What is Compiler?

Compiler is a computer 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).

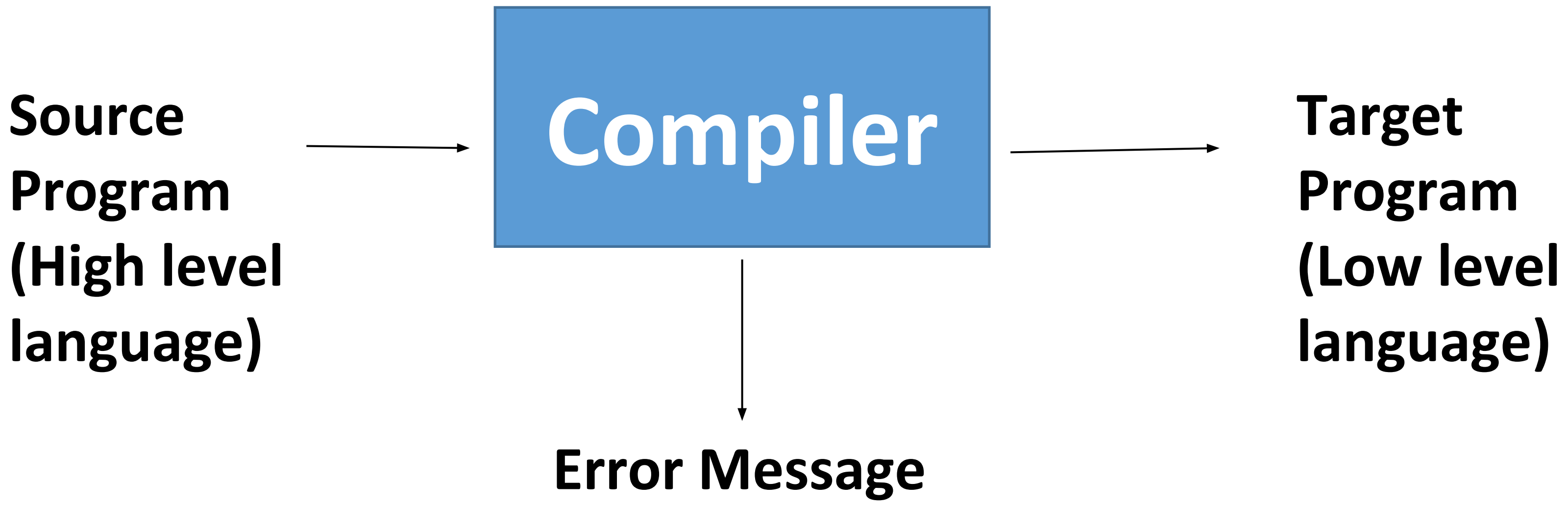

Source Program (High level language)
Compiler
Target Program (Low level language)
Error Message

Fig: A Compiler

- A **source program/code** is a program/code written in the source language, which is usually a high-level language.
- A **target program/code** is a program/code written in the target language, which often is a machine language or an intermediate code.

If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs.

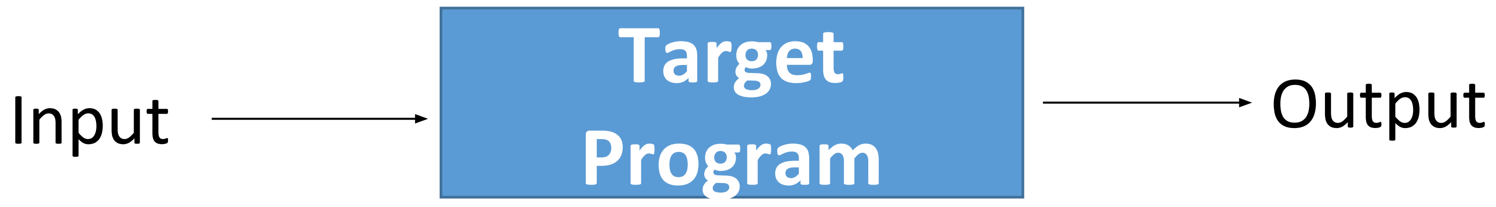


Fig: Running the Target Program

What is Interpreter?

- Another common kind of language processor.
- Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user.

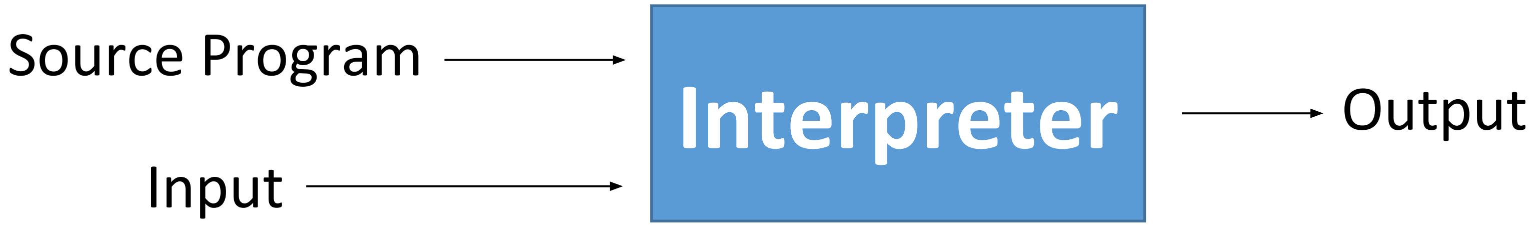


Fig: An Interpreter

Compiler vs Interpreter

- 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.

Example

Java language processors combine compilation and interpretation. A Java source program may first be compiled into an intermediate form called bytecodes. The bytecodes are then interpreted by a virtual machine. A benefit of this arrangement is that bytecodes compiled on one machine can be interpreted on another machine, perhaps across a network. In order to achieve faster processing of inputs to outputs, some Java compilers, called just-in-time compilers, translate the bytecodes into machine language immediately before they run the intermediate program to process the input.

Hybrid Compiler

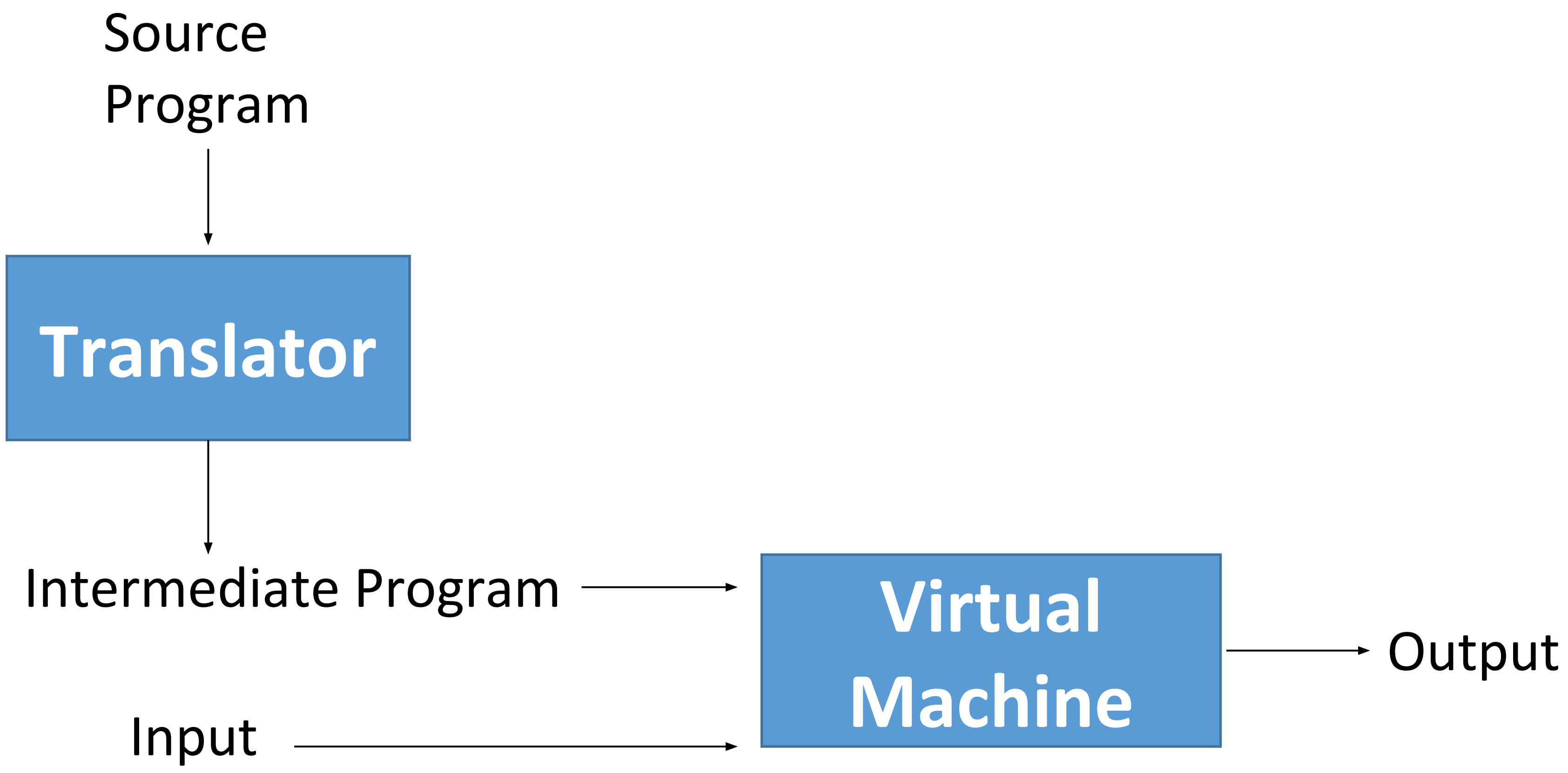


Fig: A Hybrid Compiler

A Language Processing System

- **High Level Language (Source program) –** If a program contains #define or #include directives such as #include or #define it is called HLL. They are closer to humans but far from machines. These (#) tags are called pre-processor directives. They direct the pre-processor about what to do.
- **Pre-Processor –** The pre-processor removes all the #include directives by including the files called file inclusion and all the #define directives using macro expansion. It performs file inclusion, augmentation, macro-processing etc.
- **Assembly Language –** Its neither in binary form nor high level. It is an intermediate state that is a combination of machine instructions and some other useful data needed for execution.
- **Assembler –** For every platform (Hardware + OS) there is a assembler. They are not universal since for each platform it has one. The output of assembler is called object file. Its translates assembly language to machine code.

A Language Processing System (Cont.)

- **Interpreter –** An interpreter converts high level language into low level machine language, just like a compiler. But they are different in the way they read the input. The Compiler in one go reads the inputs, does the processing and executes the source code whereas the interpreter does the same line by line. Interpreted programs are usually slower with respect to compiled ones.
- **Relocatable Machine Code –** It can be loaded at any point and can run. The address within the program will be in such a way that it will cooperate for the program movement.
- **Loader/Linker –** It converts the relocatable code into absolute code and tries to run the program resulting in a running program or an error message (or sometimes both can happen). Linker loads a variety of object files into a single file to make it executable. Then loader loads it in memory and executes it.

A Language Processing System (Cont.)

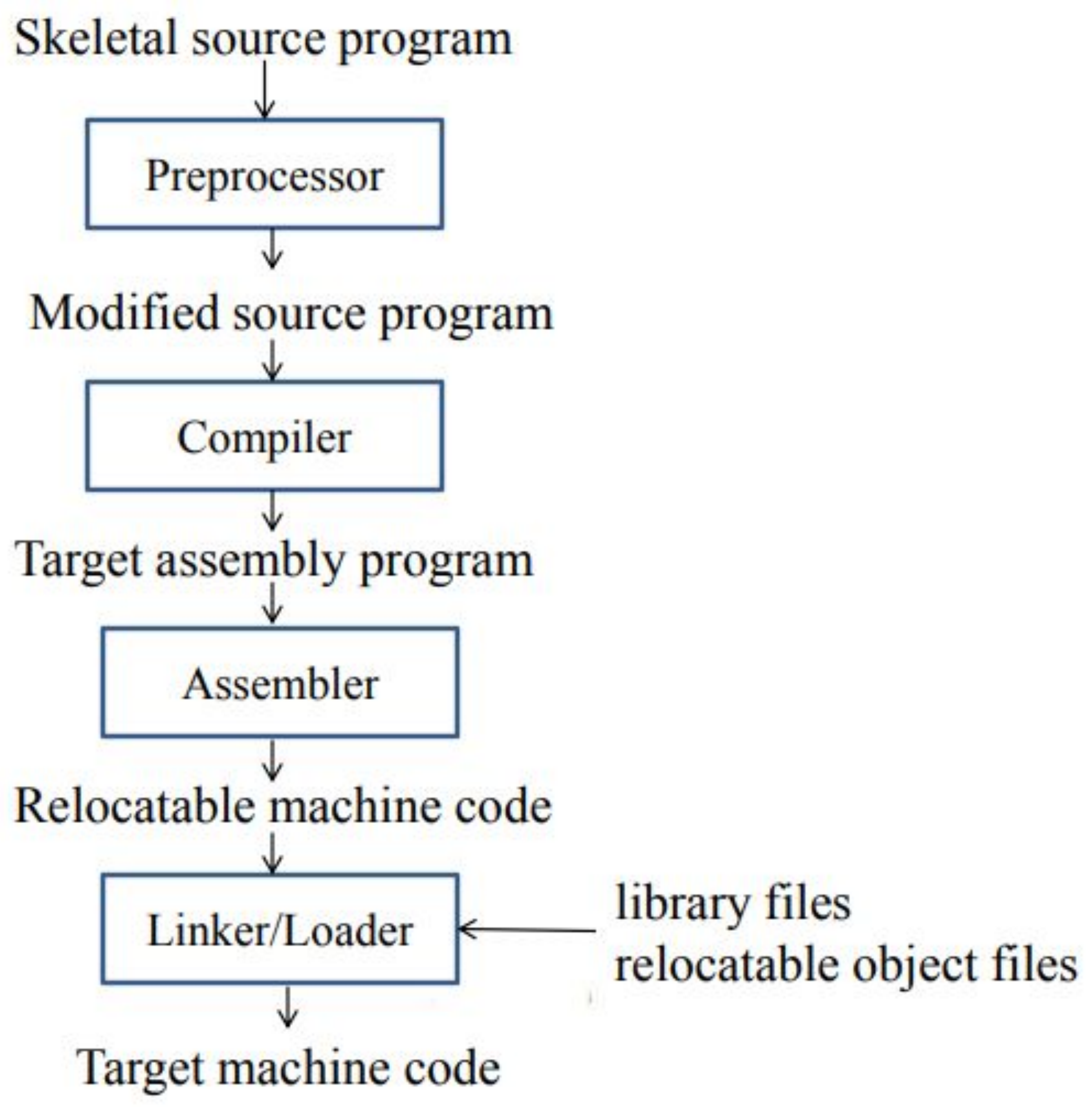


Fig: A Language Processing System

Compiler Generated Code

- **Pure Machine Code**
  - Machine instruction set without assuming the existence of any operating system or library.
  - Mostly being OS or **embedded applications**.
- **Augmented Machine Code**
  - Code with OS routines and runtime support routines.
  - More often used.
- **Virtual Machine Code**
  - Virtual instructions, can be run on any architecture with a virtual machine interpreter or a just-in-time compiler.
  - Ex. Java.

Structure of a Compiler

The TWO Fundamental Parts (The Analysis-Synthesis Model of Compilation**):**

- **Analysis:** Decompose Source program into an intermediate representation. The analysis part breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program. If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so the user can take corrective action. The analysis part also collects information about the source program and stores it in a data structure called a symbol table, which is passed along with the intermediate representation to the synthesis part.

Structure of a Compiler(Cont.)

- **Synthesis:** Target program generation from intermediate representation. The synthesis part constructs the desired target program from the intermediate representation and the information in the symbol table.

The analysis part is often called the front end of the compiler; the synthesis part is the back end.

The Phases of a Compiler:

If we examine the compilation process in more detail, we see that it operates as a sequence of phases, each of which transforms one representation of the source program to another. A typical decomposition of a compiler into phases is shown

The Phases of a Compiler

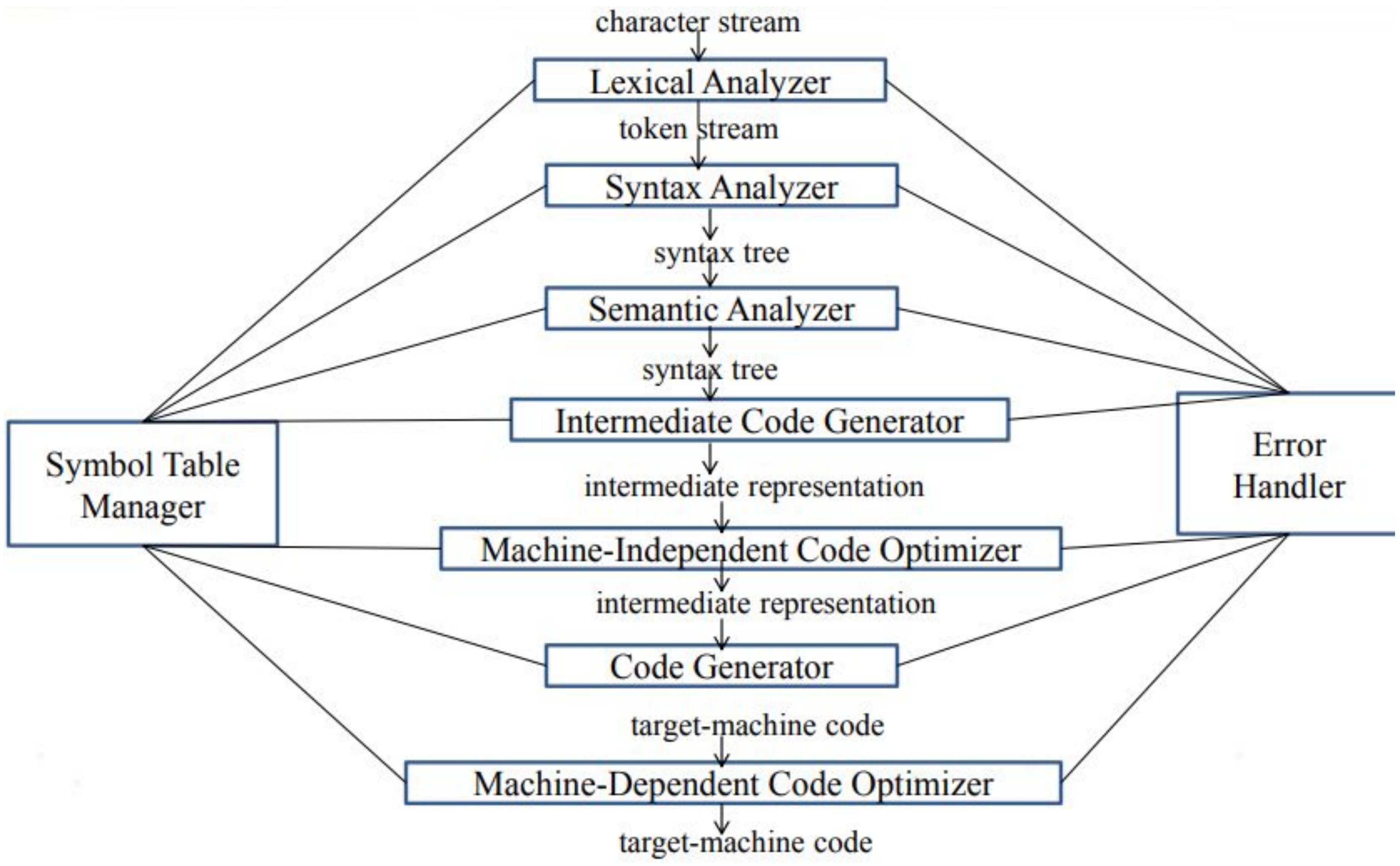


Fig: Phases of a Compiler

The Phases of a Compiler(cont.)

- **Single Pass Compiler:** If we combine or group all the phases of compiler design in a **single** module known as single pass compiler.

- **Two Pass compiler *or* Multi Pass compiler:** A Two pass/multi-pass Compiler is a type of compiler that processes the *source code* or abstract syntax tree of a program multiple times.

Analysis of a Source Program

Three basic Phases:

1) **Linear / Lexical Analysis:**
   - Read Left -to-right and group into Tokens.
   - token: sequence of chars having a **Collective Meaning**.
2) **Hierarchical Analysis:**
   - Grouping of Tokens Into **Meaningful Collection**.
3) **Semantic Analysis:**
   - Checking to ensure **Correctness** of Components

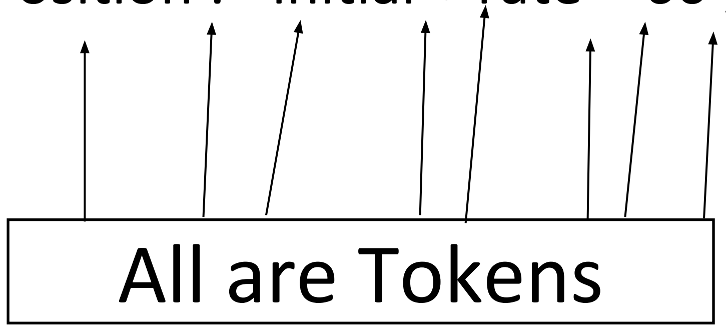
The Phases of a Compiler (Cont.)

Lexical Analysis:

- The first phase of a compiler is called lexical analysis or scanning.
- The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes.
- Identify tokens which are the basic building blocks.
- Blanks, Line breaks, etc. are scanned out.

For Example:

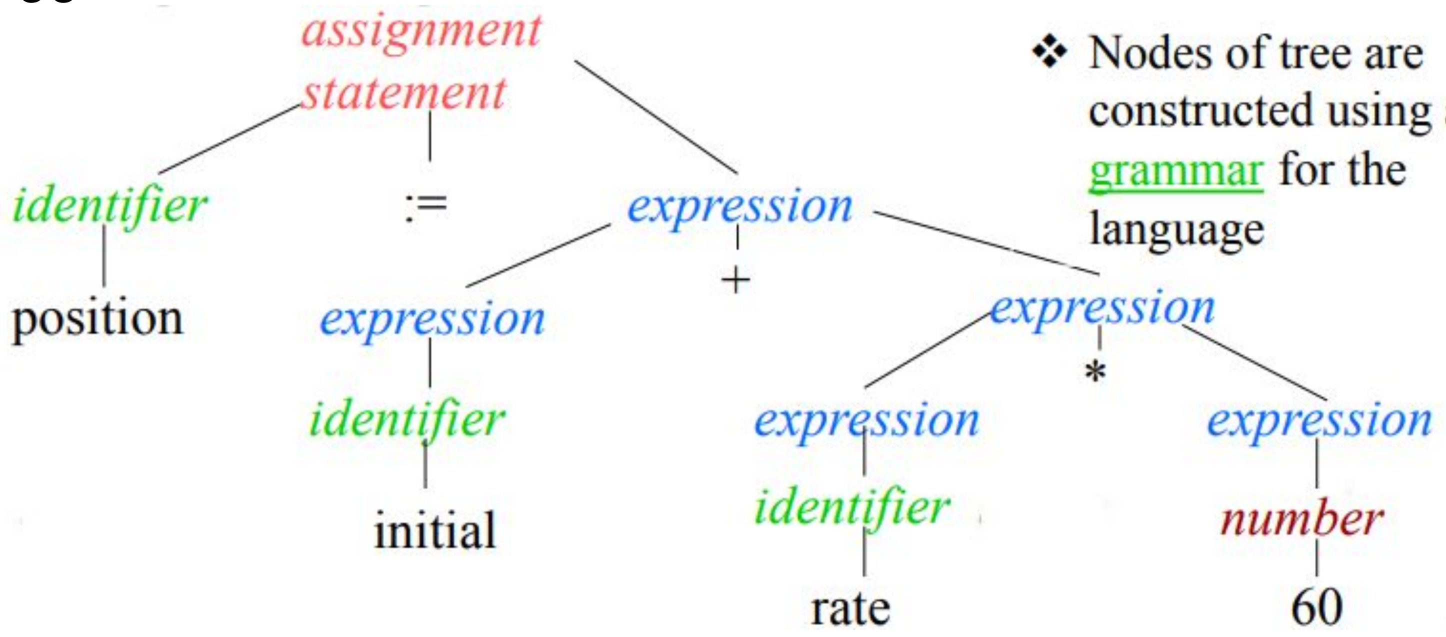
Position := initial + rate * 60 ;



The Phases of a Compiler (Cont.)

 Syntax Analysis:

- The second phase of the compiler is syntax analysis or parsing.
- Involves grouping the tokens of the source program into grammatical phrases that are used by the compiler to synthesize output.
- In syntax tree each interior node represents an operation and the children of the node represent the arguments of the operation.
- position = initial + rate * 60

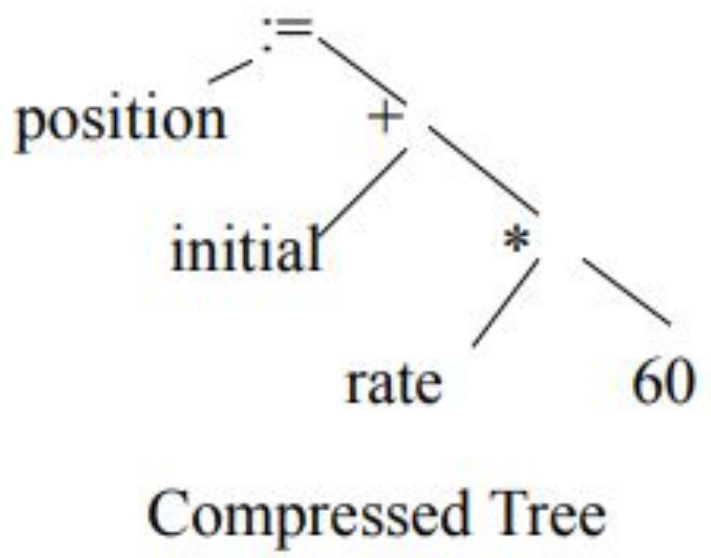


❖ Nodes of tree are constructed using a grammar for the language

The Phases of a Compiler (Cont.)

- **Semantic Analysis:**
  - Checks the source program for sematic errors and gathers type information.
  - Saves the information in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.
  - Uses the syntax tree with Semantic Actions.
  - An important part of semantic analysis is type checking, where the compiler checks that each operator has matching operands.
  - **For example, many programming language definitions require an array index to be an integer; the compiler must report an error if a floating-point number is used to index an array.**

The Phases of a Compiler (Cont.)



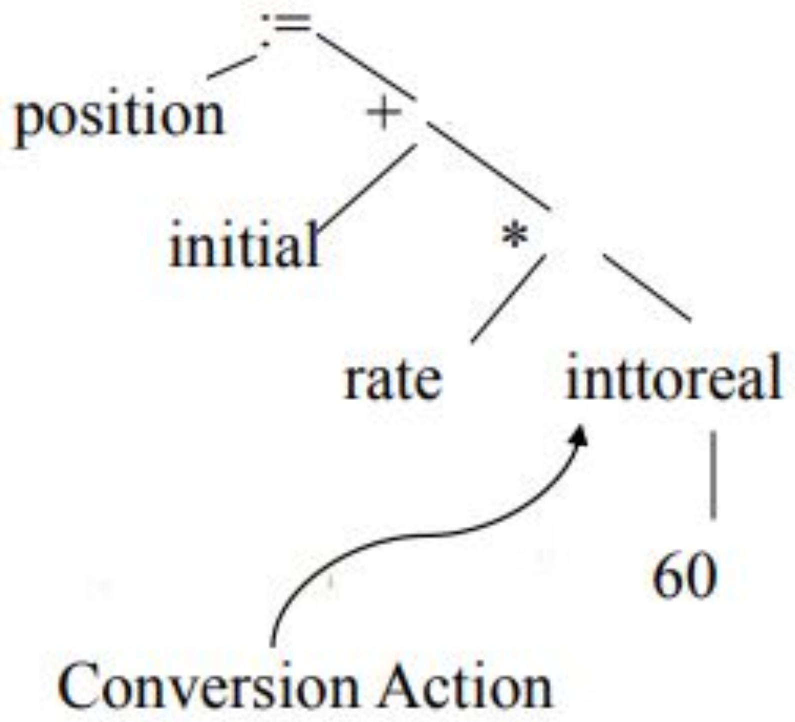


Fig: Semantic Analysis

The Phases of a Compiler (Cont.)

 Symbol Table Creation / Maintenance:

- The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name.
- Contains Info (storage, type, scope, args) on Each “Meaningful” Token, Typically Identifiers.
- Data Structure Created / Initialized During Lexical Analysis.
- Utilized / Updated During Later Analysis & Synthesis.
- The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly.

The Phases of a Compiler (Cont.)

- **Error Handling:**
    - Detection of different errors which correspond to all phases.
    - What Kinds of Errors Are Found During the Analysis Phase?
    - What Happens When an Error Is Found?

The Phases of a Compiler (Cont.)

 Intermediate Code Generation:

- After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or **machine-like** intermediate representation of source program.
- This intermediate representation should have two important properties:
  1. It should be easy to produce and
  2. It should be easy to translate into the target machine.

The Phases of a Compiler (Cont.)

Code Optimization:

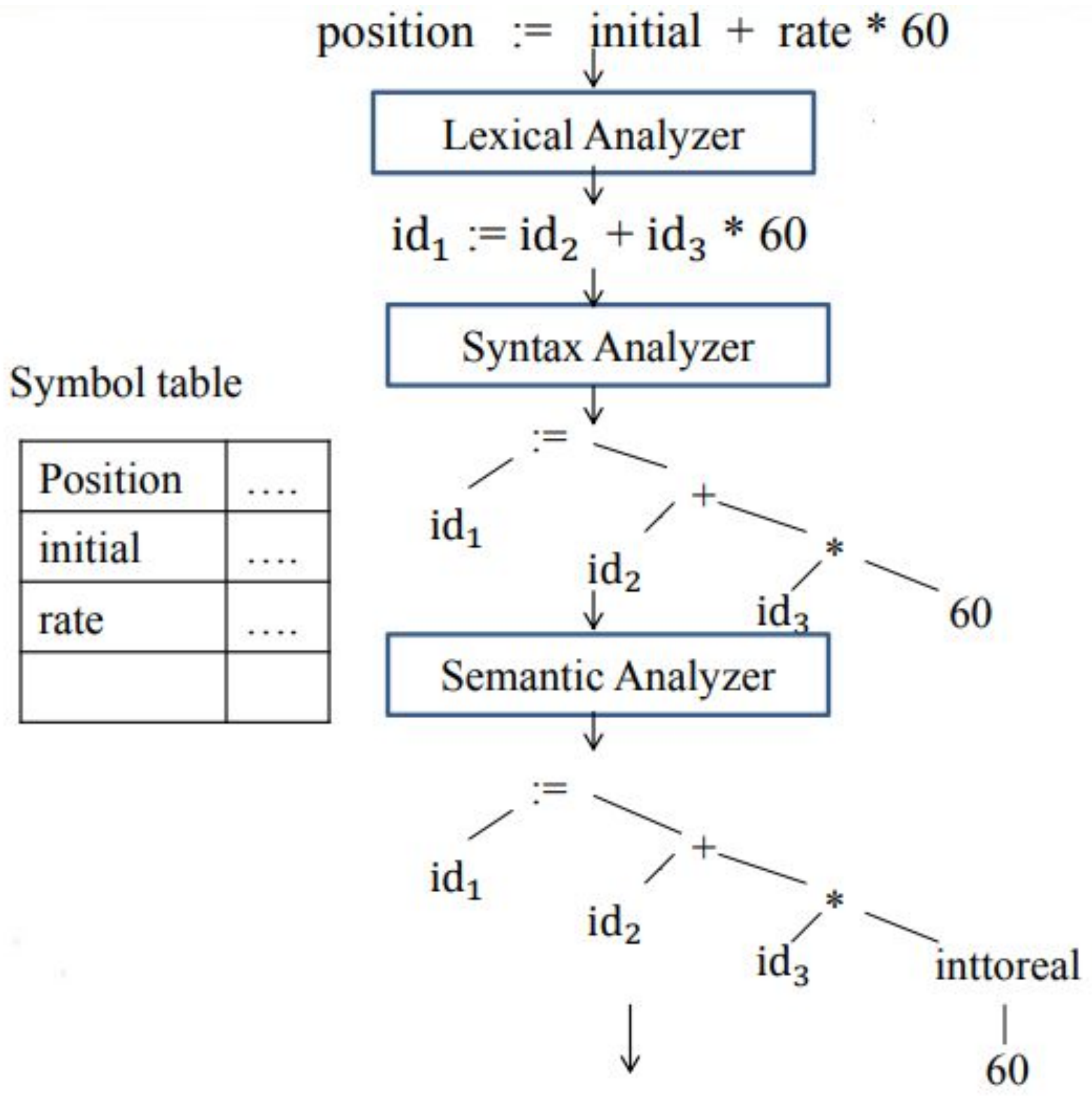
- Attempts to improve the intermediate code so that better target code will result.
- Find More Efficient Ways to Execute Code.
- Replace Code With More Optimal Statements.
- 2-approaches:
  1. High-level Language &
  2. “Peephole” Optimization

The Phases of a Compiler (Cont.)

 Final Code Generation:

- Takes as input an intermediate representation of the source program and maps it into the target language.
- Generate Relocatable Machine Dependent Code.

The Phases of a Compiler (Cont.)



The Phases of a Compiler (Cont.)

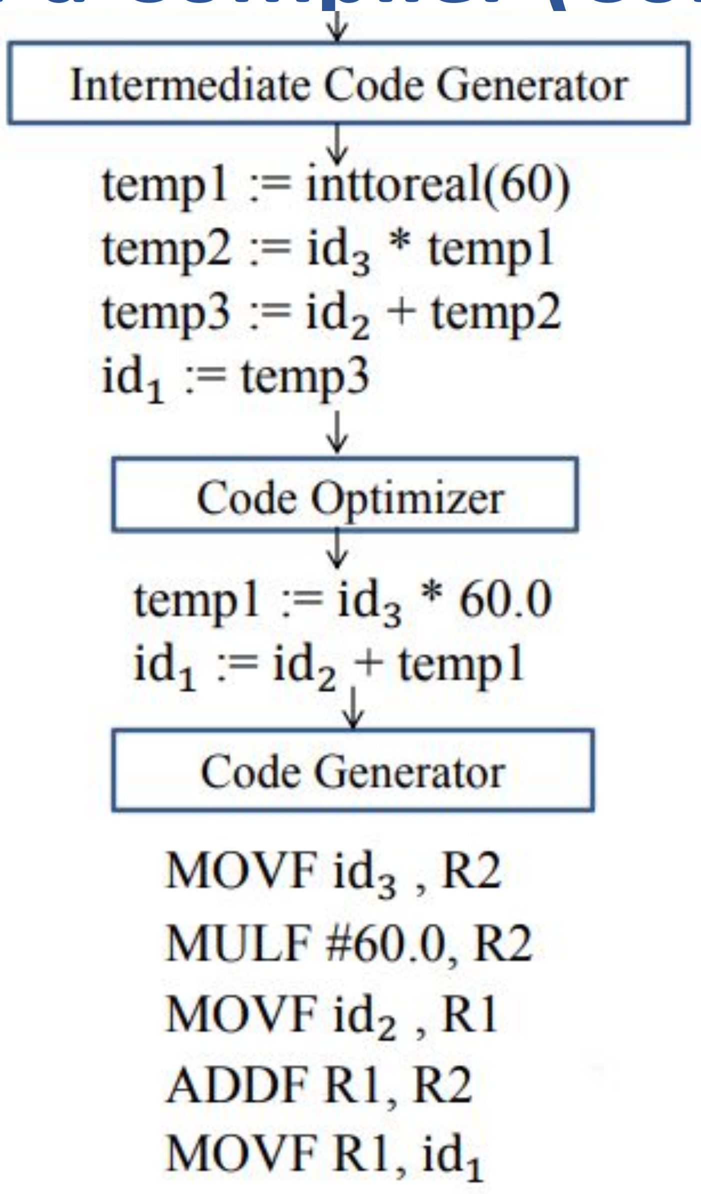


Fig: Translation of an assignment statement

Symbol Table in Compiler

Symbol Table is an important data structure created and maintained by the compiler in order to keep track of semantics of variable, i.e. it stores information about **scope and binding information** about names, information about instances of various entities such as variable and function names, classes, objects, etc.

- It is built in lexical and syntax analysis phases.
- The information is collected by the analysis phases of compiler and is used by synthesis phases of compiler to generate code.
- It is used by compiler to achieve compile time efficiency.

Symbol Table entries – Each entry in symbol table is associated with attributes that support compiler in different phases.

Symbol Table in Compiler (Cont.)

▯ Items stored in Symbol table:

- Variable names and constants
- Procedure and function names
- Literal constants and strings
- Compiler generated temporaries
- Labels in source languages

▯ Information used by compiler from Symbol table:

- Data type and name
- Declaring procedures
- Offset in storage
- If structure or record then, pointer to structure table.
- For parameters, whether parameter passing by value or by reference
- Number and type of arguments passed to function
- Base Address

Symbol Table in Compiler (Cont.)

- **Implementation of Symbol Table**
  - List
  - Linked list
  - Hash table
  - Binary Search Tree

Applications of Compiler Technology

1. Implementation of High-Level Programming Languages
2. Optimizations for Computer Architectures.
3. Design of New Computer Architectures.
4. Program Translations.
5. Software Productivity Tools