Automata Theory and Introduction to Compilers.

By: MEGHA L
Assistant Professor.
Dept. of CSE, Dr. AIT

MODULE - 5
Introduction to Compiler Design

→ Introduction to Compiler Design:
1.2 → The structure of a compiler,
1.1 → Language Processors,
1.2 → Phases of Compilers,
1.5 → Applications of compiler Technology,
1.6 → Programming language Basics.
→ Lexical Analysis Phase of Compiler Design:
→ Role of lexical Analyzer,
→ Input Buffering,
→ Specification of Token,
→ Recognition of Token.

Text Book:-

1. Alfred W Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman, "Compilers - Principles, Techniques and Tools" Publisher: Pearson Education; second edition (1 January 2011) ISBN-10: 8131759024 ISBN-13: 978-8131759028

Chapter - 1
Introduction

Programming languages are notations for describing computations to people and to machines. The world depends on 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.

1.1 Language Processors

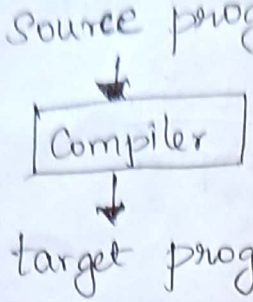

Figure : A compiler

A compiler is a 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.

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

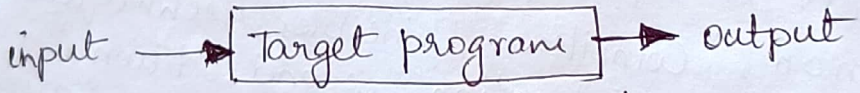

Figure : Running the target program

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

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.

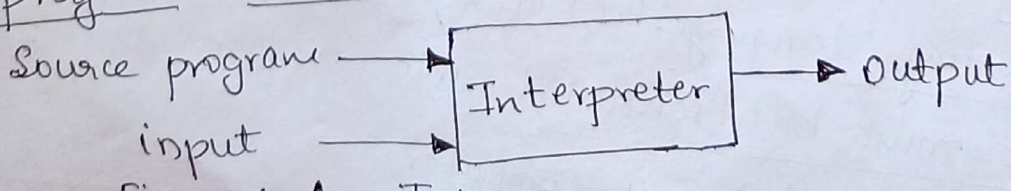

Figure : An Interpreter

NOTE :- Difference between compiler and Interpreter.

1. The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs.
2. An interpreter, can usually give better error diagnostics than a compiler, because it executes the source program statement by statement.

Example 1 :-

Java language processors combine compilation and interpretation, as shown in figure below.

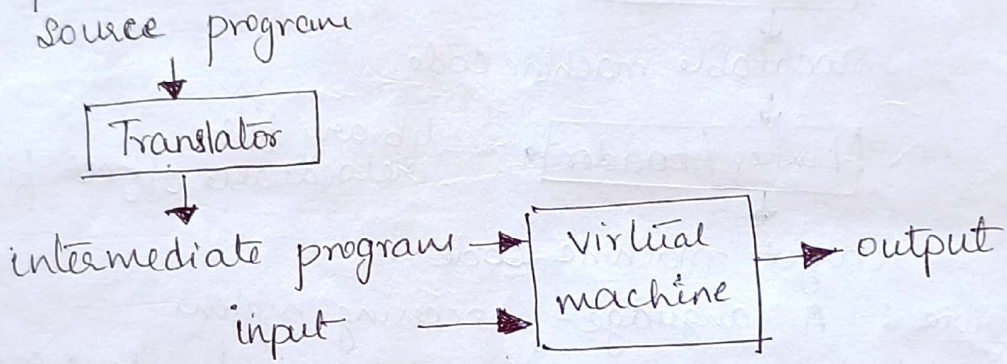

Figure : A hybrid compiler.

- 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.
- The task of collecting the source program is sometimes entrusted to a separate program, called a preprocessor. Preprocessor may also expand shorthands, called macros, into source language statements.

In addition to a compiler, several other programs may be required to create an executable target program as shown in figure below.

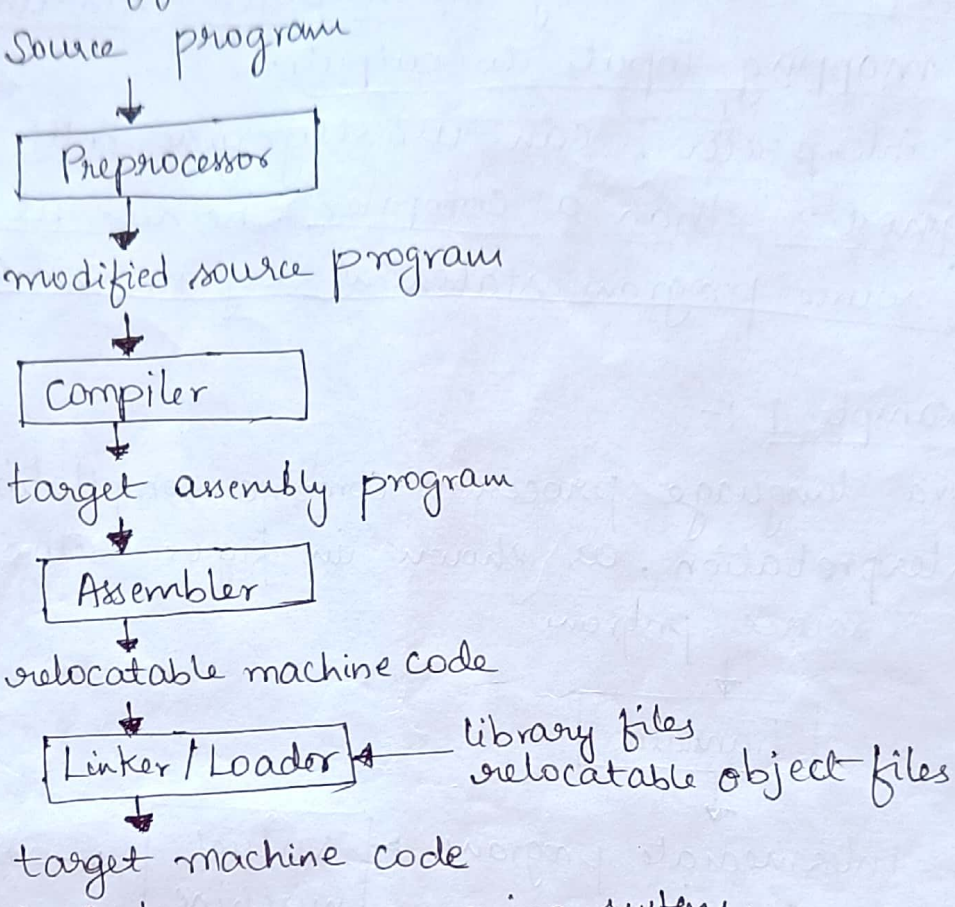

Figure : A language-processing system.

- The modified source program is then fed to a compiler. Compiler may produce an assembly-language program as its output, because assembly language is easier to produce as output and is easier to debug.
- The assembly language is then processed by a program called an assembler that produces relocatable machine code as its output.
- Large programs are often compiled in pieces, The linker resolves external memory addresses, where the code in one file may refer to a location in another file.
- The Loader then puts together all of the executable object files into memory for execution.

1.2 The Structure of a Compiler

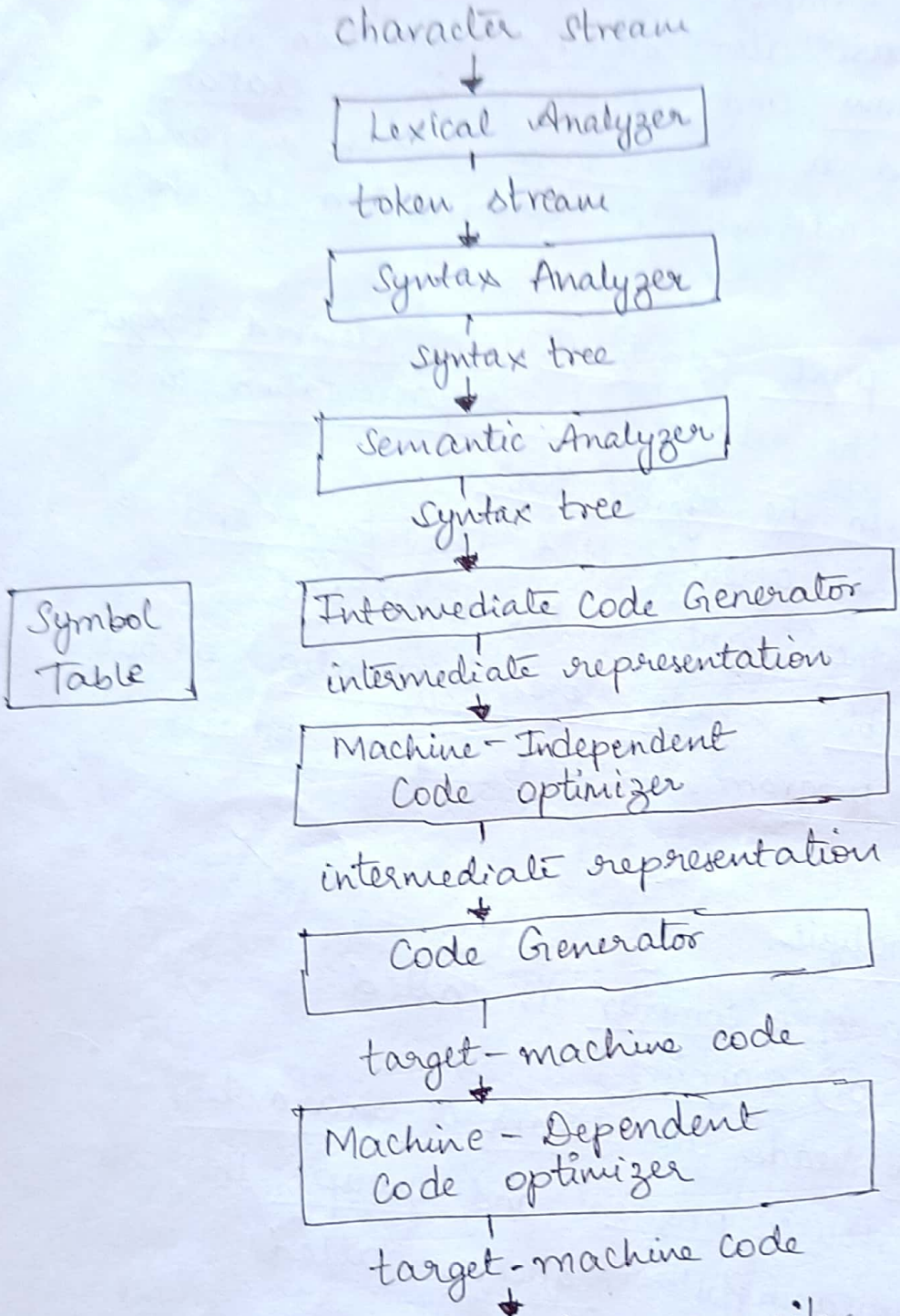

Figure: Phases of a compiler

- Compiler is mapped into two parts namely :-
  1. Analysis.
  2. Synthesis.
- 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.
- The analysis part also collects information about the source program and stores it in a data structures called a symbol Table, which is passed along with the intermediate representation to the Synthesis part.
- The Synthesis part constructs the desired target program from the intermediate representation and the information in the symbol table.
- Analysis part is often called the front end of the compiler, Synthesis part is the back end.
- The Symbol table, which stores information about the entire source program, is used by all phases of the compiler.

1.2.1 Lexical Analysis

- The first phase of a compiler is called Lexical analysis (or) scanning.
- Lexical analyzer 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 as output a "token" of the form that it passes on to the subsequent phase, syntax analysis.

<token-name, attribute-value>

- In the token, the first component token-name is the abstract symbol that is used during

Syntax analysis, and the second component attribute-value points to an entry in the Symbol table for this token.

- Information from the symbol-table entry is needed for semantic analysis and code generation.

For example, suppose a source program contains the assignment statement

position = initial + rate * 60 ——→ (1)

The characters in this assignment could be grouped into the following lexemes and mapped into the following tokens passed on to the syntax analyzer:

1.) position - is a lexeme, mapped into a token <id, 1>, where id is an abstract symbol standing for identifier and 1 points to the symbol-table entry for position. The symbol-table entry for an identifier holds information about the identifier, such as its name and type.

2). The assignment symbol = is a lexeme that is mapped into the token <=>.

3). initial is a lexeme that is mapped into the token <id, 2>, where 2 points to the symbol-table entry for initial.

4). + is a lexeme that is mapped into the token <+>.

5). rate is a lexeme that is mapped into the token <id, 3>, where 3 points to the symbol-table entry for rate.

6). * is a lexeme that is mapped into the token <*>.

7). 60 is a lexeme that is mapped into the token <60>.

• The blanks separating the lexemes would be discarded by the lexical analyzer.

<id, 1> <=> <id, 2> <+> <id, 3> <*> <60> → (2).

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

1.2.2 Syntax Analysis

The second phase of the compiler is syntax analysis (or) parsing.

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.

The root of the tree, labelled =, indicates that we must store the result.

1.2.3 Semantic Analysis

The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition.

It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.

An important part of semantic analysis is type checking, where the compiler checks that each operator has matching operands. The compiler must report an error if a floating-point number is used to index an array.

The language specification may permit some type conversions called coercions. For example, a binary arithmetic operator may be applied to a floating-point number and an integer, the compiler may convert or coerce the integer into a floating-point number. The output of the semantic analyzer has an extra node for the operator inttofloat, which explicitly converts its integer argument into a floating-point number.

1.2.4 Intermediate Code Generation

In process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms.

After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation, which we can think of as a program for an abstract machine.

This intermediate representation should has two important properties:

- it should be easy to produce and
- it should be easy to translate into the target machine.

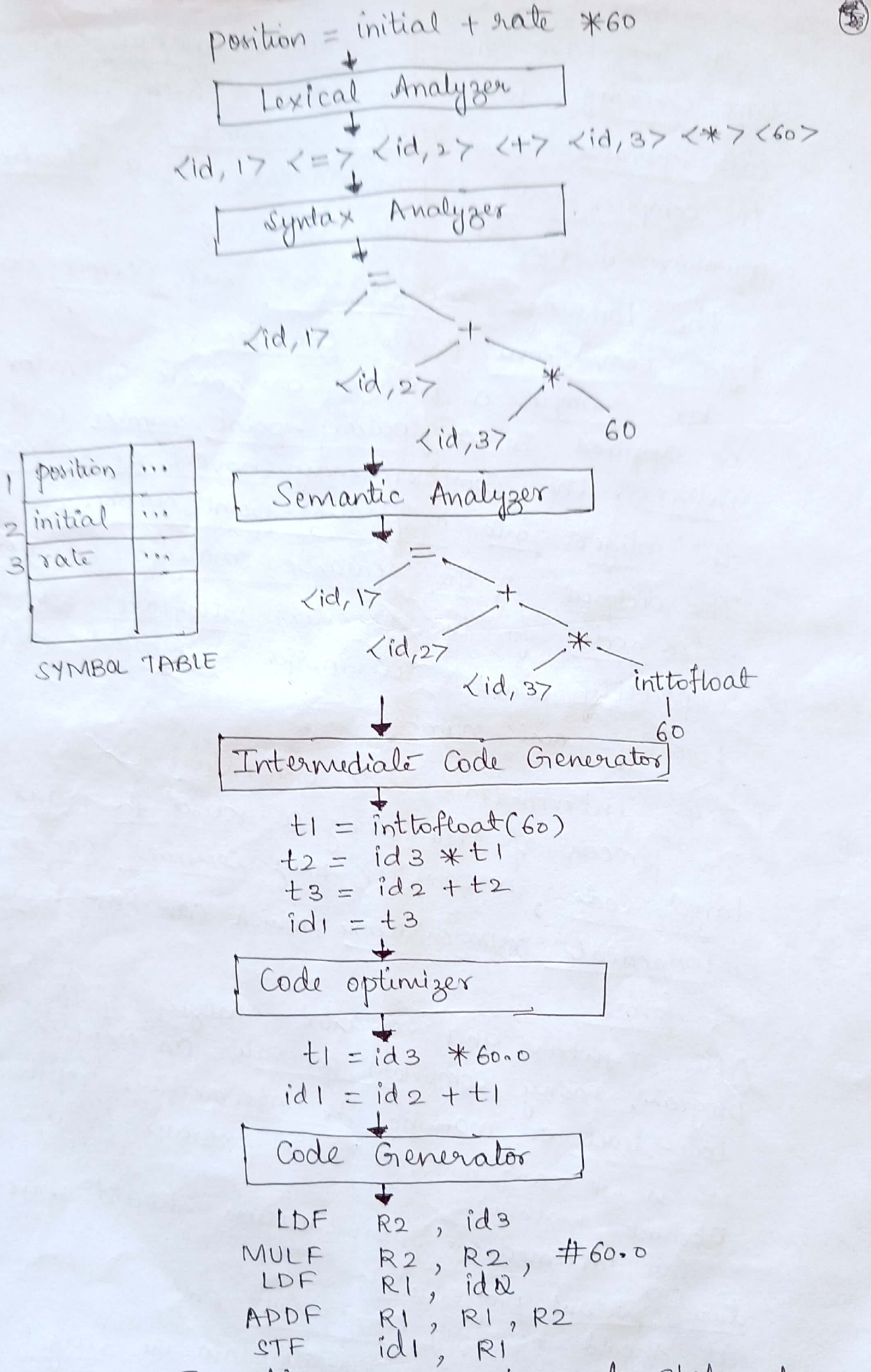

Figure : Translation of an assignment statement.

- We consider an intermediate form called three-address code, which consists of a sequence of assembly-like instructions with three operands per instruction.
- Each operand can act like a register.
- The output of the intermediate code generator consists of the three-address code sequence.

```
t1 = intTofloat(60)
t2 = id3 * t1          ——→ (3).
t3 = id2 + t2
id1 = t3.
```

1.2.5 Code Optimization

The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result.

- Straight forward algorithm generates the intermediate code.
- Simple intermediate code generation algorithm followed by code optimization is a reasonable way to generate good target code.
- 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 intTofloat operation can be eliminated by replacing the integer 60 by the floating point number 60.0.

```
t1 = id3 * 60.0        ——→ (4).
id1 = id2 + t1
```

- There is a great variation in the amount of code optimization different compilers perform. In those that do the most, the so-called "optimizing compilers", a significant amount of time is spent on this

phase.

1.2.6 Code Generation

The code generator 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.
- For example, using registers R1 and R2, the intermediate code in (4) might get translated into the machine code.

```
LDF   R2, id3
MULF  R2, R2, #60.0   ------> (5).
LDF   R1, id2
ADDF  R1, R1, R2
STF   id1, R1
```

- The first operand of each instruction specifies a destination.
- The F in each instruction tells us that it deals with floating-point numbers.
- The code in (5) loads the contents of address id3 into Register R2, then multiplies it with floating-point constant 60.0.
- # signifies that 60.0 is to be treated as an immediate constant.

- The third instruction moves id2 into register R1 and the fourth adds to it the value previously computed in register R2.
- Finally, the value in Register R1 is stored into the address of id1, so the code correctly implements the assignment statement (1).
- Storage-allocation decisions are made either during intermediate code generation or during code generation.

1.2.7 Symbol-Table Management

An essential function of a compiler is to record the variable names used in the source program and Collect information about various attributes of each name. These attributes may provide information about the storage allocated for a name, its type, its scope, and in the case of procedure names, such things as the number and types of its arguments, the method of passing each argument, and the type returned.

The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name.

1.2.8 The Grouping of phases into Passes

In implementation, activities from several phases may be grouped together into a pass that reads an input file and writes an output file.

For example, the front-end phases of lexical analysis, syntax analysis, semantic analysis, and

intermediate code generation might be grouped together into one pass.

- Code optimization might be an optional pass.
- Then there could be a back-end pass consisting of code generation for a particular target machine.

1.2.9 Compiler-construction Tools

The compiler writer, like any software developer, can profitably use modern development environments containing tools such as editors, debuggers, version managers, profilers, test harnesses, and so on.

Some commonly used compiler-construction tools include:

1. Parser generators - that automatically produce syntax analyzers from a grammatical description of a programming language.
2. Scanner generators - that produce lexical analyzers from a regular-expression description of the tokens of a language.
3. Syntax-directed translation engines that produce collections of routines for walking a parse tree and generating intermediate code.
4. Code-generator generators that produce a code generator from a collection of rules for translating each operation of the intermediate language into the machine language for a target machine,

5. Data-flow analysis engines that facilitate the gathering of information about how values are transmitted from one part of a program to each other part. Data-flow analysis is a key part of code optimization.

6. Compiler-construction toolkits that provide an integrated set of routines for constructing various phases of a compiler.

1.5 Applications of compiler Technology

Compiler design is not only about compilers, and many people use the technology learned by studying compilers in school, written a compiler for a major programming language.

1.5.1 Implementation of high-level Programming Languages.

A high-level programming language defines a programming abstraction:

- the programmer expreses an algorithm using the language, and
- the compiler must translate that program to the target program.

- Generally, high-level programming languages are easier to program in, but are less efficient, i.e (target programs run more slowly).
- Programmers using a low-level language have more control over a computation and can, in principle, produce more efficient code.
- lower-level programs are harder to write and -

coorse still - less portable, more prone to errors, and harder to maintain.

Example 1.2:

The keyword Register in C programming language is an early example of the interaction between Compiler technology and language evolution. C language was created in mid 1970's.

- In fact programs that use the register keyword may lose efficiency, because programmers often are not the best judge of very low-level matters like register allocation.
- Hardwiring low-level resource management decisions like register allocation may in fact hurt performance.
- C was the predominant systems programming language of the 80's, C++ in 90's, Java introduced in 1995.
- All programming languages including C, Fortran and Cobol, support user-defined aggregate data types, such as array and structures, and high-level control flow, such as loops and procedure invocations.
- A body of compiler optimizations, known as data-flow optimizations, developed to analyze the flow of data through the program and removes redundancies across these constructs.

- Object Orientation introduced in simula in 1967, and has been incorporated in languages such as small talk, C++, C#, and java.

The Key ideas behind object orientation are:

1. Data abstraction and
2. Inheritance of properties.

- Java has many features that make programming easier, Java language is type-safe (i.e, an object cannot be used as an object of an unrelated type).
- All array accesses are checked to ensure that they lie within the bounds of the array.
- Java has no pointers and does not allow pointer arithmetic, It has a built-in garbage-collection facility that automatically frees the memory of variables that are no longer in use. All these features make programming easier, they incur a run-time overhead.
- Compiler optimizations have been developed to reduce the overhead (i.e, by eliminating unnecessary range checks and by allocating objects that are not accessible beyond a procedure on the stack instead of the heap.
- Effective algorithms also have been developed to minimize the overhead of garbage collection.
- Java is designed to support portable and mobile code.

- Programs are distributed as Java bytecode, which must either be interpreted or compiled into native code dynamically, i.e at run time.

1.5.2. Optimizations for Computer Architectures

The rapid evolution of computer architectures has also led to an insatiable demand for new compiler technology.

- Almost all high-performance systems take advantages of the same two basic techniques:
  1. parallelism and
  2. Memory hierarchies.
- Parallelism can be found at several levels: at the instruction level, where multiple operations are executed simultaneously and at the processor level, where different threads of the same application are run on different processors.
- Memory hierarchies are a response to the basic limitation that we can build very fast storage or very large storage, but not storage that is both fast and large.

Parallelism:

- All modern microprocessors exploit instruction-level parallelism. (parallelism can be hidden from the programmer).
- Programs are written as if all instructions were executed in sequence, hardware dynamically checks for dependancies

in sequential instruction steam and issues them in parallel when possible. the machine includes a hardware scheduler that can change the instruction ordering to increase the parallelism in the programs.

whether the hardware reorders the instructions or not, Compilers can rearrange the instructions to make instruction-level parallelism more effective.

Instruction-level parallelism can also appear explicitly in the instruction set.

VLIW (Very long Instruction Word) machines have instructions that can issue multiple operations in parallel.

In Intel IA64 is a well-known example of such an architecture.

Compiler techniques have been developed to generate code automatically for such machines from sequential programs.

Multiprocessors have also become prevalent; even personal computers often have multiple processors. Programmers can write multithreaded code for multiprocessors, or parallel code can be automatically generated by a compiler from conventional sequential programs.

Parallelization techniques have been developed to translate automatically sequential scientific programs into multiprocessor code.

Memory Hierarchies:

A Memory hierarchy consists of several levels of storage with different speeds and sizes, with the level

closest to the processor being the fastest but smallest.

- The average memory-access time of a program is reduced if most of its accesses are satisfied by the faster levels of the hierarchy.
- Both parallelism and memory hierarchy improve the potential performance of a machine, but they must be harnessed effectively by the compiler to deliver real performance on an application.
- Memory hierarchies are found in all machines.
- A processor usually has a small number of registers consisting of hundreds of bytes, several levels of caches containing kilobytes to megabytes; physical memory containing megabytes to gigabytes, and finally secondary storage that contains gigabytes and beyond.
- The performance of a system is often limited not by the speed of the processor but by the performance of the memory subsystem.
- While compilers traditionally focus on optimizing the processor execution, more emphasis is now placed on making the memory hierarchy more effective.
- Using registers effectively is probably the single most important problem in optimizing a program.
- Cache-management policies implemented by hardware are not effective in some cases (in scientific code that has large data structures (arrays)).

It is possible to improve the effectiveness of the memory hierarchy by changing the layout of the data, or changing the order of instructions accessing the data.

- We can also change the layout of code to improve the effectiveness of instruction caches.

1.5.3 Design of New Computer Architectures.

In early days of computer architecture design, compilers were developed after the machines were built. That has changed. Since programming in high-level languages is the norm, the performance of a computer system is determined not by its raw speed but also by how well compilers can exploit its features.

Thus, in modern computer architecture development, compilers are developed in the processor-design stage, and compiled code, running on simulators, is used to evaluate the proposed architectures features.

RISC :-

One of the best known examples of how compilers influenced the design of computer architecture was the invention of the RISC (Reduced Instruction-Set Computer) architecture.

Prior to this invention, the trend was to develop progressively complex instructions sets intended to make assembly programming easier;

these architectures were known as CISC (Complex Instruction-set Computer).

For example, CISC Instruction sets include Complex memory-addressing modes to support data-structure accesses and procedure-invocation instructions that save registers and pass parameters on the stack.

Compiler optimizations can often reduces these instructions to a small number of simple operations by eliminating the redundancies across complex instructions.

Compilers can use them effectively and the hardware is much easier to optimize.

Most general-purpose processor architectures, including PowerPC, SPARC, MIPS, Alpha, and PA-RISC, are based on the RISC concept.

X86 architecture - the most popular microprocessor - has a CISC instruction set, many of the ideas developed for RISC machines are used in the implementation of the processor itself.

Moreover, the most effective way to use a high-performance X86 machine is to use just its simple instructions.