Compiler Design

Introduction

Shyamalendu Kandar
Assistant Professor, IT
IIEST, Shibpur

Meaning

Dictionary Meaning:

----to gather a number of things together into one.

----To put together; to assemble; to make by gathering things from various sources.

It is a verb.

Origin : Latin word '*compilo*'

Definition

A **compiler** is a program takes a program written in a source language and translates it into an equivalent program in a target language.

In general…..

A compiler translates (or compiles) a program written in a high-level programming language that is suitable for human programmers into the low-level machine language that is required by computers.

Program & Programming Language

Program:

is a specification of what data computer is required to process, by using what operation, an by going in what sequence.

Programming Language:

notation used to specify the data operation/ instructions and sequences which are understandable to the computer.

Features of a Good Language:

Easy to Understand
Expressive power
Interoperability: to operate in conjunction with each other.
Good Turnaround Time
Portability
Error Reporting
Error Recovery
Efficient Memory usage
Good Run time environment
To model real world problems.

History of Compiler

- Developed by by Grace Hopper (1951 and 1952) for **A-0 system** (*Arithmetic Language version 0*), written for the UNIVAC I, (**UNIV**ersal **A**utomatic **C**omputer **I**).
- The Term 'Compiler' was first used by him.

More about A-0 Compiler:

- *Have similarity with operation of loader or linker.*
- *A program was specified as a sequence of subroutines and arguments. The subroutines were identified by a numeric code and the arguments to the subroutines were written directly after each subroutine code.*
- *The A-0 system converted the specification into machine code that could be fed into the computer a second time to execute the said program.*

- Credit of designing first complete compiler goes to John W. Backus at IBM for designing FORTRAN in 1957.
- ALGOL 58(1958), Extended Fortran (1960), COBOL (Dec 1960).
- COBOL was the first Compiler written in High Level language.

Where Compiler lies?

A program for a computer must be built by combining very simple commands.

Machine language: A language for computer.

But difficult for end user.

High level language: A language for End User.

Compiler **Bridges the gap** between these two languages.

Then why High-level?

- The notation used is closer to the way humans think about problems.
- End to be shorter than equivalent programs written in machine language.

Whether Machine language is obsolete?

- Programs that are written in a high-level language and automatically translated to machine language may run somewhat slower than programs that are hand-coded in machine language.
- some time-critical programs (real time system) are still written partly in machine language.

A Comparison among HLL & LLL.

Advantages of a high-level language over machine or assembly language

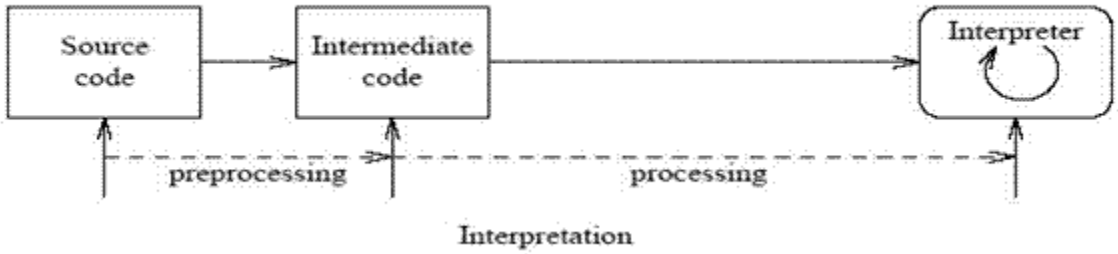
(1) Machine language (and even assembly language) is difficult to work with and difficult to maintain.
(2) With a high-level language you have a much greater degree of machine independence and portability from one kind of computer to another (as long as the other machine has a compiler for that language).
(3) You don't have to retrain application programmers every time a new machine (with a new instruction set) is
introduced.
(4) High-level languages may support data abstraction (through data structures) and program abstraction (procedures and functions).

Any disadvantages?

(1) The programmer doesn't have complete control of the machine's resources (registers, interrupts, I/O buffers).
(2) The compiler may generate inefficient machine language programs.
(3) Additional software – the compiler – is needed in order to use a high-level language. As compiler development and hardware have improved over the years, these disadvantages have become less problematic. Consequently, most programming today is done with high-level languages.

The Term – ‘Interpreter’

- ***Computer Program which serves a purpose very similar to that of*** *a* compiler.
- The input to an interpreter is a program written in a high-level language. Each time when an interpreter gets a high level language code to be executed, it converts the code into an intermediate code before converting it into the machine code.
- Each part of the code is interpreted and then execute separately in a sequence and an error is found in a part of the code it will stop the interpretation of the code without translating the next set of the codes.
- High level instruction or language is converted into intermediate from by an interpreter.



Compiler Vs Interpreter

	Compiler	Interpreter
Program I/P	Entire program.	Single instruction as input.
Program O/P	Machine Language	Intermediate form
Execution Time	less amount of time to analyze the source code but the overall execution time is slower.	large amount of time to analyze the source code but the overall execution time is comparatively faster.
Execution of conditional control statements.	Faster	slower
Memory Required:	More	Less
Error Displays	Checking entire program.	When a error occurs it stops and display the particular.
Example	C Compiler	LISP, BASIC

Does Compiler do all?

Question: Whether compiler generate the machine code?

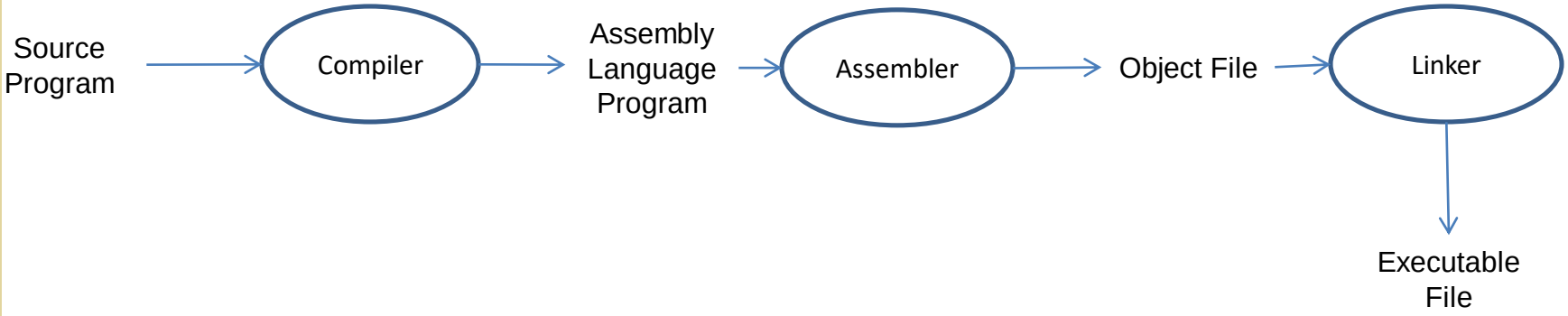
Several other program are required to create an executable target program.

Assembler: Converts assembly language programs into *object files*

Object files contain a combination of machine instructions, data, and information needed to place instructions properly in memory

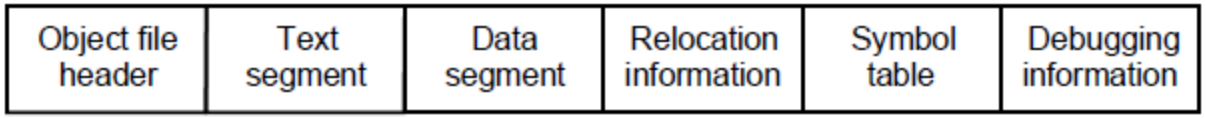
Linker : Library routines are linked with machine language module. Generate executable files.

Loader: Part of the OS that brings an executable file residing on disk into memory and starts it running



Object File

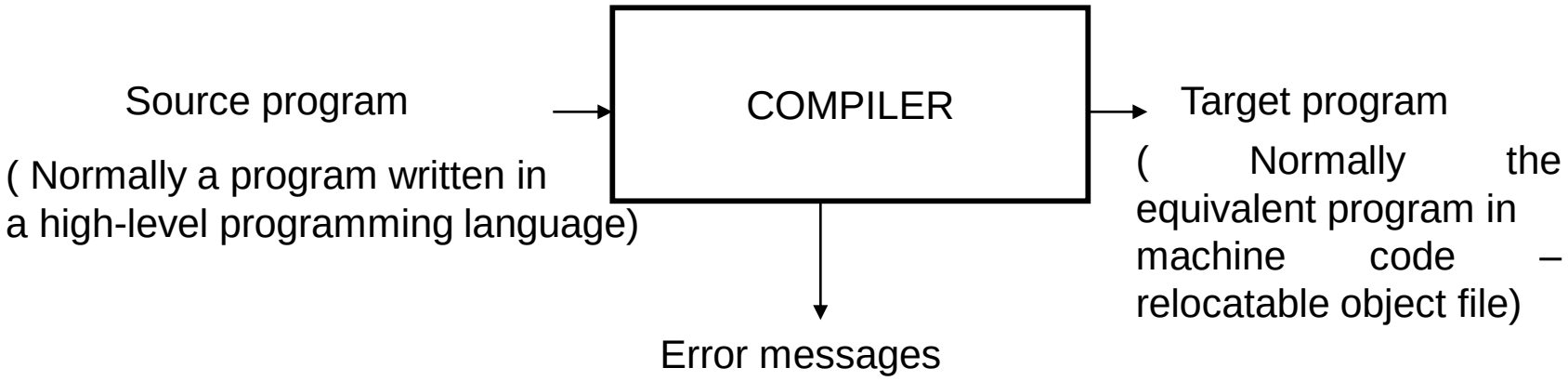
Object file format



- Object file header describes the size and position of the other pieces of the file
- Text segment contains the machine instructions
- Data segment contains binary representation of data in assembly file
- Relocation info identifies instructions and data that depend on absolute addresses
- Symbol table associates addresses with external labels and lists unresolved references
- Debugging info

BlackBox Model of Compiler

A **compiler** is a program takes a program written in a source language and translates it into an equivalent program in a target language.



Types of Compiler

Depending on Passes:

Single Pass, Multi pass

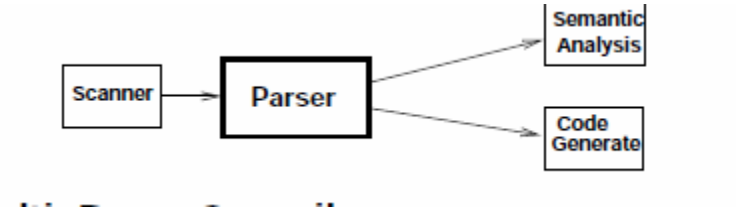
Single pass:

- that passes through the parts of each compilation unit only once, immediately translating each part into its final machine code.
- does not "look back" at code it previously processed. Another term sometimes used is **narrow compiler**

  Advantages: smaller and faster

  Disadvantages: unable to generate as efficient programs, due to the limited scope of available information.

  Code optimization is not properly done. Larger target code.



Example: **forward** declaration in Pascal.

Types of Compiler

Multi Pass:

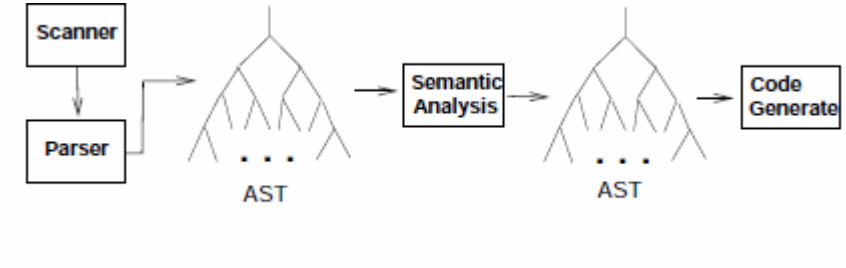
- that processes the source code or abstract syntax tree of a program several times.
- Each pass takes the result of the previous pass as the input, and creates an intermediate output. In this way, the (intermediate) code is improved pass by pass, until the final pass emits the final code.
- sometimes called **wide compilers**, referring to the greater scope of the passes

Advantages:

allows better code generation (e.g. smaller code size, faster code)

Disadvantages:

higher compiler time and memory consumption.



Types of Compiler

Some Other Types:

Source to Source Compiler:

- is a type of compiler that takes the source code of a programming language as its input and outputs the source code into another programming language.
- A source-to-source compiler translates between programming languages that operate at approximately the same level of abstraction.

Cross Compiler: A **cross compiler** is a compiler capable of creating executable code for a platform other than the one on which the compiler is running.

Just In Time(JIT): known as **dynamic translation**

done during execution of a program

rather than prior to execution

Challenges in Compiler Design

Language semantics: Presence of constructs with complex semantics [Example: case, loop, break, next].

Hardware Platform: RISC or CISC?:

individual instructions are very simple and regular in terms of size and execution time for RISC.

Instructions are of variable size and require variable execution time.

Operating System: The target code generated by a compiler finally goes to be executed in an environment created by an OS.

Error Handling:

Aid in debugging:

Speed of Compilation:

Few other application of techniques used in Compiler

The techniques used in compiler design can be applicable to many problems in computer science.

- Techniques used in a lexical analyzer can be used in text editors, information retrieval system, and pattern recognition programs.
- Techniques used in a parser can be used in a query processing system such as SQL.
- Many software having a complex front-end may need techniques used in compiler design.
- Most of the techniques used in compiler design can be used in Natural Language Processing (NLP) systems.

Major Parts of Compilers

- two major parts of a compiler: **Analysis** and **Synthesis**

- In analysis phase, the source program is broken into constituent pieces and an intermediate representation is created from the given source program.
- Analysis part consists of three phases

i. Linear Analysis
ii. Hierarchical Analysis
iii. Semantic Analysis

Major Parts of Compilers

i) Linear Analysis:

- stream of characters making up the source program is read from left to right .
- grouped into tokens, that are sequence of characters having a collective meaning.
- Linear Analysis is called lexical analysis or scanning.

Example

Total = initial+rate_of_interest * 5

After linear analysis this would be grouped into following tokens

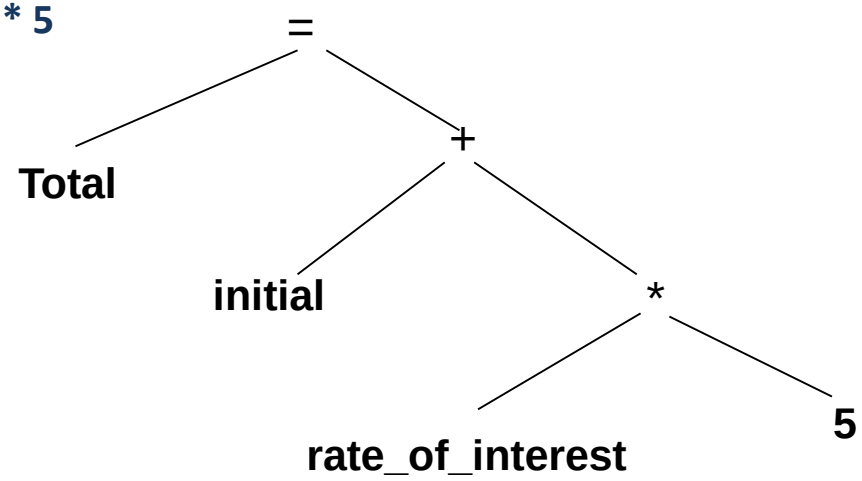
1. The identifier **'Total'**
2. The assignment symbol **'='**
3. The identifier **'initial'**
4. The **'+'** sign
5. The identifier **'rate_of_interest'**
6. The multiplication sign **'*'**
7. The number **'5'.**

Major Parts of Compilers

ii) Hierarchical Analysis:

- often called syntax analyzer or ***parser.***
- In this phase characters or tokens are grouped hierarchically into nested collection of collective meaning.
- check for proper syntax, issue appropriate error messages, and determine the underlying structure of the source program.
- In involves grouping the tokens of the source program into grammatical phases that are used by the compiler to synthesize output. Usually grammatical phases of the source program are represented by parse tree.

Total = initial+rate_of_interest * 5



Major Parts of Compilers

Semantic Analysis:

- certain checks are performed to ensure that the components of a program fit together meaningfully.
- checks the source program for semantic errors and gathers type information for subsequent code-generation.
- task of ensuring that the declarations and statements of a program are semantically correct,

  i.e , that their meaning is clear and consistent with the way in which control structures and data types are supposed to be used.

<u>Semantic analysis typically involves:</u>

Type checking

- Data types are used in a manner that is consistent with their definition (i. e., only with compatible data types, only with operations that are defined for them, etc.)
- Operator has operands that are permitted by the source language specification.

Label Checking

Labels references in a program must exist.

Flow control checks

control structures must be used in their proper fashion (no GOTOs into a FORTRAN DO statement, no breaks outside a loop or switch statement, etc)

Major Parts of Compilers

In synthesis phase, the equivalent target program is created from this intermediate representation.

- Intermediate Code Generator, Code Generator, and Code Optimizer are the parts of this phase.

Phases of A Compiler

- Each phase transforms the source program from one representation into another representation.
- They communicate with error handlers.
- They communicate with the symbol table.

Phases of A Compiler

Lexical analysis:

- reading and analyzing the program text.
- Source program is divided into tokens.
- each token corresponds to a symbol in the programming language, i.e. a variable name, keyword or number.
- Additional job: removing white spaces like blank, tab, newline character etc.

Syntax analysis:

- takes the list of tokens produced by lexical analysis.
- arranges these in a tree-structure , that reflects the structure of the program.
- This phase often called parsing.

Semantic Analysis:

- check for semantic correctness is done if the program is syntactically correct.
- Very common check that is done in this phase is check for the type of variables & expression occurring in the program.
- While deriving the types of expressions, normally two checks are needed.

a) Applicability of operators to operands, i.e. integer division is only allowed when operands are integers.

b) Determining the definition of the variables to be used [Same name may conflict.]

Phases of A Compiler

Intermediate code generation:

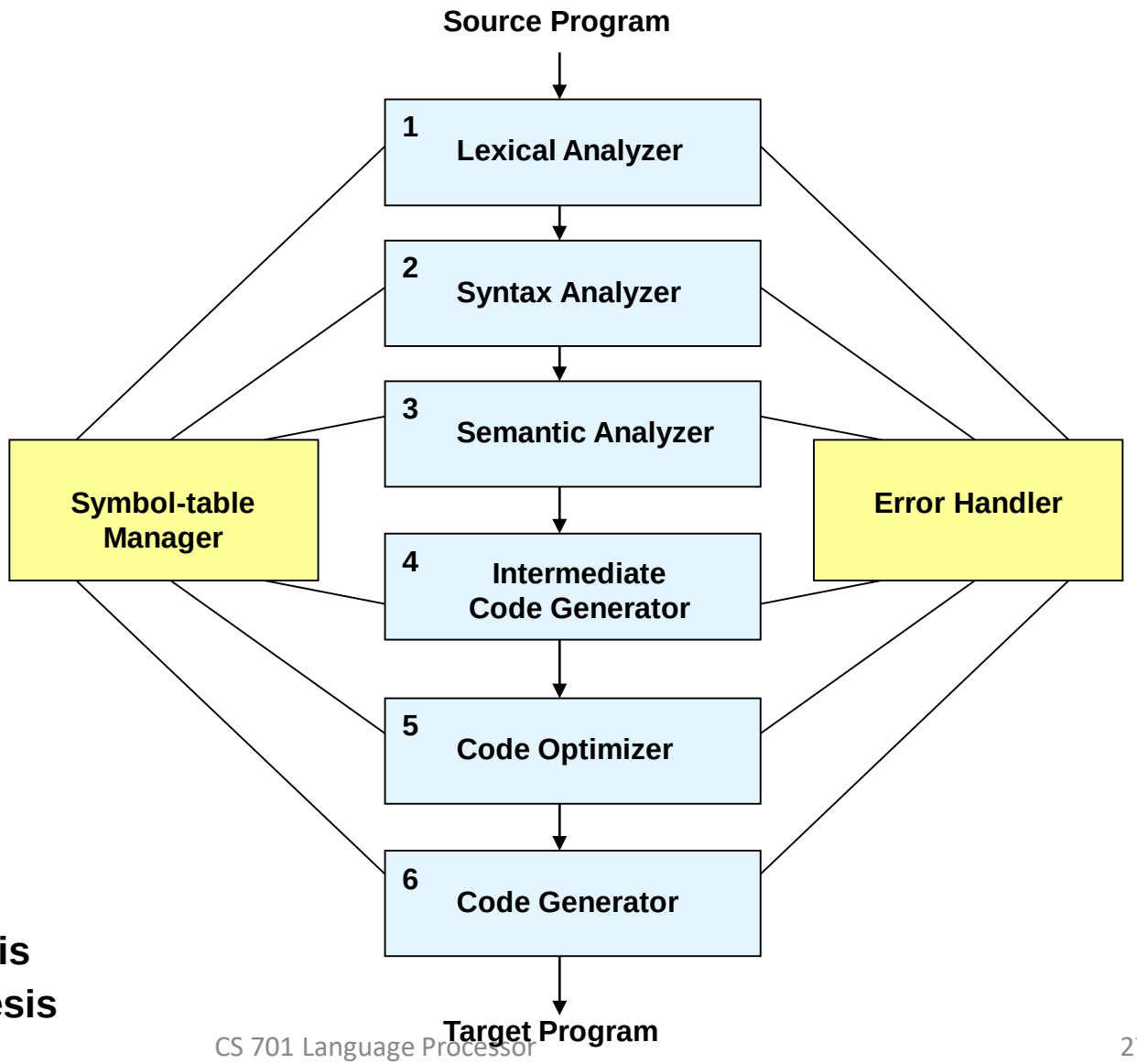
- After syntax and semantic analysis some compilers generates an intermediate representation of the source program.
- intermediate representation can be thought as a program for an abstract machine.
- This intermediate representation should have two important properties

a) Easy to produce b) easy to translate into the target program.

Code optimization: The code optimization phase attempts to improve the intermediate code so that faster running machine code will result.

Code generation: The final phase of compiler is the generation of target code, consisting normally of relocatable machine code or assembly code.

The Phases of a Compiler



1, 2, 3 : Analysis

4, 5, 6 : Synthesis

The Phases of a Compiler

Symbol table management:

A symbol table is a data structure containing a record for each identifier, with fields for the attribute of the identifier.

The data structure allows us to find the record for each identifier quickly and to store or retrieve data from that record quickly.

When an identifier in the source program is detected by the Lexical analyzer, the identifier is entered into the symbol table.

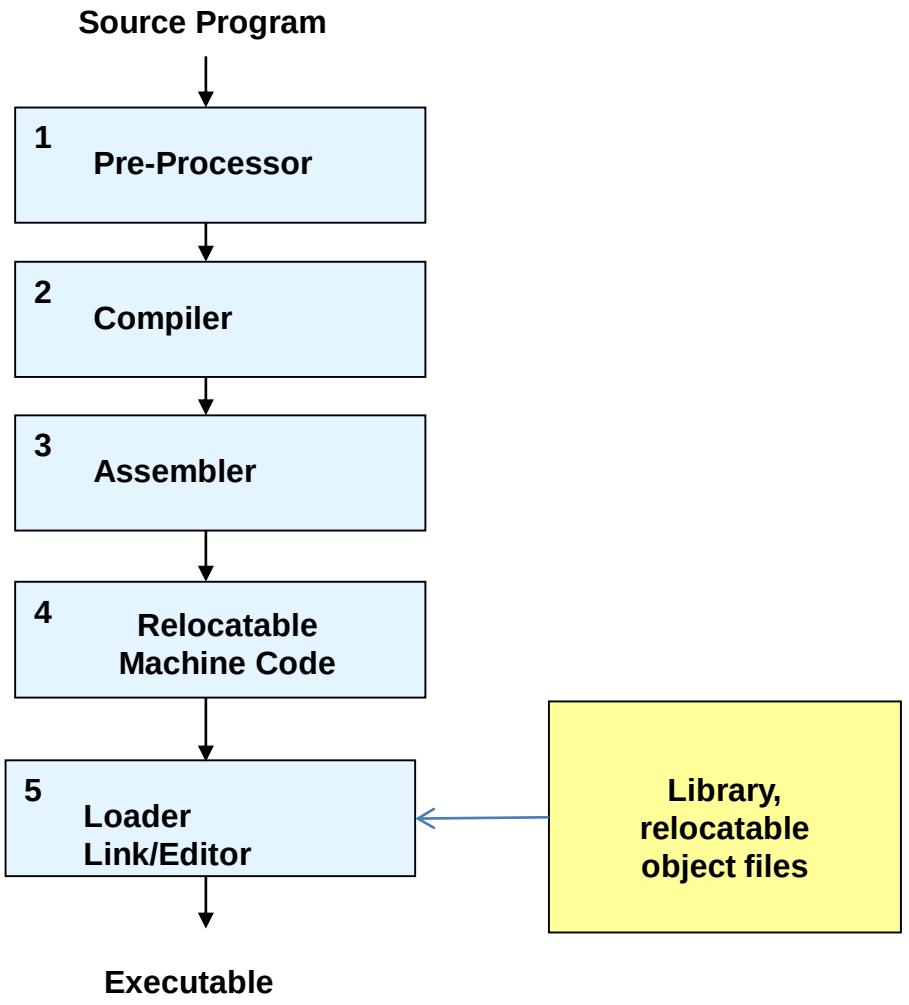
The remaining phases enter information about identifiers into the symbol table and then use this information in various ways.

Typical information stored in the symbol table includes a) Name of the variable b) Type c) Size etc.

Error detection & reporting: Each phase can encounter errors. However after detecting an error, a phase must somehow deal with that errors, so that the compilation can proceed. So that further errors in the source program to be detected.

The syntax & semantic analysis phase usually handle a large fraction of the errors detectable by the compiler.

Language-Processing System



Lexical Analyzer

- **Lexical Analyzer** reads the source program character by character and returns the *tokens* of the source program.
- A *token* describes a pattern of characters having same meaning in the source program. (such as identifiers, operators, keywords, numbers, delimeters and so on)

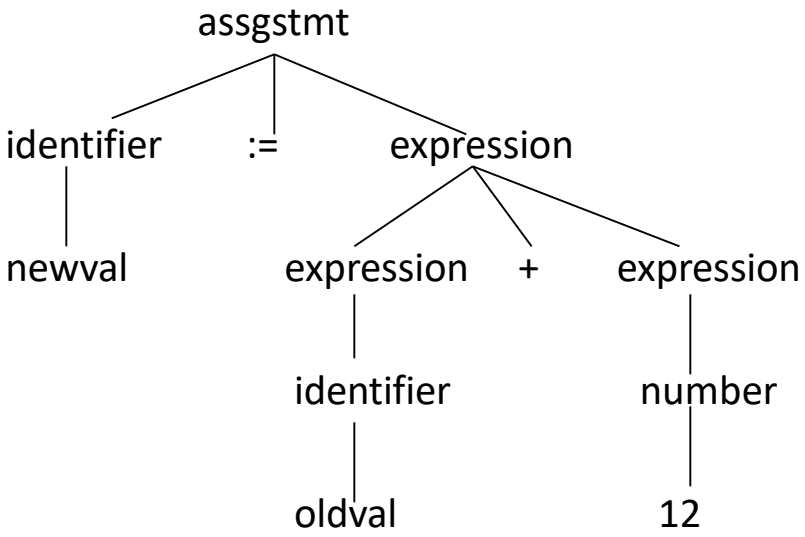
Ex: newval := oldval + 12 tokens:

newval	identifier
:=	assignment operator
oldval	identifier
+	add operator
12	a number

- Puts information about identifiers into the symbol table.
- Regular expressions are used to describe tokens (lexical constructs).
- A (Deterministic) Finite State Automaton can be used in the implementation of a lexical analyzer.

Syntax Analyzer

- A **Syntax Analyzer** creates the syntactic structure (generally a parse tree) of the given program.
- A syntax analyzer is also called as a **parser**.
- A **parse tree** describes a syntactic structure.



- In a parse tree, all terminals are at leaves.
- All inner nodes are non-terminals in a context free grammar.

Syntax Analyzer (CFG)

- The syntax of a language is specified by a **context free grammar** (CFG).
- The rules in a CFG are mostly recursive.
- A syntax analyzer checks whether a given program satisfies the rules implied by a CFG or not.
  - If it satisfies, the syntax analyzer creates a parse tree for the given program.

- Ex: We use BNF (Backus Naur Form) to specify a CFG

  assgstmt -> identifier := expression

  expression -> identifier

  expression -> number

  expression -> expression + expression

Syntax Analyzer versus Lexical Analyzer

- Which constructs of a program should be recognized by the lexical analyzer, and which ones by the syntax analyzer?
  - Both of them do similar things; But the lexical analyzer deals with simple non-recursive constructs of the language.
  - The syntax analyzer deals with recursive constructs of the language.
  - The lexical analyzer simplifies the job of the syntax analyzer.
  - The lexical analyzer recognizes the smallest meaningful units (tokens) in a source program.
  - The syntax analyzer works on the smallest meaningful units (tokens) in a source program to recognize meaningful structures in our programming language.

Parsing Techniques

- Depending on how the parse tree is created, there are different parsing techniques.
- These parsing techniques are categorized into two groups:
  - ***Top-Down Parsing,***
  - ***Bottom-Up Parsing***
- **Top-Down Parsing:**
  - Construction of the parse tree starts at the root, and proceeds towards the leaves.
  - Efficient top-down parsers can be easily constructed by hand.
  - Recursive Predictive Parsing, Non-Recursive Predictive Parsing (LL Parsing).
- **Bottom-Up Parsing:**
  - Construction of the parse tree starts at the leaves, and proceeds towards the root.
  - Normally efficient bottom-up parsers are created with the help of some software tools.
  - Bottom-up parsing is also known as shift-reduce parsing.
  - Operator-Precedence Parsing – simple, restrictive, easy to implement
  - LR Parsing – much general form of shift-reduce parsing, LR, SLR, LALR

Semantic Analyzer

- A semantic analyzer checks the source program for semantic errors and collects the type information for the code generation.
- Type-checking is an important part of semantic analyzer.
- Normally semantic information cannot be represented by a context-free language used in syntax analyzers.
- Context-free grammars used in the syntax analysis are integrated with attributes (semantic rules)
  - the result is a syntax-directed translation,
  - Attribute grammars
- Ex:

  newval := oldval + 12

  - The type of the identifier *newval* must match with type of the expression *(oldval+12)*

Intermediate Code Generation

- 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.
- Ex:

newval := oldval * fact + 1

id1 := id2 * id3 + 1

MULT id2,id3,temp1

ADD temp1,#1,temp2

MOV temp2,,id1

Intermediates Codes (Quadraples)

Code Optimizer (for Intermediate Code Generator)

- The code optimizer optimizes the code produced by the intermediate code generator in the terms of time and space.

- Ex:

```
MULT    id2,id3,temp1
ADD     temp1,#1,id1
```

Code Generator

- Produces the target language in a specific architecture.
- The target program is normally is a relocatable object file containing the machine codes.

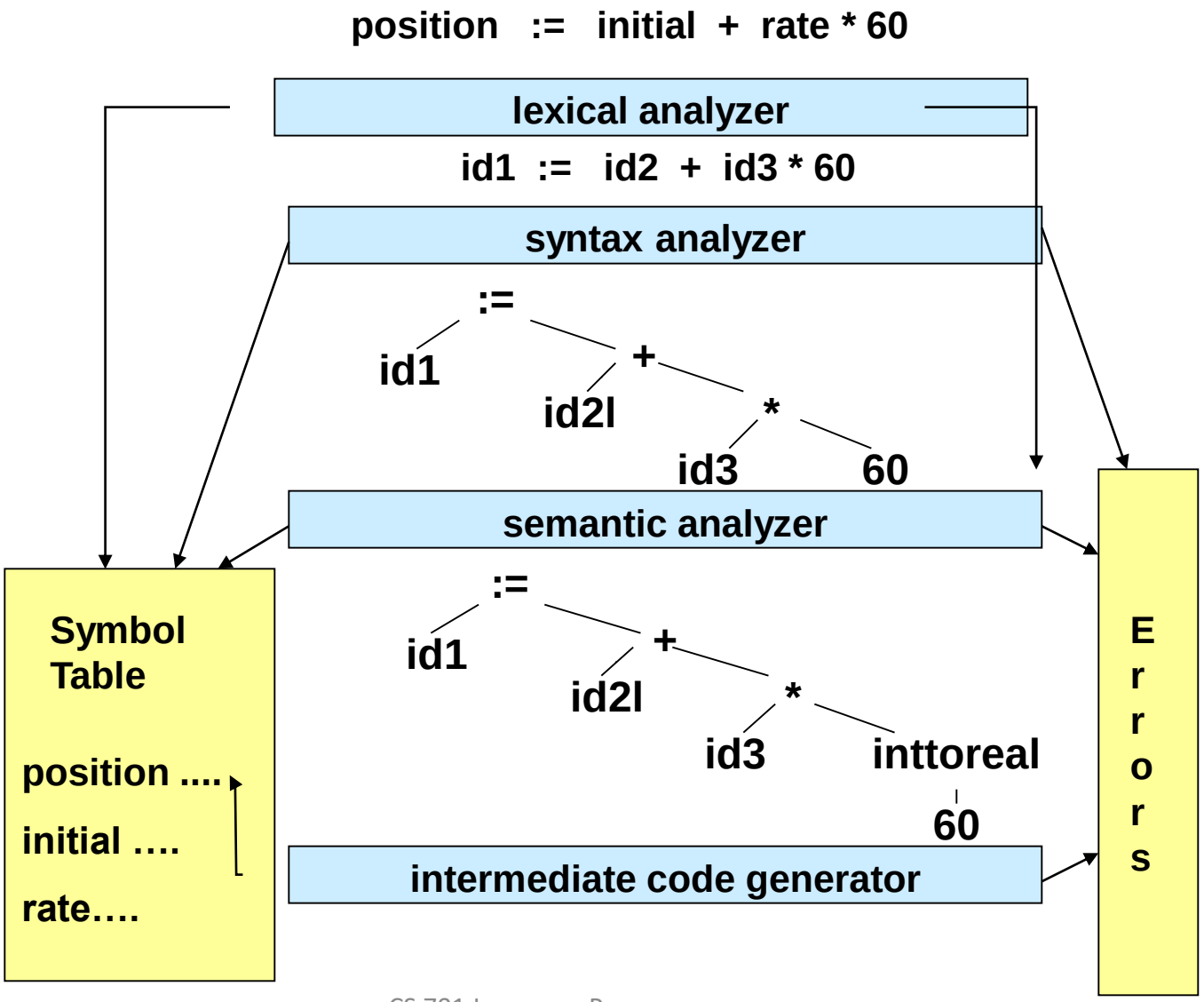
- Ex:

  (assume that we have an architecture with instructions whose at least one of its operands is

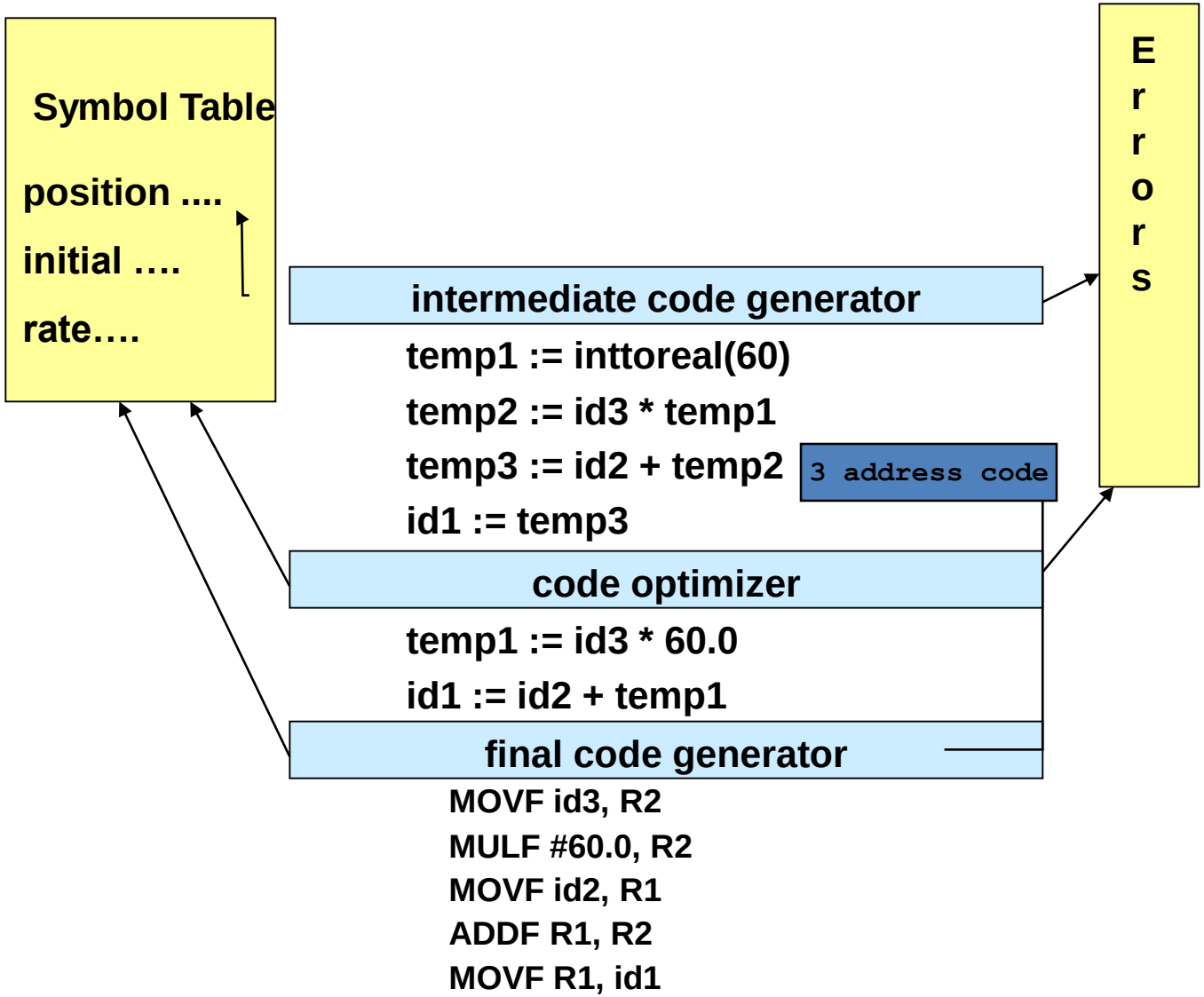
  a machine register)

```
MOVE    id2,R1
MULT    id3,R1
ADD     #1,R1
MOVE    R1,id1
```

Reviewing the Entire Process



Reviewing the Entire Process



Assemblers

- Assembly code: names are used for instructions, and names are used for memory addresses.

```
MOV a, R1
ADD #2, R1
MOV R1, b
```

- Two-pass Assembly:
  - First Pass: all identifiers are assigned to memory addresses (0-offset)
    e.g. substitute 0 for a, and 4 for b
  - Second Pass: produce relocatable machine code:

Exercise

1. Explain why a system may have several compiler but normally have a single linker.
2. “Code optimization is an optional phase of Compilation process”—Comment.

3. Show the difference between compiler output and interpreter output for each of the following source inputs:

```
a = 12;
b = 6;
c = a+b;
println (c,a,b);
```

```
a = 12;
b = 6;
if (a<b) println (a);
else println (b);
```

Example:

Show the compiler output and the interpreter output for the following C++ source code:

for (i=1; i<=4; i++) cout << i*3;

Solution:

Compiler

```
      LOD R1,='4'
     STO R1,Temp1
      MOV i,='1'
L1:   CMP i,Temp1
      BH L2     {Branch if i>Temp1}
      LOD R1,i
      MUL R1,='3'
      STO R1,Temp2
      PUSH Temp2
      CALL Write
      ADD i,='1' {Add 1 to i}
      B L1
L2:
```

Interpreter

3 6 9 12