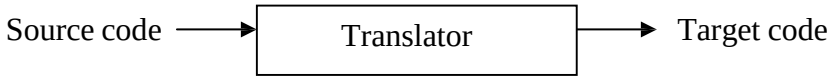
UNIT I- INTRODUCTION

INRODUCTION TO COMPILING

Translator:

It is a program that translates one language to another.



Types of Translator:

1. Interpreter
2. Compiler
3. Assembler

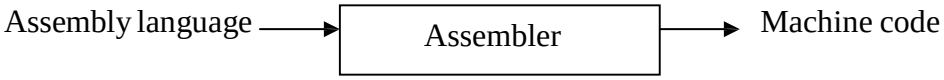
1. Interpreter:

It is one of the translators that translate high level language to low level language.

During execution, it checks line by line for errors.
Example: Basic, Lower version of Pascal.

2. Assembler:

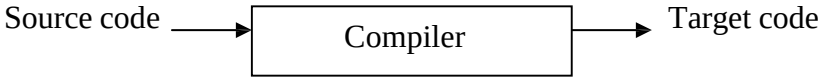
It translates assembly level language to machine code.



Example: Microprocessor 8085, 8086.

3. Compiler:

It is a program that translates one language (source code) to another language (target Code).



It executes the whole program and then displays the errors. Example: C, C++, COBOL, higher version of Pascal.

#	COMPILER	INTERPRETER
1	Compiler works on the complete program at once. It takes the **entire program** as input.	Interpreter program works line-by-line. It takes **one statement at a time** as input.
2	Compiler generates intermediate code, called the o**bject code or machine code.**	Interpreter does not generate intermediate object code or machine code.
3	Compiler executes conditional control statements (like if-else and switch-case) and logical constructs **faster than interpreter.**	Interpreter execute conditional control statements at a much **slower speed**.
4	**Compiled programs take more memory** because the entire object code has to reside in memory.	Interpreter does not generate intermediate object code. As a result, **interpreted programs are more memory efficient.**
5	Compile once and run anytime. Compiled program does not need to be compiled every time.	Interpreted programs are interpreted line-by-line every time they are run.
6	Errors are reported after the **entire program is checked** for syntactical and other errors.	Error is reported as soon as the first error is encountered. Rest of the program will not be checked until the existing error is removed.
7	A compiled language is more difficult to debug.	Debugging is easy because interpreter stops and reports errors as it encounters them.
8	Compiler does not allow a program to run until it is completely error-free.	Interpreter runs the program from first line and stops execution only if it encounters an error.
9	Compiled languages are more efficient but difficult to debug.	Interpreted languages are less efficient but easier to debug. This makes such languages an ideal choice for new students.
10	**Examples** of programming languages that use compilers: C, C++, COBOL	**Examples** of programming languages that use interpreters: BASIC, Visual Basic, Python, Ruby, PHP, Perl, MATLAB, Lisp

Kinds of compiler

There are several major kinds of compilers:

• **Native Code Compiler:** Translates source code into hardware (assembly or machine code) instructions. Example: gcc.

• **Virtual Machine Compiler:**Translates source code into an abstract machine code, for execution by a virtual machine interpreter. Example: javac.

• **JIT Compiler :** Translates virtual machine code to native code. Operates within a virtual machine. Example: Sun's HotSpot java machine.

• **Preprocessor:** Translates source code into simpler or slightly lower level source code, for compilation by another compiler. Examples: cpp, m4.

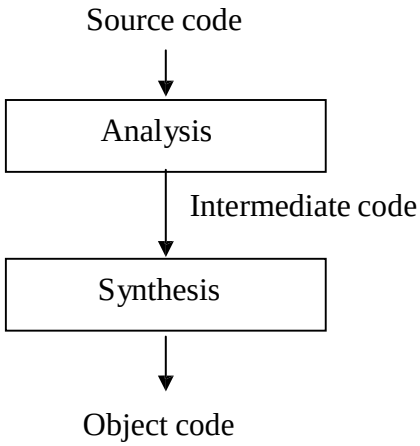
• **Pure interpreter :** Executes source code on the fly, without generating machine code. Example: Lisp.

PARTS OF COMPILATION

There are 2 parts to compilation:
1. Analysis
2. Synthesis

Analysis part breaks down the source program into constituent pieces and creates an intermediate representation of the source program.

Synthesis part constructs the desired target program from the intermediate representation.



Software tools used in Analysis part:

1) **Structure editor:**
- Takes as input a sequence of commands to build a source program.
- The structure editor not only performs the text-creation and modification functions of an ordinary text editor, but it also analyzes the program text, putting an appropriate hierarchical structure on the source program.
- For example , it can supply key words automatically - while …. do and begin….. end.

2) Pretty printers :
- A pretty printer analyzes a program and prints it in such a way that the structure of the program becomes clearly visible.
- For example, comments may appear in a special font.

3) Static checkers :
- A static checker reads a program, analyzes it, and attempts to discover potential bugs without running the program.
- For example, a static checker may detect that parts of the source program can never be executed.

4) Interpreters :
- Translates from high level language (BASIC, FORTRAN, etc..) into machine language.
- An interpreter might build a syntax tree and then carry out the operations at the nodes as it walks the tree.

- Interpreters are frequently used to execute command language since each operator executed in a command language is usually an invocation of a complex routine such as an editor or complier.

ANALYSIS OF THE SOURCE PROGRAM

Analysis consists of 3 phases:

Linear/Lexical Analysis:

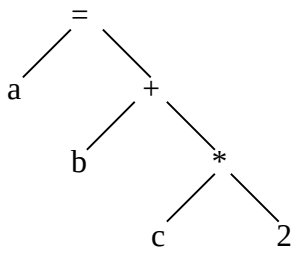
- It is also called scanning. It is the process of reading the characters from left to right and grouping into tokens having a collective meaning.

- For example, in the assignment statement a=b+c*2, the characters would be grouped into the following tokens:
  i) The identifier1 'a'
  ii) The assignment symbol (=)
  iii) The identifier2 'b'
  iv) The plus sign (+)
  v) The identifier3 'c'
  vi) The multiplication sign (*)
  vii) The constant '2'

Syntax Analysis :

- It is called parsing or hierarchical analysis. It involves grouping the tokens of the source program into grammatical phrases that are used by the compiler to synthesize output.

- They are represented using a syntax tree as shown below:



- A **syntax tree** is the tree generated as a result of syntax analysis in which the interior nodes are the operators and the exterior nodes are the operands.

- This analysis shows an error when the syntax is incorrect.

Semantic Analysis :

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

PHASES OF COMPILER

A Compiler operates in phases, each of which transforms the source program from one representation into another. The following are the phases of the compiler:

Main phases:
1) Lexical analysis
2) Syntax analysis
3) Semantic analysis
4) Intermediate code generation
5) Code optimization
6) Code generation

Sub-Phases:
1) Symbol table management
2) Error handling

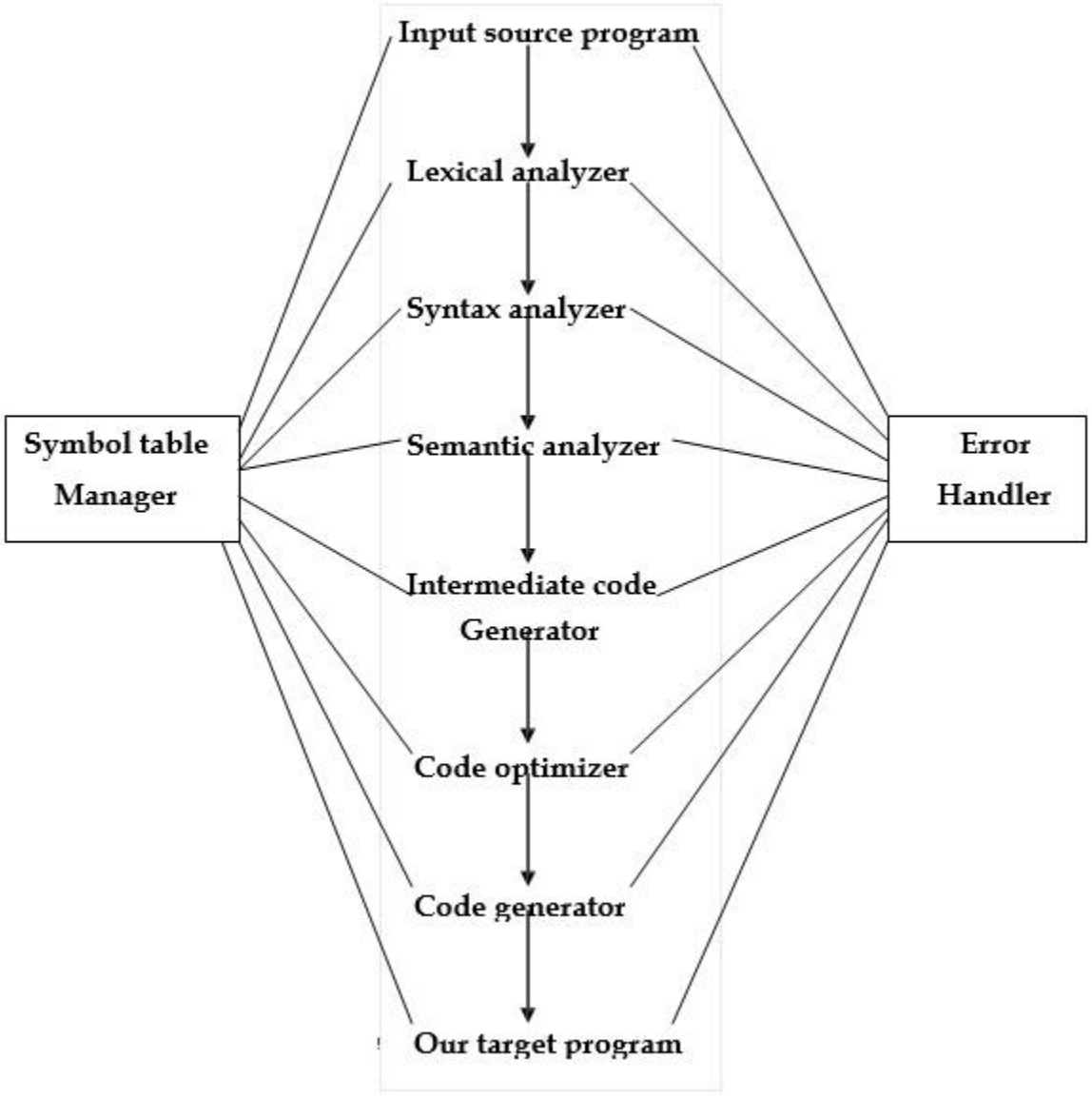


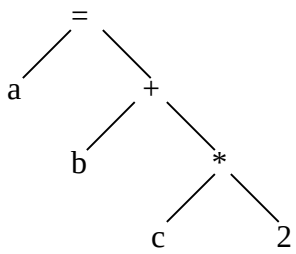
Fig: Phases of Compiler

LEXICAL ANALYSIS:

- It is the first phase of the compiler. It gets input from the source program and produces tokens as output.
- It reads the characters one by one, starting from left to right and forms the tokens.
- **Token** : It represents a logically cohesive sequence of characters such as keywords, operators, identifiers, special symbols etc.
  Example: a + b = 20
  Here, a,b,+,=,20 are all separate tokens.
  Group of characters forming a token is called the **Lexeme**.
- The lexical analyser not only generates a token but also enters the lexeme into the symbol table if it is not already there.

SYNTAX ANALYSIS:

- It is the second phase of the compiler. It is also known as parser.
- It gets the token stream as input from the lexical analyser of the compiler and generates syntax tree as the output.
- Syntax tree:
  It is a tree in which interior nodes are operators and exterior nodes are operands.
- Example: For a=b+c*2, syntax tree is



SEMANTIC ANALYSIS:

- It is the third phase of the compiler.
- It gets input from the syntax analysis as parse tree and checks whether the given syntax is correct or not.
- It performs type conversion of all the data types into real data types.

INTERMEDIATE CODE GENERATION:

- It is the fourth phase of the compiler.
- It gets input from the semantic analysis and converts the input into output as intermediate code such as three-address code.
- The three-address code consists of a sequence of instructions, each of which has atmost three operands.
  Example: t1=t2+t3

CODE OPTIMIZATION:

- It is the fifth phase of the compiler.
- 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 so that faster-running machine code will result.
- During the code optimization, the result of the program is not affected.
- 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.
  - moving code outside the loop.
  - removal of unwanted temporary variables.

CODE GENERATION:

- It is the final phase of the compiler.
- It gets input from code optimization phase and produces the target code or object code as result.
- Intermediate instructions are translated into a sequence of machine instructions that perform the same task.
- The code generation involves
  - allocation of register and memory
  - generation of correct references
  - generation of correct data types
  - generation of missing code

SYMBOL TABLE MANAGEMENT:

- Symbol table is used to store all the information about identifiers used in the program.
- It is a data structure containing a record for each identifier, with fields for the attributes of the identifier.
- It allows to find the record for each identifier quickly and to store or retrieve data from that record.
- Whenever an identifier is detected in any of the phases, it is stored in the symbol table.

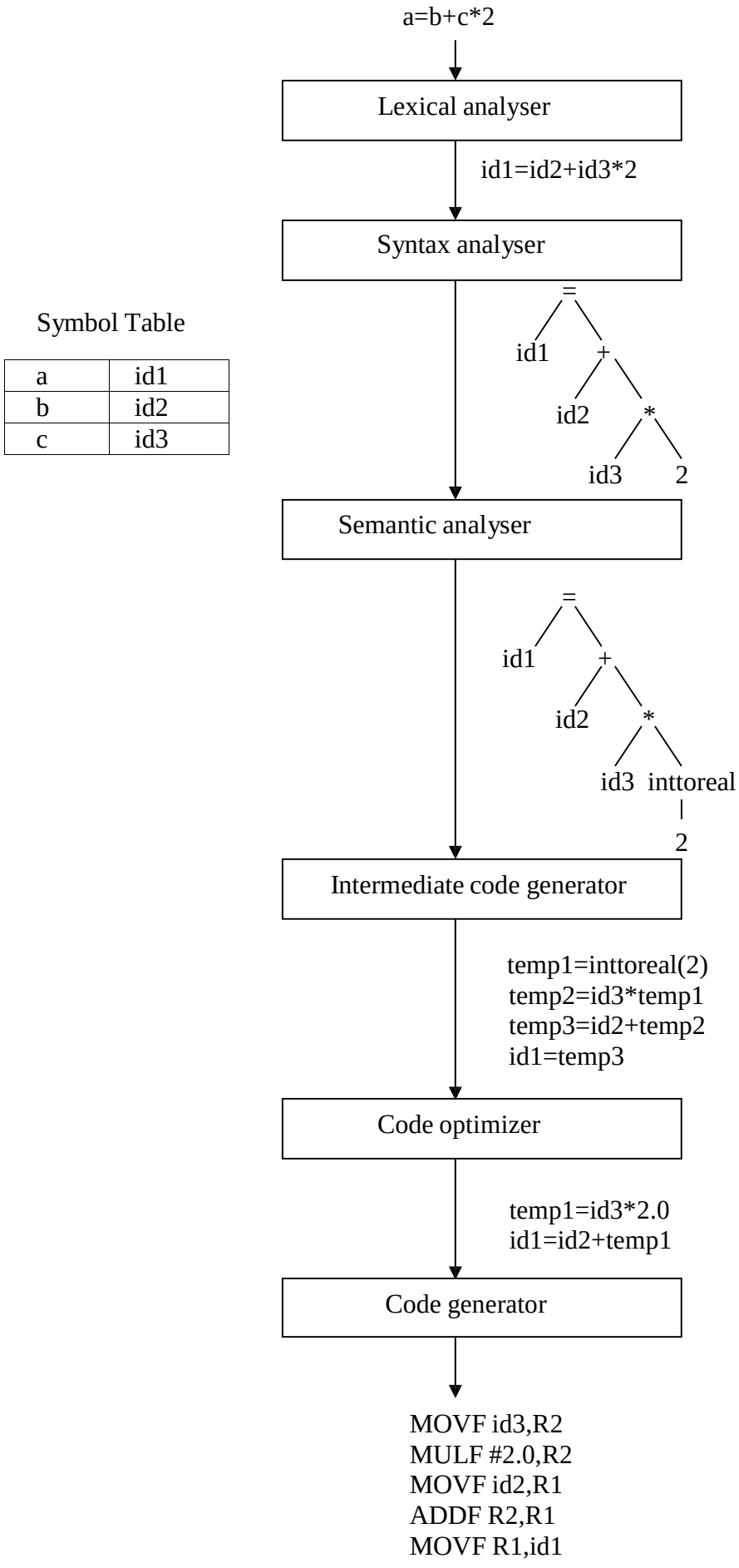
Example:

Insert("dist",id) // insert a symbol table entry associating the string "dist" with token type "id"
Lookup("dist") // an occurrence of string "dist" can be looked up in the symbol table. If found, the reference to the "id" is returned else lookup return 0.

ERROR HANDLING:

- Each phase can encounter errors. After detecting an error, a phase must handle the error so that compilation can proceed.
- In lexical analysis, errors occur in separation of tokens.
- In syntax analysis, errors occur during construction of syntax tree.
- In semantic analysis, errors occur when the compiler detects constructs with right syntactic structure but no meaning and during type conversion.
- In code optimization, errors occur when the result is affected by the optimization.
- In code generation, it shows error when code is missing etc.

To illustrate the translation of source code through each phase, consider the statement a=b+c*2. The figure shows the representation of this statement after each phase:


a=b+c*2
Lexical analyser
id1=id2+id3*2
Syntax analyser
=
id1
+
id2
*
id3
2
Symbol Table
a id1
b id2
c id3
Semantic analyser
=
id1
+
id2
*
id3
inttoreal
2
Intermediate code generator
temp1=inttoreal(2)
temp2=id3*temp1
temp3=id2+temp2
id1=temp3
Code optimizer
temp1=id3*2.0
id1=id2+temp1
Code generator
MOVF id3,R2
MULF #2.0,R2
MOVF id2,R1
ADDF R2,R1
MOVF R1,id1

COUSINS OF COMPILER

1. Preprocessor
2. Assembler
3. Loader and Link-editor

PREPROCESSOR

A preprocessor is a program that processes its input data to produce output that is used as input to another program. The output is said to be a preprocessed form of the input data, which is often used by some subsequent programs like compilers.

They may perform the following functions :

1. Macro processing
2. File Inclusion
3. Rational Preprocessors
4. Language extension

1. Macro processing:

A macro is a rule or pattern that specifies how a certain input sequence should be mapped to an output sequence according to a defined procedure. The mapping process that instantiates a macro into a specific output sequence is known as macro expansion.

2. File Inclusion:

Preprocessor includes header files into the program text. When the preprocessor finds an #include directive it replaces it by the entire content of the specified file.

3. Rational Preprocessors:

These processors change older languages with more modern flow-of-control and data-structuring facilities.

4. Language extension :

These processors attempt to add capabilities to the language by what amounts to built-in macros. For example, the language Equel is a database query language embedded in C.

ASSEMBLER

Assembler creates object code by translating assembly instruction mnemonics into machine code. There are two types of assemblers:

- One-pass assemblers go through the source code once and assume that all symbols will be defined before any instruction that references them.
- Two-pass assemblers create a table with all symbols and their values in the first pass, and then use the table in a second pass to generate code.

LINKER AND LOADER

A **linker** or **link editor** is a program that takes one or more objects generated by a compiler and combines them into a single executable program.

Three tasks of the linker are :

1. Searches the program to find library routines used by program, e.g. printf(), math routines.
2. Determines the memory locations that code from each module will occupy and relocates its instructions by adjusting absolute references
3. Resolves references among files.

A **loader** is the part of an operating system that is responsible for loading programs in memory, one of the essential stages in the process of starting a program.

GROUPING OF THE PHASES

Compiler can be grouped into front and back ends:

- **Front end**: analysis (machine independent)
  These normally include lexical and syntactic analysis, the creation of the symbol table, semantic analysis and the generation of intermediate code. It also includes error handling that goes along with each of these phases.

- **Back end**: synthesis (machine dependent)
  It includes code optimization phase and code generation along with the necessary error handling and symbol table operations.

Compiler passes

A collection of phases is done only once (single pass) or multiple times (multi pass)

- Single pass: usually requires everything to be defined before being used in source program.
- Multi pass: compiler may have to keep entire program representation in memory.

Several phases can be grouped into one single pass and the activities of these phases are interleaved during the pass. For example, lexical analysis, syntax analysis, semantic analysis and intermediate code generation might be grouped into one pass.

COMPILER CONSTRUCTION TOOLS

These are specialized tools that have been developed for helping implement various phases of a compiler. The following are the compiler construction tools:

1) **Parser Generators:**
-These produce syntax analyzers, normally from input that is based on a context-free grammar.
-It consumes a large fraction of the running time of a compiler.
-Example-YACC (Yet Another Compiler-Compiler).

2) **Scanner Generator:**
-These generate lexical analyzers, normally from a specification based on regular expressions.
-The basic organization of lexical analyzers is based on finite automation.

3) **Syntax-Directed Translation:**
-These produce routines that walk the parse tree and as a result generate intermediate code.
-Each translation is defined in terms of translations at its neighbor nodes in the tree.

4) **Automatic Code Generators:**
-It takes a collection of rules to translate intermediate language into machine language. The rules must include sufficient details to handle different possible access methods for data.

5) **Data-Flow Engines:**
-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.