CS202 – System Software

Dr. Manish Khare

Lecture 3

Major Components of a Programming System

- The Major Components of a programming system is:
  - Operating system
  - Language translators
    - Compilers
    - Interpreters
    - Assemblers
    - Preprocessors
    - Loaders
    - Linkers
    - Macro processors.

Operating System

- It is the most significant system program that performs as an interface among the users and the system. It makes the computer simpler to utilize.
- It offers an interface that is more comprehensible than the fundamental hardware.
- The functions of OS are:
  - Process management,
  - Memory management,
  - Resource management,
  - Secondary Storage management
  - User Management
  - Security Management.

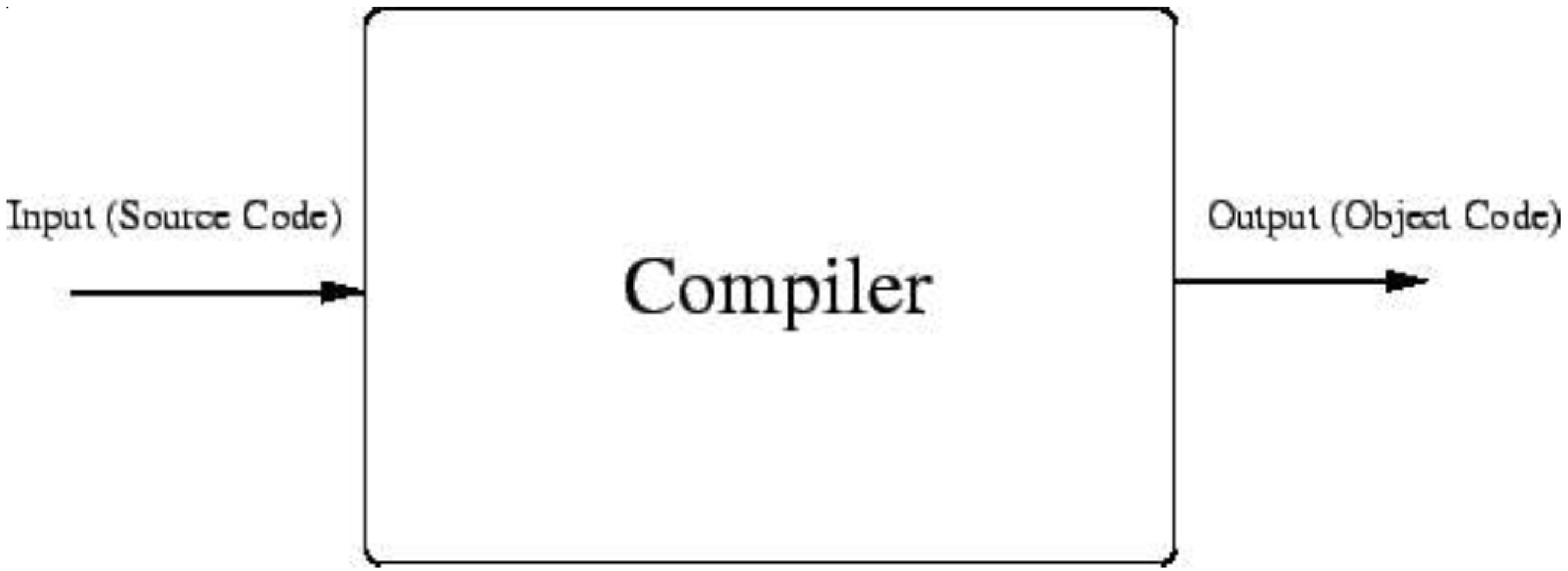
Language Translators

- Language translator is the program that takes an input program in one language and generates an output in another language.

Source Program → Language Translator → Object Program

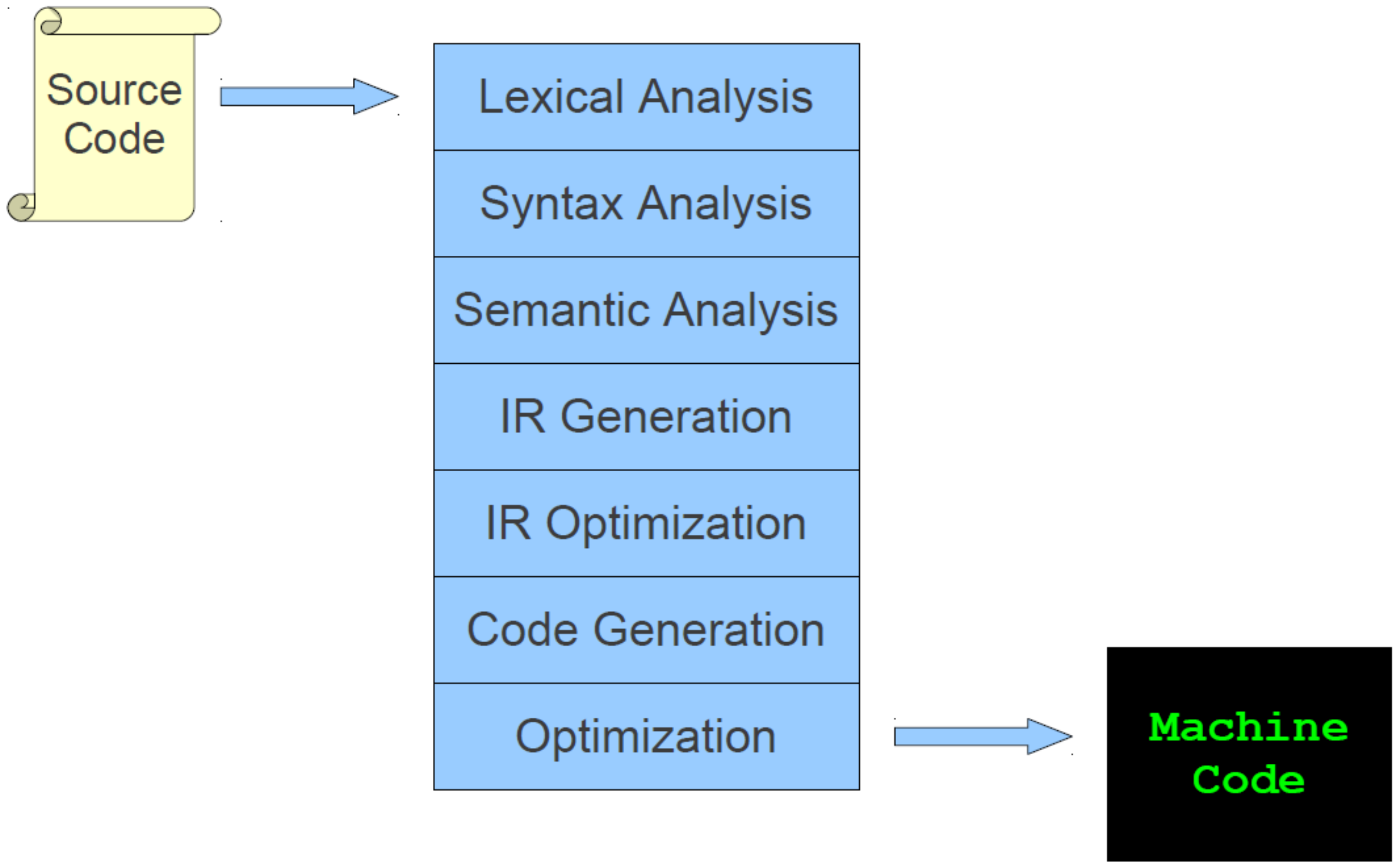
Compiler

- A compiler is a system software that converts a source language program unto an equivalent target language program.
- It also validates the input program to be conforming to the source language specification.
- There are some fundamental properties for a compiler.
    - A compiler must be error-free.
    - A compiler must at all times terminate, regardless of what the input appears like.

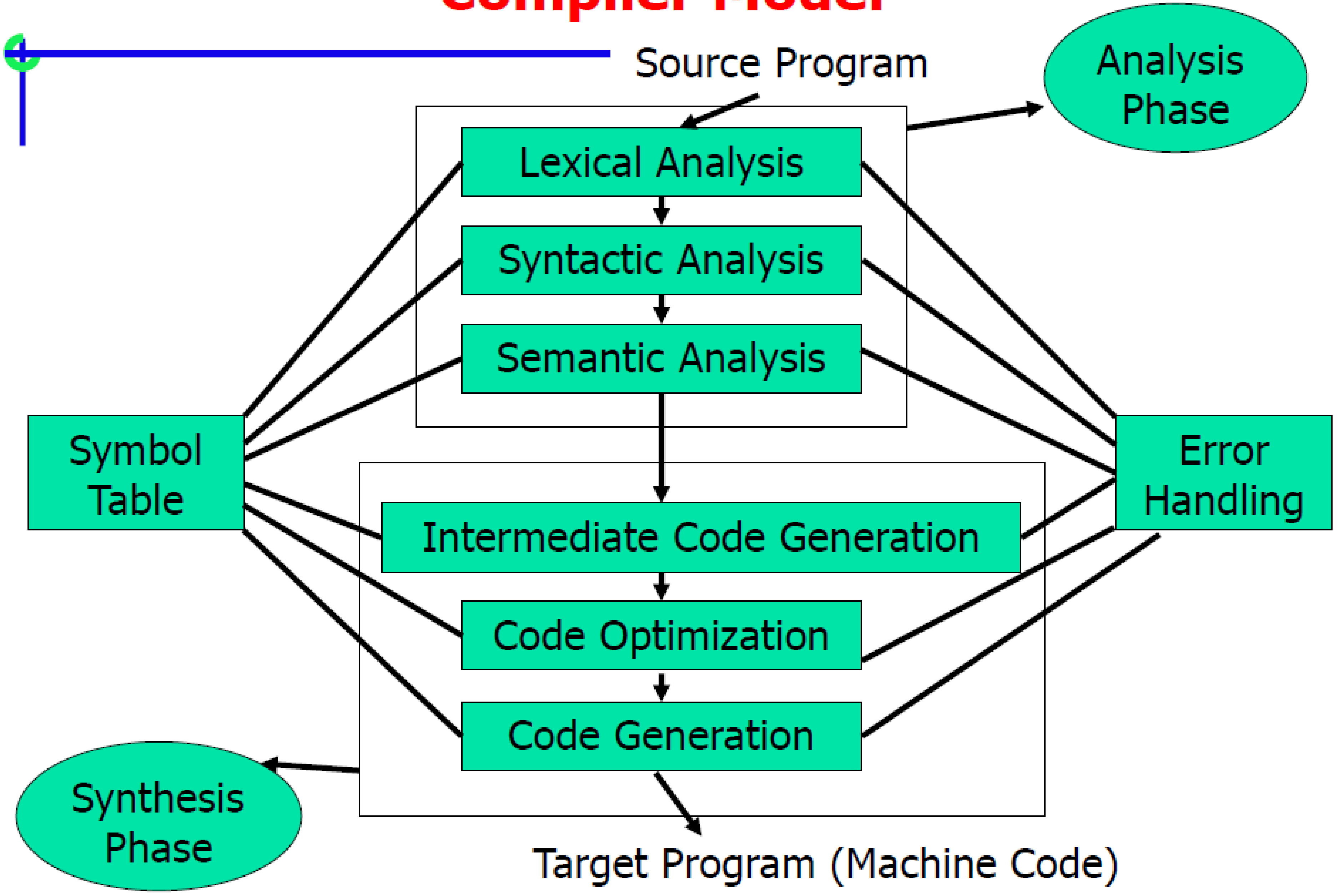

Input (Source Code)
Compiler
Output (Object Code)

- A compiler can be conceptually divided into a number of phases:
  - Lexical analysis
  - Syntax analysis
  - Semantic analysis
  - Intermediate code generation
  - Target Code generation
  - Symbol Table management
  - Error handling and recovery

The Structure of a Modern Compiler



Compiler Model



The Phases of a compiler

- Lexical analysis phase scan the input and separates the characters of source language that are logically one (e.g. int), these group of characters are called tokens. The tokens are keywords, identifiers, constants, operator symbols or punctuation symbols. These tokens are passed to next phase i.e. syntax analyzer.

- Syntax analyzer groups these token into syntactic structures called expressions. These are in the form of a tree called parse tree whose leaves are tokens.

- Semantic analyzer is optional phase of compiler. Here we make syntax tree of tokens with its attributes.

The Phases of a compiler

- The intermediate code generator uses syntactic structure to create stream if simple instructions. These can be in the form of simple statements. These can be in the form of simple statements or some of the intermediate forms like postfix, triples and quadruples etc. depending upon the languages, its strength and for which machine it will generate the code. It can be mixture of two or more forms also. It is different from machine code, as it does not require registers.

- Code optimization is the optional phase of compiler which is executed only if the intermediate code needs to be optimized. Its purpose is to remove redundancies and reduce the time of execution, in order to increase the efficiency of execution of source program.

The Phases of a compiler

- The final phase, code generation produces the object code, assigning memory and allocating registers for computations
- Table management as a bookkeeping process which keeps tracks of essential information's like name, attributes, data types etc.
- Error handlers acts when some error occurs in program, like syntax error, therefore, proper diagnostics should be called to locate and warn about the error with appropriate message and error point, like line number at which error has occurred.
- Both table management and error handler cooperate and interact with each phase of compilers

The Phases of a compiler

- **Lexical analyzer**
- In a compiler, linear analysis is called lexical analyzer or scanning. For ex. Lexical analysis of the characters in the assignment statement

position=initial + rate * 60

will be grouped into the following tokens

- The identifier 1 ‘position’
- The assignment symbol ‘=’
- The identifier 2 ‘identifier’
- The plus operator ‘+’ sign
- The identifier 3 ‘rate’
- The multiplication operator ‘*’ sign
- The number ‘60’

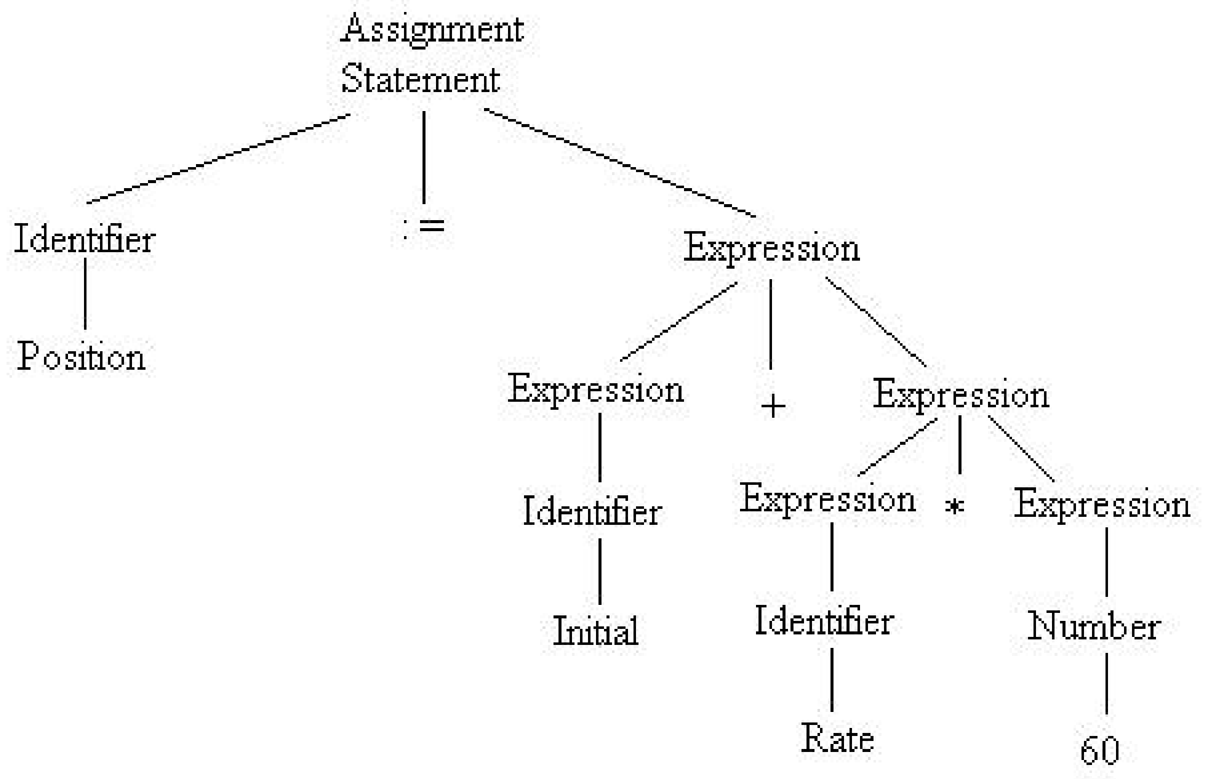
The blanks separating the characters of these token would normally be eliminated during lexical analysis

The Phases of a compiler

- **Syntax analyzer**
- Syntax analysis is also called ‘parsing’. It involves grouping the tokens of the source program into grammatical phrases that are used by the compiler to synthesize output. Usually the grammatical phrases of the source program are represented by a parse tree.
- Parse tree is a tree whose leaves are labeled with tokens and each of its parent-children portion forms a rule tree that graphically represents a rule.

The Phases of a compiler

- For expression [position=initial + rate * 60]

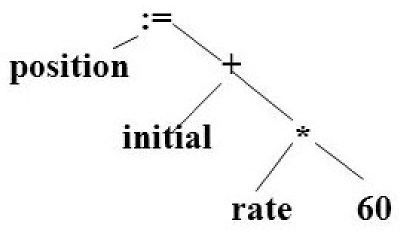


Parse Tree for Position: = Initial + Rate *60

The Phases of a compiler

- The hierarchical structure of a program is usually expressed by recursive rules. For ex. We might have the following rules as part of definition of expression
  - Any identifier is an expression
  - Any number is an expression
  - If expression1 and expression2 are expression, then expression1+expression2, expression1*expression2 are also expression.

- A more common internal representation of this syntactic structure is given by syntax tree



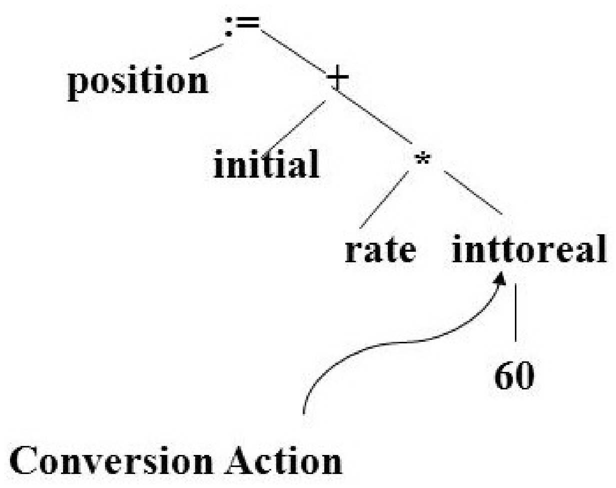
The Phases of a compiler

Semantic analysis

- The semantic analysis phase checks the source program for semantic errors and gathers type information for the subsequent code-generation phase.
- It uses the hierarchical structure determined by the syntax-analysis phase to identify the operators and operands of expressions and 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.

The Phases of a compiler

- Suppose for ex., that all identifier in parse tree of [position=initial + rate * 60], have been declared to be reals and that ’60’ by itself is assumed to be an integer. Type checking of above figure reveals that ‘*’ is applied to a real, rate and an integer 60. the general approach is to convert the integer into a real. This has been achieved in figure.



The Phases of a compiler

- **Intermediate code generation**
- After syntax and semantic analysis, some compilers generates an explicit intermediate representation of the source program.
- We can think of this intermediate representation as a program for an abstract machine.
- This intermediate representation should have two important properties, it should be easy to produce and easy to translate into the target program.
- The intermediate representation can have a variety of forms. In which one form is called ‘three-address code’.
- Other forms – Triples, Quadruples

The Phases of a compiler

- Three address code consists of a sequence of instruction, each of which has at most three operands. The source program [position=initial + rate * 60] might appear in three address code

  - temp1=int to real (60)
  - temp2=id3*temp1
  - temp3=id2+temp2
  - id1=temp3

The Phases of a compiler

- The intermediate form has several properties as
  - Each three address instruction has at most one operator in addition to the assignment.
  - The compiler must generate a temporary name to hold the value computed by the each instruction.

The Phases of a compiler

- **Code optimization**
- This is optional phase and attempts to improve the intermediate code. So that faster running machine code will result, some optimization are trivial.
- For ex., a natural algorithm generates the intermediate code using an instruction for each operator in the three representation after semantic analysis, even through there is better way to perform the same calculation using the two instruction.
    - temp1=id3*60.0
    - id1=id2+temp1

The Phases of a compiler

- **Code generation**
- The final phase of the compiler is the generation of target code, consisting normally to relocatable machine code or assembly code.
- Memory locations are selected for each of the variables used by the program.
- Then intermediate code instruction are translated into a sequence of machine instruction that perform the same task.

The Phases of a compiler

- For ex., using register 1 & 2, the translation of the code of optimized code might become
    - MOV F id3,R2
    - MUL F #60.0, R2
    - MOV F id2, R1
    - ADD F R2,R1
    - MOV F R1,id1

The Phases of a compiler

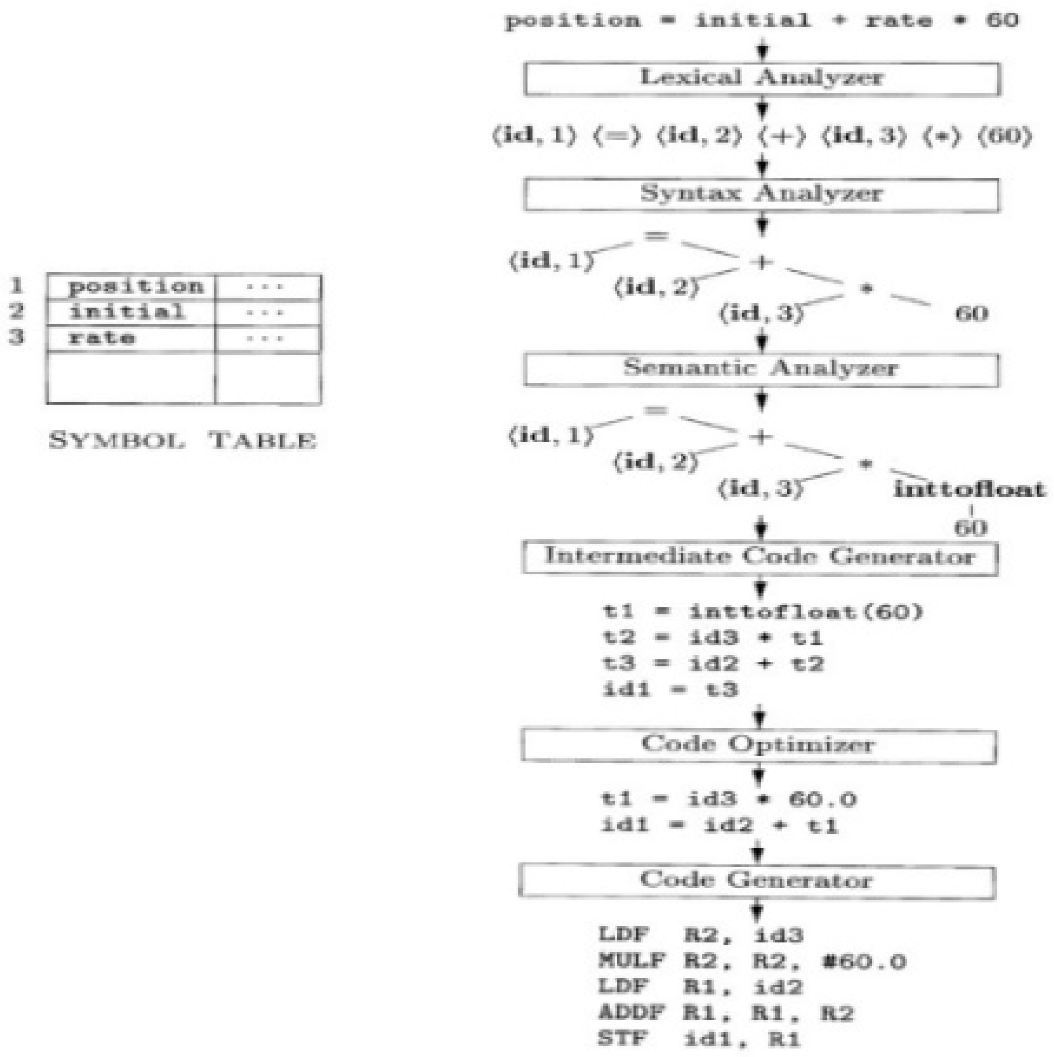
Symbol table manager

- Symbol table is a data structure containing a record for each identifier, with fields for the attributes of the identifier.
- Symbol table management is a mechanism that associates each identifier with relevant information, such as name, type and scope.
- The data structure allows us to find the record for each identifier quickly and to store or retrieve data from that record quickly.
- Most of this information is collected during analysis.

The Phases of a compiler

- **Error Detection**
- Each phase can encounter errors. However after detecting an error, a phase must some how deal with that error, so that compilation can proceed, allowing further errors in the source program to be detected.
- A compiler that stop when it find the first error is not as helpful as it could be.
- The syntax &semantic analysis phases usually handle a large fraction of the errors detectable by the compiler.
- the lexical phase can detect errors when the characters remaining in the input do not form any token of the language.

The Phases of a compiler



```
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
```

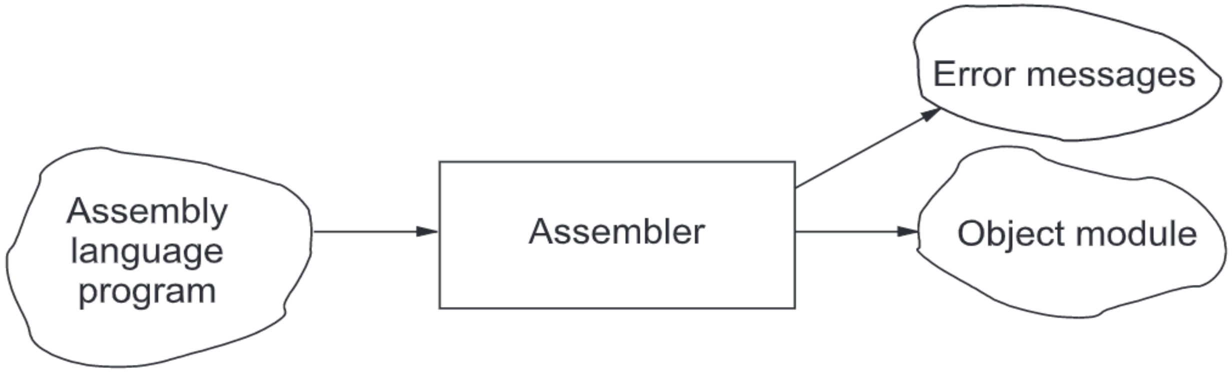
1	position	...
2	initial	...
3	rate	...

SYMBOL TABLE

Assembler

- An assembly language program is not directly executable. To be executed, first it is required to change it into its machine language equivalent.

- An Assembler is a program which is used to translate an assembly language program into its machine level language equivalent. The program in assembly language is termed as source code and its machine language equivalent is called object program.



- Assembler is a program that automatically converts the source program written in assembly language and to create as output an object code written in binary machine code.

- Given an assembly language program, an assembler checks for its syntactic correctness, ensures that all the variables and labels referred to in the source program are either defined inside the program, or have been marked to be supplied later (external definitions) in linking phase. The object file produced needs to have a definite format which is often dictated by the linker input format. An assembler may support different object file formats.

- Sometimes it is necessary to generate object code to be executed on a different CPU than that used in the machine for development. This is particularly true for designing new computer systems. For example, on a system that uses Pentium processor, we may use its assembler to generate object code for another system that uses, say Motorolla processor. This type of assembler is known as cross-assembler.

Linker

- Linking is the process of binding an exterior reference to the accurate link time address. An external reference is said to be uncertain until linking is performed for it. It is said to be resolved when its linking is accomplished.

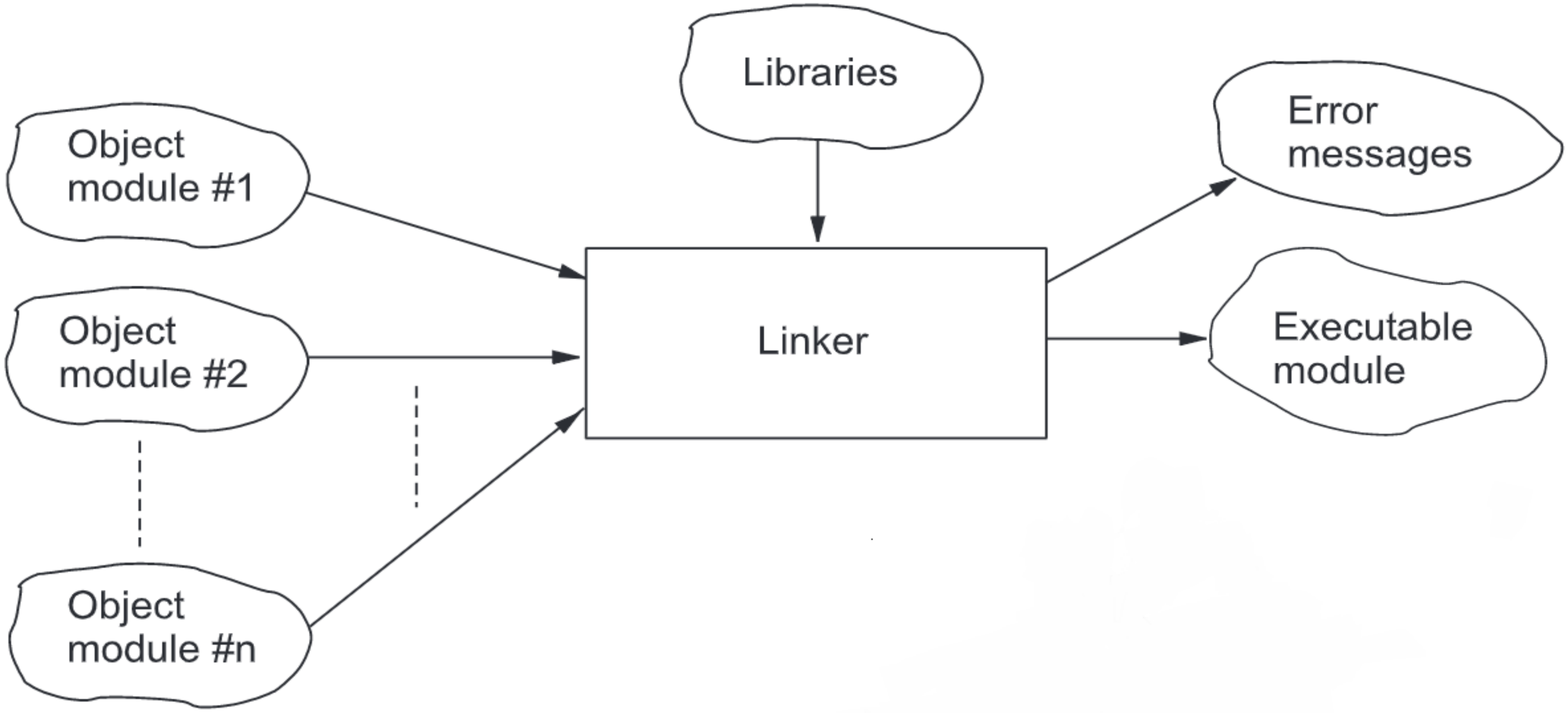
- In a large project, generally a team of people work together to design the entire system. They may develop their modules separately. Separate compilation is an important feature of programming languages designed for the development of large software systems.

- The compiler/assembler is able to compile/assemble each module of a program by itself with little or no information from other parts of the program. It is the responsibility of the linker to combine all such object files produced by the compiler/assembler and produce the executable form of the program.

- In the process of combining object modules, the linker needs to relocate them, since the codes are generated by the compiler /assembler, assuming the same fixed start address for all modules. The address sensitive portions within the code need to be modified properly.

- Another important responsibility of the linker is to adjust external references. As pointed out earlier, in a collaborative development environment, one module may use a variable/function which is defined in some other module. Thus, the undefined entity in the first module is an external symbol, which gets defined when the modules are combined together at the time of linking.

- The linker needs to fill up the blanks left by the assembler/compiler regarding all these external references. Some of the external symbols may need some library elements to be included for resolving them. For example, the basic input/output routines, mathematical functions, etc. come in the form of libraries.


Libraries
Object module #1
Object module #2
Object module #n
Linker
Error messages
Executable module

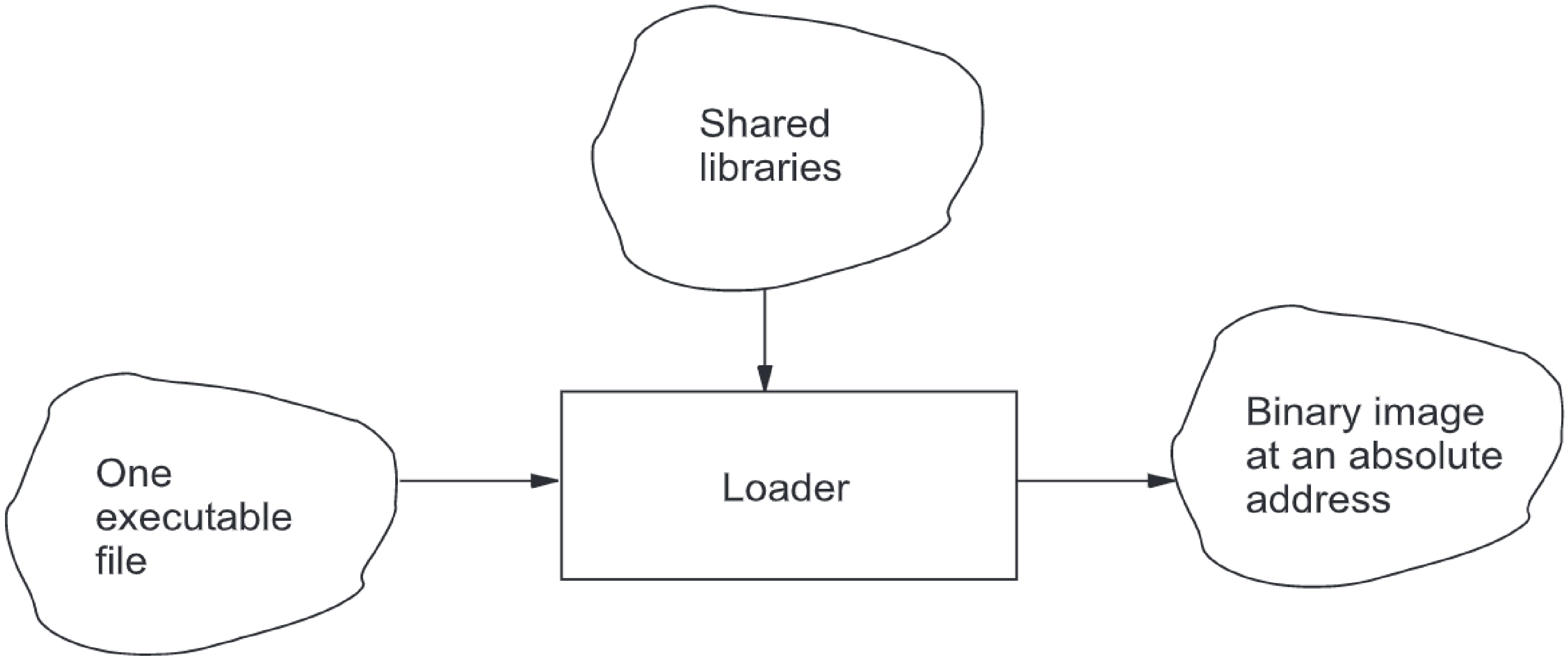
Loader

- Loading is the process of bringing an executable file produced by the linker from the secondary storage to the primary storage in such a way that it can be executed by the operating system.
- In most of the modern systems, loader is a part of the operating system itself, making it invisible to the common users. It may be noted that the executable module produced by the linker is assumed to start from a fixed memory location. However, the process of loading may necessitate it to start from a different start address depending upon the availability of memory locations.
- Thus, all address sensitive portions within the code need to be modified again to reflect the change. Thus, loading is essentially the same as the relocation performed by a static linker, cxcept that it applics to just a single file (the cxecutable module) instead of a number of object files. The objective of loading is to bring a binary image into memory and bind relocatable addresses to absolute ones.
- Input-output of the loader is shown in Fig.

- It may be noted that a number of executable files may be using the same set of libraries (for example, the input/output routines of the system are used by all executables).

- Hence, instead of loading the libraries several times, it is advisable to share the loaded library between the processes.

- The library routines, being read-only code, are not affected by multiple programs using them simultaneously.


Shared libraries
One executable file
Loader
Binary image at an absolute address

Macros

- A macro displays a generally used group of statements in the source programming language.

- The macro processor substitutes every macro instruction with the equivalent group of source language statements.

- This is known as expanding the macros.

- Macro instructions permit the programmer to write a shorthand edition of a program, and leave the mechanical details to be managed by the macro processor.

Text Editor

- Editor's the utility used most by the program developers. The user creates the program using editor.
- Thus, a very simple editor should do the bare minimum job of taking the characters from the user and store it in the same order in a file. However, this basic operation needs to be augmented in many ways.
- First of all, the files may become too large to allow a number of users to do editing simultaneously. Thus, the portions of the file need to be stored in the
- secondary storage and brought back to memory when the user wants to edit those portions.
- The first generation of editors was known as line editors as their unit of editing was a full line. That is, even to modify a single character, one had to re-enter the entire line.

- The EDLIN available with DOS operating system is an example of this category. The next generation of editors is screen editors that allow the use of cursor keys to go to any desired location in the file and modify it.

- The vi editor of Unix is an example of screen oriented text editor.

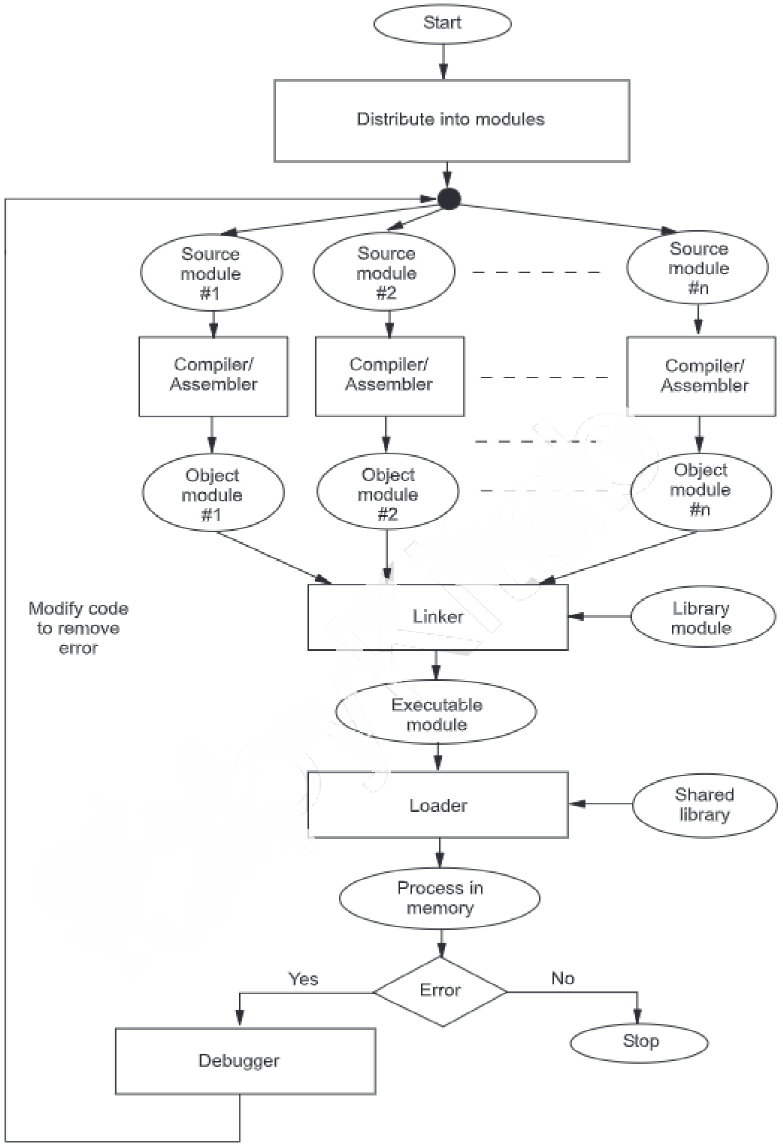
- Since most of the modern operating systems are windows based, editors have also gone through another advancement. The window based editors possess a mouse interface through which any segment of the program can be selected for necessary operation (for example, copy, delete, move, etc.).

- They also provide very convenient and powerful search-and-replace strategy to make editing easy. Language-sensitive editors have made the life of the programmer much easier with their ability to display language keywords and variables in different colours for ready identification, automated parentheses matching to reduce the burden of checking balanced parentheses, and many other language dependent features.

Debugger

- The tools that we have described so far help in writing grammatically correct programs, satisfying syntactic and semantic requirements of a language.
- However, these tools cannot catch any logical mistakes in the program.
- In fact, it is the responsibility of the programmer to ensure that the idea of the algorithm is correctly coded into the program.
- In the case of logical bugs, it is necessary to run the program step-by-step, and check the program variables at these intermediary stages

Program Development Flow


Start
Distribute into modules
Source module #1
Source module #2
Source module #n
Compiler/ Assembler
Compiler/ Assembler
Compiler/ Assembler
Object module #1
Object module #2
Object module #n
Modify code to remove error
Linker
Library module
Executable module
Loader
Shared library
Process in memory
Yes
Error
No
Debugger
Stop