Chapter-One

Introduction to compiler Design

Preliminaries Required

- Basic knowledge of **FSA** and **CFG**.
- Basic knowledge of programming languages.
  - ✓ Knowledge of a high programming language for the programming assignments.

Textbook:

❖ *Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, "**Compilers: Principles, Techniques, and Tools**" Second Edition, Addison-Wesley, 2007.*

Objective of the course

- To learn techniques of a modern compiler
- Be able to build a compiler for a simplified (programming) language
- Know how to use compiler construction tools, such as *generators of scanners and parsers etc.*
- Be familiar with ***compiler analysis*** and ***optimization techniques.***
- … learn how to work on a larger software project!

Topics will be covered in Chapter-1

- *History of compiler*
- *Why study compiler?*
- *What is a compiler?*
- *Cousins of the compiler*
- *Structure of compiler*
- *Phases of compiler*
- *The Grouping of Phases into Passes*
- *Compiler Construction Tools*
- *The Evolution of Programming Languages*
- *Application of compiler technology*
- *Impacts on compiler*
- *Programming language basics*

History of compiler

- No high-level languages were available, so all programming was done in *machine language* and assembly language.
- As expensive as these early computers were, most of the money companies spent was for software development, due to the complexities of assembly.
- In 1953, **John Backus** came up with the idea of **"speed coding"**, and developed the first ***interpreter***.
- Unfortunately, this was ***10-20 times*** slower than programs written in assembly.
- *He was sure he could do better.*

John Backus

- In 1954, Backus and his team released a research paper titled *"Preliminary Report, Specifications for the IBM Mathematical FORmula TRANslating System, FORTRAN."*

History of compiler (cont'd)

- The initial release of **FORTRAN** was in 1956, totaling 25,000 lines of assembly code.
- Compiled programs run almost as fast as handwritten assembly!
- Projects that had taken two weeks to write now took only 2 hours.
- By 1958 more than half of all software was written in **FORTRAN**.
- **Programming languages** are notations for describing computations to people and to machines.
- The world as we know it depends on programming languages, because all the software running on all the computers was written in some ***programming language***.
- But, before a program can be run, it first must be translated into a form in which it can be executed by a computer.
- The software systems that do this translation are called ***compilers***.

Programming Language (PL)

- **What's a Programming language?**
  - Formal language designed to communicate instructions to a computer.
- There are two major types of PL: ***Low-level and high level language***

i. Low-level language (LLL)

- Very close to hardware components
- Language that are easier for HW to understand
- Machine oriented and require extensive knowledge of computer HW and its configuration.
- Two categories of LLL:

  a. Machine Language & b. Assembly Language

i. Low-level language (LLL) cont'd

a. Machine Language

- is only language that is directly understood by the computer
- It does not need to be translated
- Understood by CPU.
- Example: binary notation (0 & 1)- 00011101

b. Assembly Language

- The first step to improve the ***program instructions*** make machine language is more readable by humans.
- Set of symbol and letters
- 2nd Generation language. Example: MOVE, ADD, SUB, END

ii. High-Level programming (HLL)

- Is programming language that uses English and mathematical symbol in its instructions like **+, -, %** and any information
- HLL- that's what you usually use? E.g. *Java, C++, python , etc*
- HLL is most close to the logic of human language
- **E.g. consider ATM machine, some one want withdrawal $100.**
- Instructions in high level in computer language would looks something like this:

```
X=100
    if    balance <X:
              print 'Insufficient balance'
     else
             Print 'please take your money'
```

Why study compilers?

- We study compiler construction for the following reasons:
  - Writing a compiler gives a student experience with large-scale applications development
  - To understand performance issues for programming
  - Increases understanding of language semantics
  - Seeing the machine code generated for language constructs helps understand performance issues for languages.

What is a compiler?

- A **compiler** is a program that translates source program written in one language into the target program of another language.

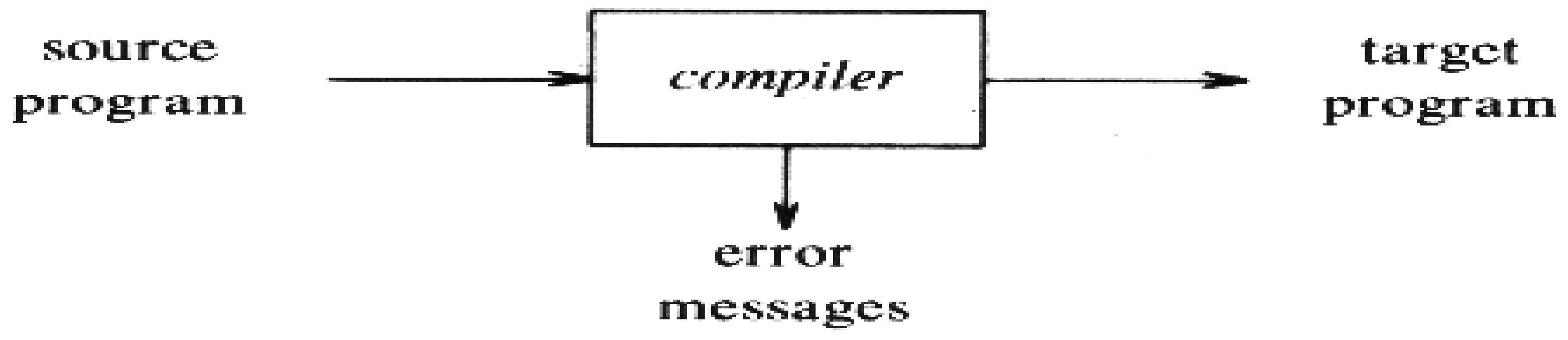


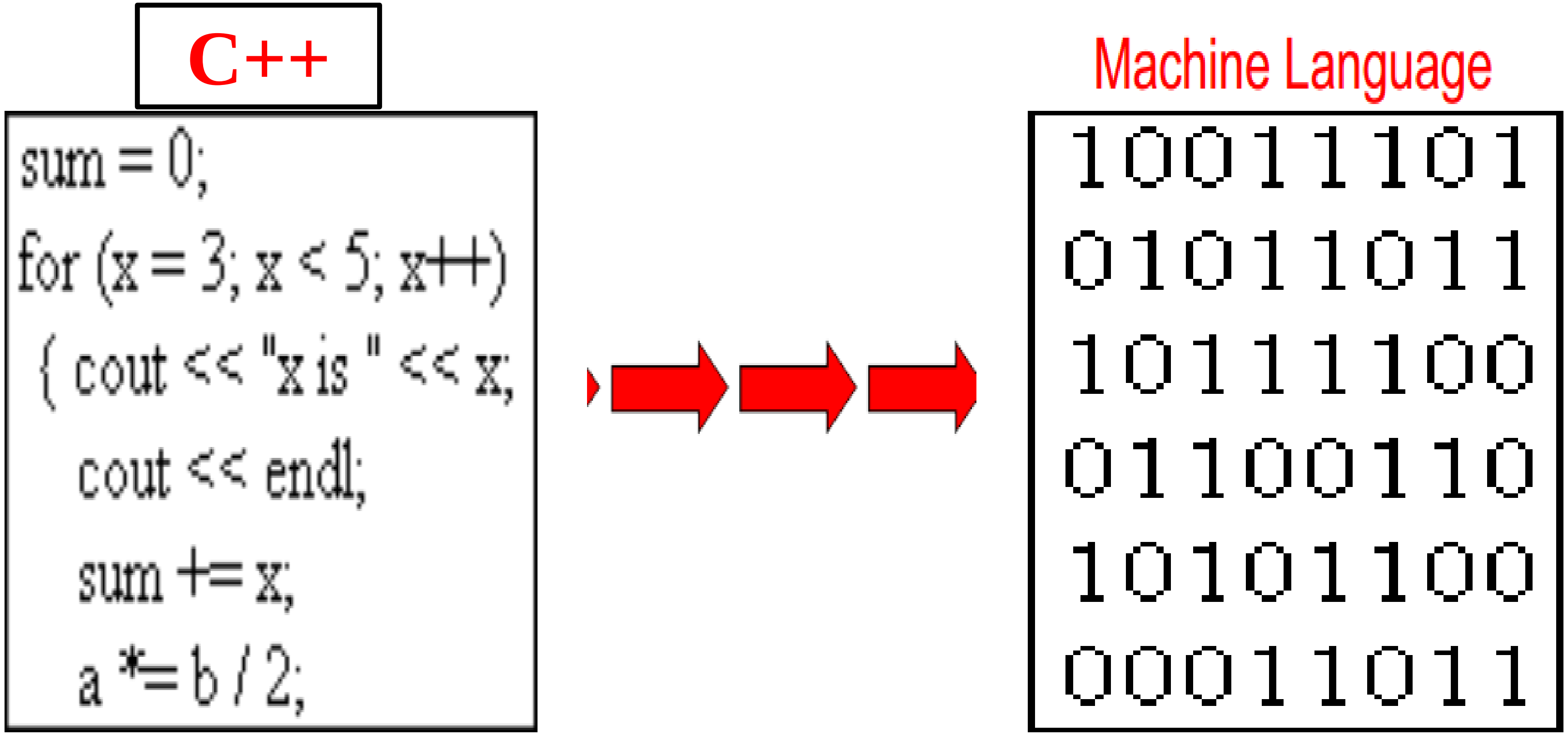
Fig. 1.1. A compiler.

- An important role of the compiler is to report any errors in the source program that it detects during the translation process.

What is a compiler? (cont'd)

- Usually the **source language (program)** is a high level language like Java, C, C++, Python, etc.
- whereas the **target language (program)** is machine code or "code" that a computer's processor understands.
- The **source language** is optimized for humans.
  - It is more user-friendly, to some extent platform-independent.
  - *They are easier to read, write, and maintain, and hence it is easy to avoid errors.*
- A program written in any language must be translated into a form that is understood by the computer.
  - ✓ This form is typically known as Machine Language (ML) or Machine Code, or Object Code.
- **Consists of streams of 0's and 1's**

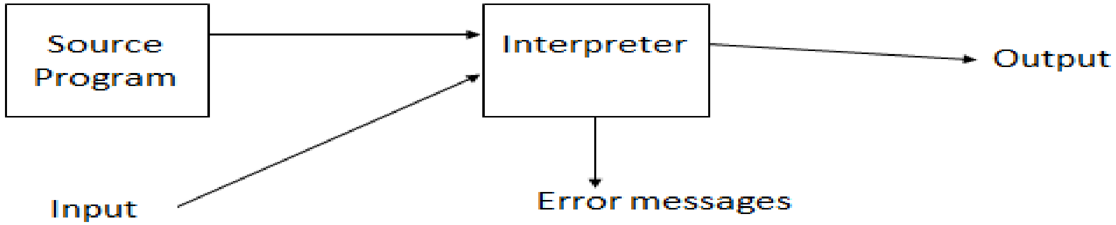
Example: From Source Code into Machine language



- So, the program that carries out the translation activity is called the ***compiler***.

Interpreters

- An **interpreter** is 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.



- ***Interpreter:*** *processes an internal form of the source program and data at the same time (at run time).*
- *no object program is generated.*

Compiler vs Interpreter

- **Compiler** takes entire program as input and creates all executable instructions together as compiled code.
- Compilation initially takes time, because it has to create whole **HLL** into machine language at once, and save it on **Hard disk**.
- The file got saved is called ***compiled code*** or ***object code***.
- The compiled code can be used again and again by the **OS** for execution.
- **Once the compiled code is done,**
  - the OS do not need the compiler once again
  - Its easy for the OS to schedule the executable instructions to the CPU
  - If error is occurs then the compiler does not create any single machine instruction

Compiler vs Interpreter

- But , An **Interpreter**:
  - takes single instructions as input and create one or more corresponding machine instructions.
  - does not create a file on the disk.
- Overall delay by an Interpreter is more than a compiler
  - Because there is no any saved object file on the disk.
- The OS always need to wait for the Interpreter for execution of the code.
- If error occurs then interpreter creates machine instructions for all the line before the error.

Combining Both Interpreter and Compiler

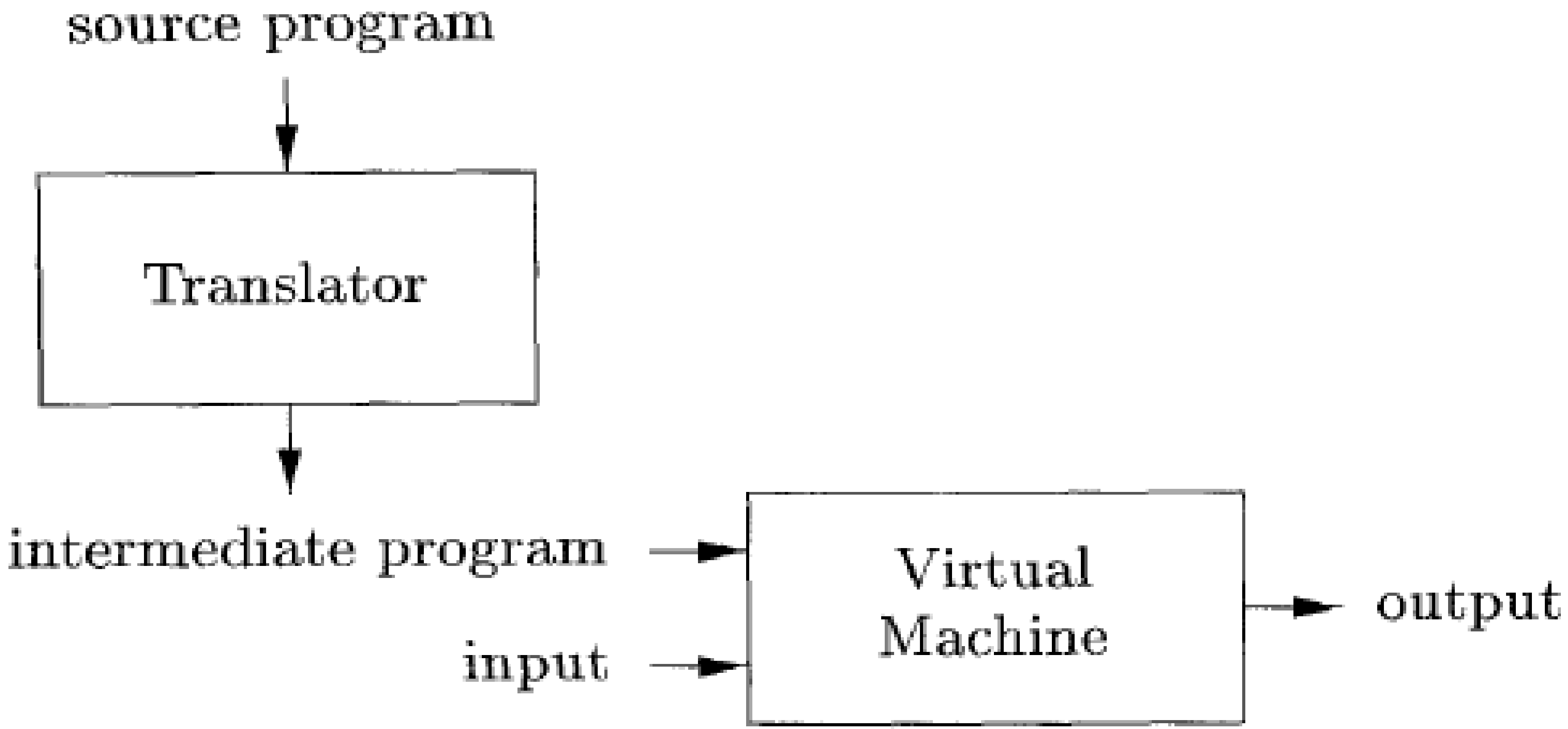


Fig: A hybrid compiler

Example: How its work

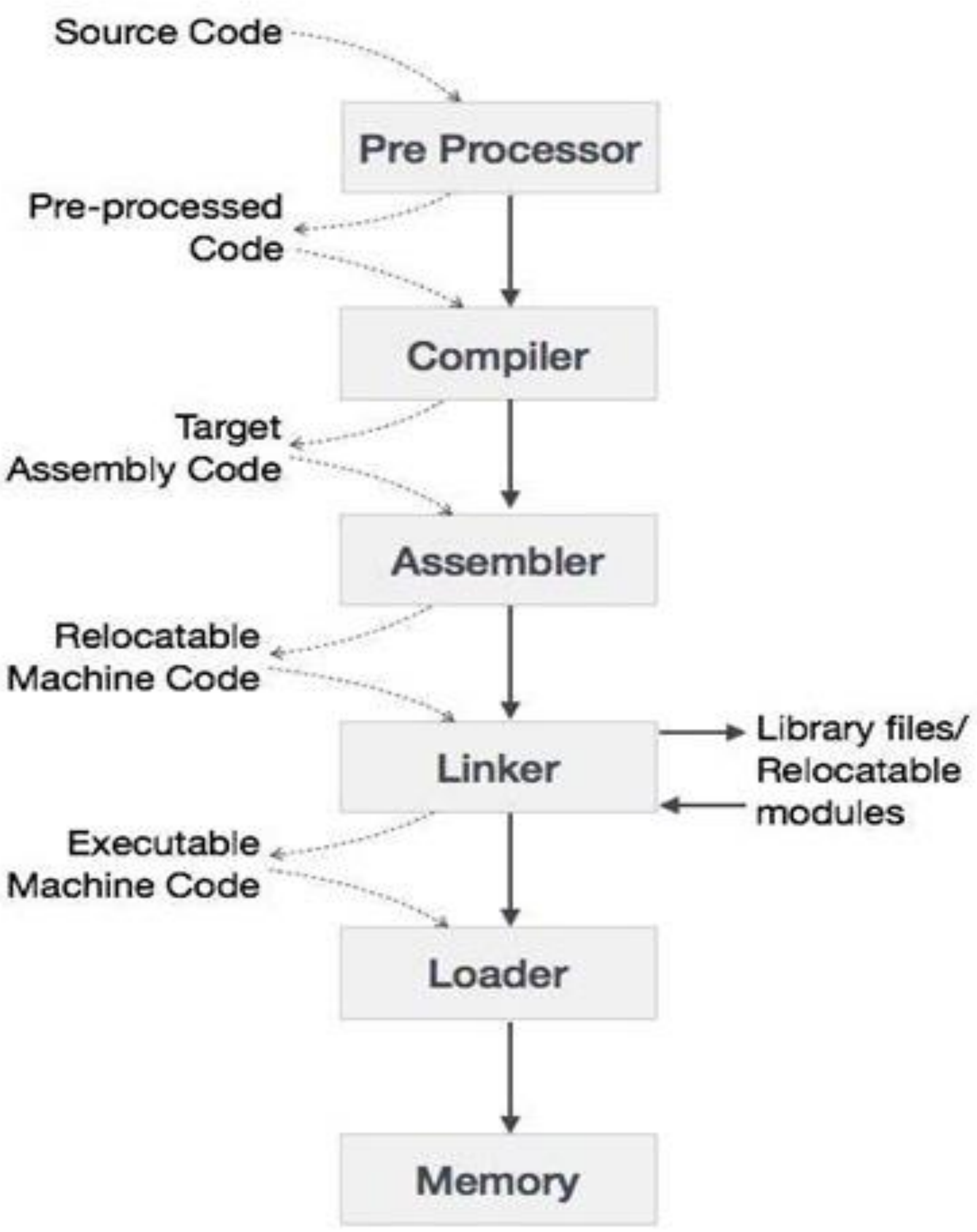
- Java language processors combine compilation and interpretation, as shown in figure.
- A Java source program may first be compiled into an intermediate form called *bytecodes*.
- The bytecodes are then interpreted by a virtual machine.
- In order to achieve faster processing of inputs to outputs, some ***Java compilers***, called ***just-in-time*** compilers,
- translate the **bytecodes** into machine language immediately before they run the intermediate program to process the input.

Example #1

- Programming language use compiler
  - *C*, *C++*
- Programming language use Interpreter
  - Python, perl
- Programming language use both
  - Java

Cousins of a compiler

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



Cousins of a compiler (cont'd)

a. **Pre-processor:**

- 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***.
- It includes all the **header files** and also evaluates if any macro is included.
- It is the optional because if any language which does not support **#include** and ***macro*** preprocessor is not required.

Cousins of a compiler (cont'd)

- They may perform the following functions:

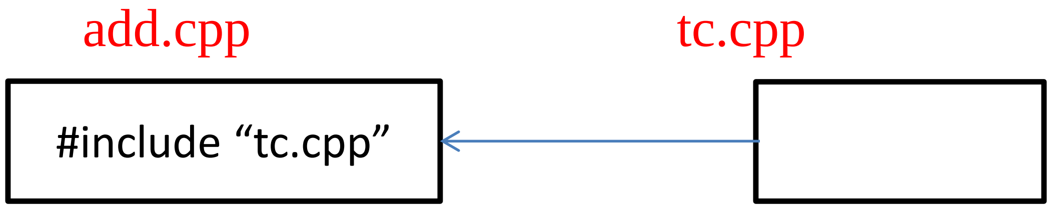
i. Macro processing

- 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.
- E.g C program

```
#define Cube(x);
Void main(){
Cout<<cube(3);
```

ii. 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.



Cousins of a compiler (cont'd)

b. **Compiler**:

- It takes pure high level language as a input and convert into assembly code.

c. Assembler

- It takes the assembly code that has been generated by the compiler and convert it into ***re-locatable machine code***.

Example: Address binding

Mov a, b

add c, 28

d. Linker :

- 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.
- Also used to link all parts of program together for execution.
- Library routine or link to library

Header file

printf();

scanf();

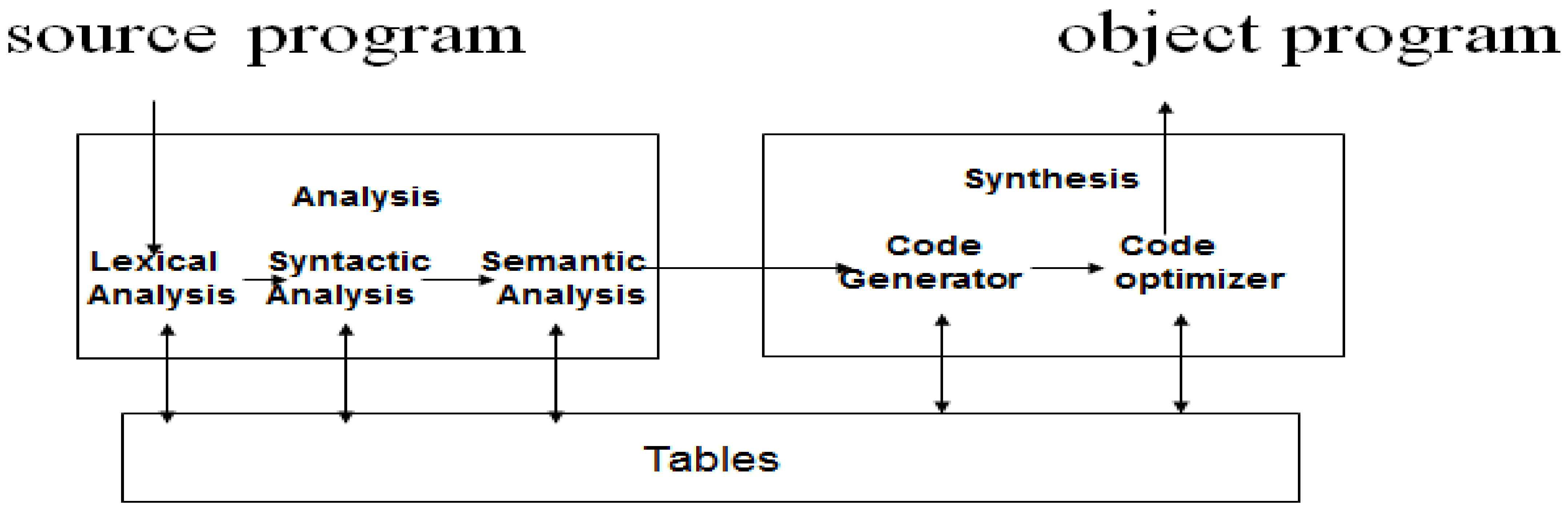
Cousins of a compiler (cont'd)

e. Loader:

- A **loader** is the part of an operating system that is responsible for loading programs in memory.
- It loads the executable file into permanent storage.
  - *That means: takes the entire program to main memory for execution. Example: **exe.file***
- **Note:** Here ,we only concentrate on Compiler
  - *What exactly the compiler will do?*
  - *What is internal process of the compiler?*

The Structure of a Compiler

- Any compiler must perform two major tasks: such as ***Analysis and Synthesis*** *as shown in fig*



The Structure of a Compiler

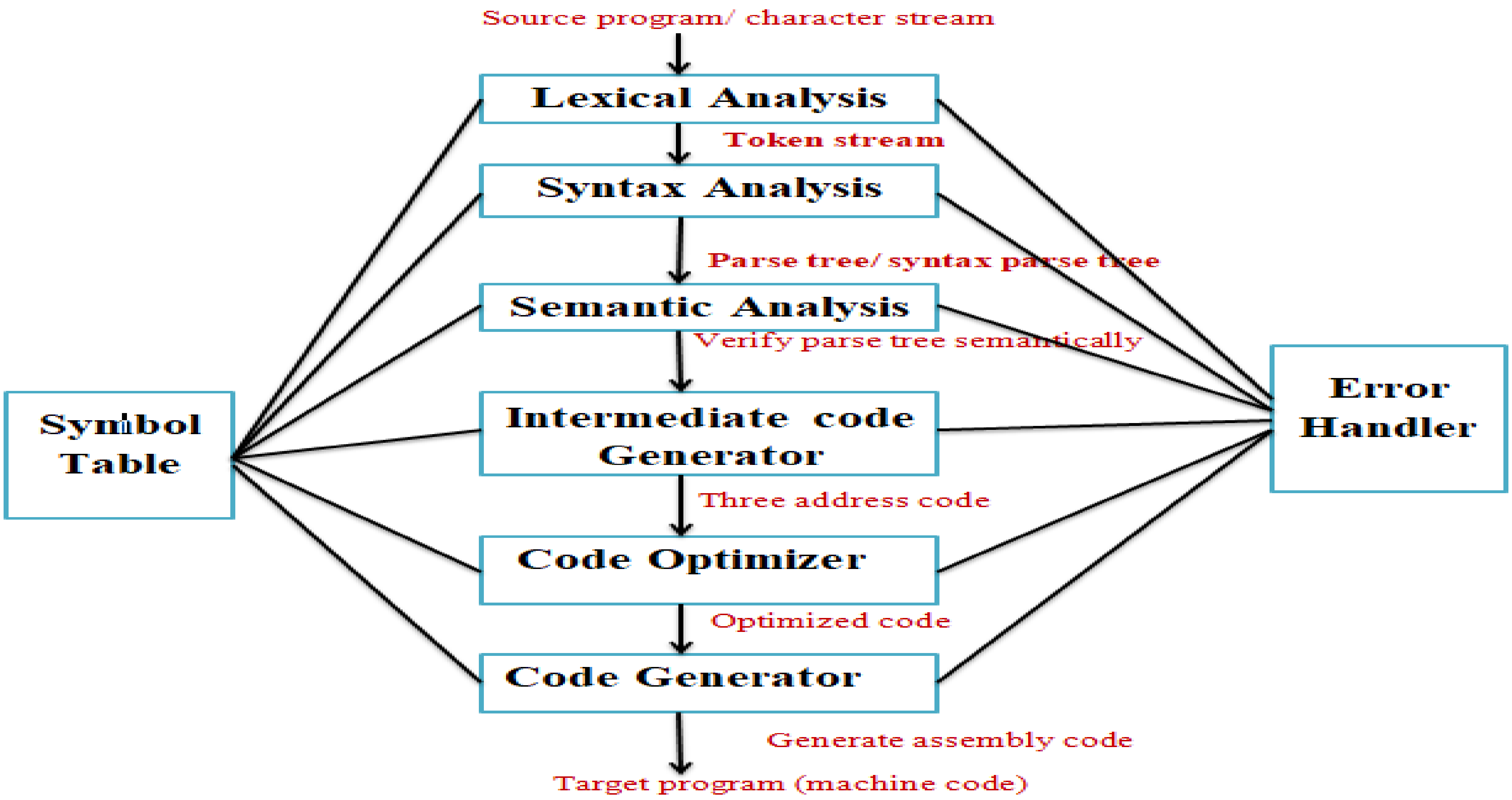
i. Analysis:

- It takes the input source program and breaks it into parts.
- It then creates an intermediate representation of source program.
- the *front end* of the compiler and Machine independent
- *Lexical Analyzer, Syntax Analyzer and Semantic Analyzer* are the parts of this phase.

ii. Synthesis:

- It takes the intermediate representation as the input and creates the desired target program.
- the ***back end*** of the compiler and Machine dependent
- *Code Generator, and Code Optimizer are the parts of this phase.*

Phases of a compiler



Phases of a compiler(cont'd.)

- Each phase transforms the source program from one representation into another representation.
- Each phase communicate with **error handlers** and **symbol table.**
- **Example:** How to compiler translate source program into target program.

  The given Source program: ***Position = initial + rate * 60***
- *To answer the above example, let us discuss each phase of the compiler as follows:*

Phase 1. Lexical Analysis (scanner)

- First phases of the compiler
- It 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 a **token** of the form that it passes on to the subsequent phase, **syntax analysis.**

  (token-name, attribute-value)

- **Token-name:** an abstract symbol is used during syntax analysis, an
- **attribute-value:** points to an entry in the symbol table for this token.

Example: what will do at lexical phases?

Position = initial + rate * 60

Cont'd

- **"position"** is a lexeme that is mapped into the ***token (id, 1),*** where 1 points to the symbol-table entry for **position**.
- **=** is a lexeme that is mapped into the **token (=).**
- **"initial"** is a lexeme that is mapped into the ***token (id, 2),*** where 2 points to the symbol-table entry for **initial**.
- **+** is a lexeme that is mapped into the **token (+).**
- **"rate"** is a lexeme mapped into the ***token (id, 3),*** where **3** points to the symbol-table entry for rate.
- ***** is a lexeme that is mapped into the **token (*)** .
- **60** is a lexeme that is mapped into the **token (60)**

Cont’d

- **Blanks separating the lexemes would be discarded by the lexical analyzer.**
- after lexical analysis as the sequence of tokens

  <i d , l > < = > **<id,** 2> <+ > **<id,** 3> <*> <60)>

- In this representation, the token names =, +, and * are abstract symbols for the assignment, addition, and multiplication operators, respectively.

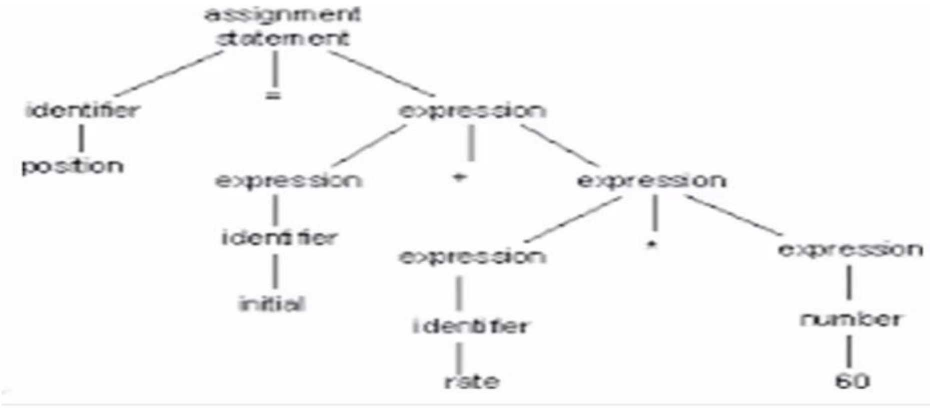
Phase 2. Syntax Analysis (parser tree)

- The second phase of the compiler
- 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 typical representation is
  - a **syntax tree** in which each interior node represents an operation and
  - the **children** of the node represent the arguments of the operation.

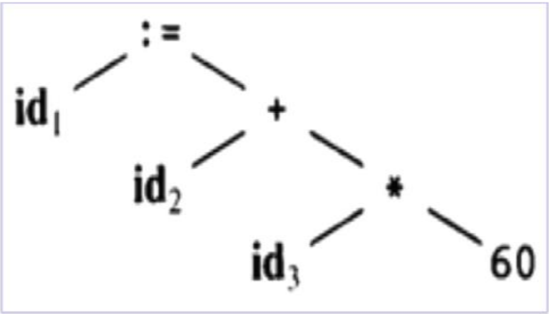
Cont’d

- **Rule of Syntax analysis:**
    - *Any identifier is an expression*
    - *Any number is an expression*
    - *If E1 and E2 are expression, then E1*E2 and E1+E2 is an expression*
    - *If id is an identifier and E is an expression then id=E*

- This tree shows the order in which the operations in the assignment: **Position = initial + rate * 60**

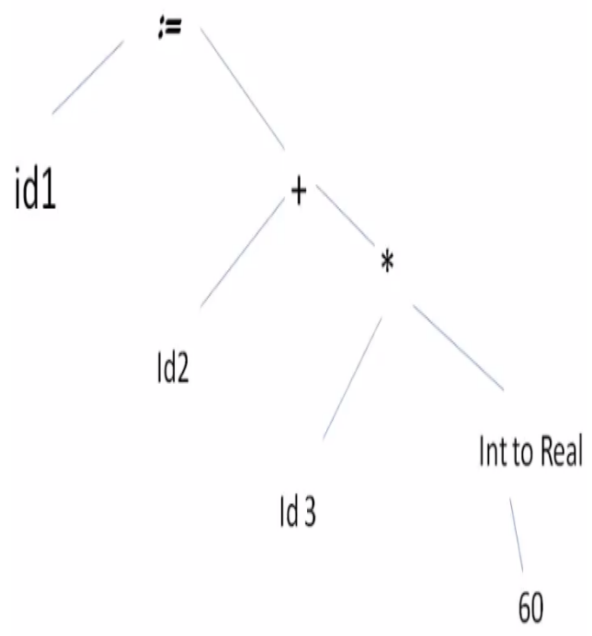


- **Sequence of token produced form lexical Analysis: id1=id2+id3*60**



Phase 3:- Semantic Analysis

- The semantic analyzer uses the **syntax tree**
- Performing 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.



Phase 4:- Intermediate Code Generation

- After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine
- The considered intermediate form called three-address code, which consists of a sequence of assembly
- ICG-should have two important properties:
  - *it should be easy to produce and*
  - *it should be easy to translate into the target machine.*
- Each instructions has at most three **operands**
- Each operand can act like a register.

t1 = int to float (60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3

Phase 5:- Code Optimization

- The code-optimization phase attempts to improve the intermediate code so that generate better target code will result.
- Usually better means: *faster, shorter code, or target code that consumes less power*.
- The optimizer can deduce that the conversion of 60 from integer to floating point can be done once and for all at compile time,
- so the int to float operation can be eliminated by replacing the integer 60 by the floating-point number 60.0. Moreover, t3 is used only once .

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

Phase 6:- Code Generation

- 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.
- A crucial aspect of code generation is the judicious assignment of registers to hold variables.

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

7. Symbol-Table Management

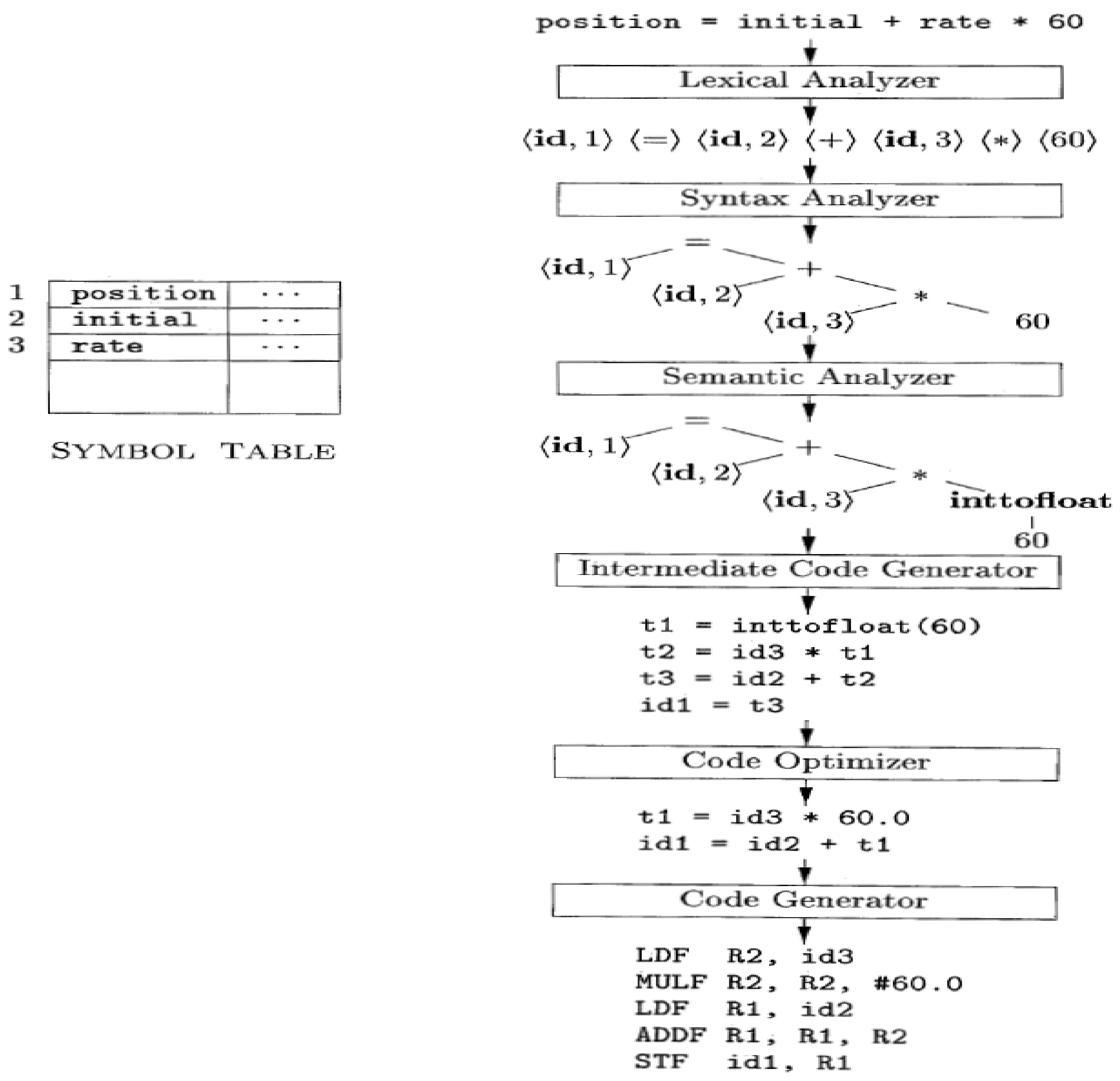
- The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name.
- The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly.

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

SYMBOL TABLE

Example:

- How all phases of compiler works together



Grouping of Compiler Phases

- **Front end**
  - Consist of those phases that depend on the source language but largely independent of the target machine.
  - *For example, the front-end phases of lexical analysis, syntax analysis, semantic analysis*
- **Back end**
  - Consist of those phases that are usually target machine dependent such as code optimization and code generation.

Compiler-Construction Tools (CCTs)

- **Defining CCTS:**
  - programs or environments that assist in the creation of an entire compiler or its parts.
- **Uses for CCTs to generate:**
  - *lexical analyzers, syntax analyzers, semantic analyzers, intermediate code, optimized target code.*
- Some commonly used compiler-construction tools include:
  - *Scanner generators, Parser generators Syntax-directed translation engines, Code-generators, Data-flow analysis engines, Compiler-construction toolkits* .

Reading Assignment

1. How do compiler works?
2. How do computers read a code (source program)?
3. What are the advantages of (a) a compiler over an interpreter (b) an interpreter over a compiler?
4. The Evolution of Programming Languages
5. Application of compiler technology
6. **Impacts on Compilers**
7. Programming language basics

Reading Assignment: Cont'd

9. What advantages are there to a language-processing system in which the compiler produces assembly language rather than machine language?

10. A compiler that translates a high-level language into another high-level language is called a *source-to-source* translator. What advantages are there to using C as a target language for a compiler?

Reading Assignment: Cont'd

12. Indicate which of the following terms: apply to which of the following languages:

a) Imperative b) declarative c) von Neumann d) object-oriented e) functional

f) third-generation g) fourth-generation h) scripting

1) C 2) C + + 3) Cobol

4) Fortran 5) Java 6) Lisp

7) ML 8) Perl

9) Python 10) VB.

End of Chapter- One