



CS6109 – COMPILER DESIGN

Module – 1

Presented By

Dr. S. Muthurajkumar,
Assistant Professor,
Dept. of CT, MIT Campus,
Anna University, Chennai.

MODULE - 1

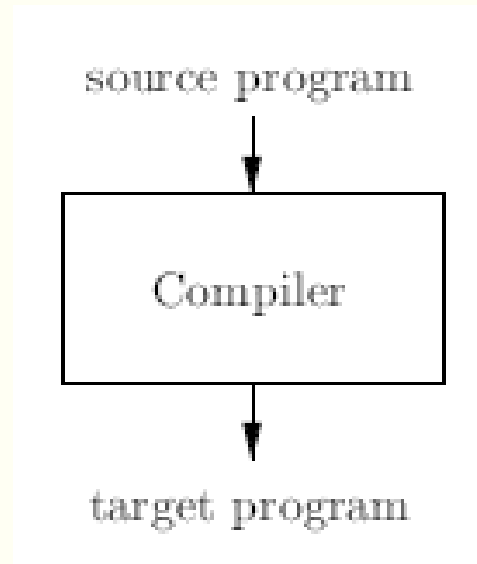
- Phases of the compiler
- Compiler construction tools
- Role of assemblers
- Macro-processors
- Loaders
- Linkers

COMPILER

- 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.

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.



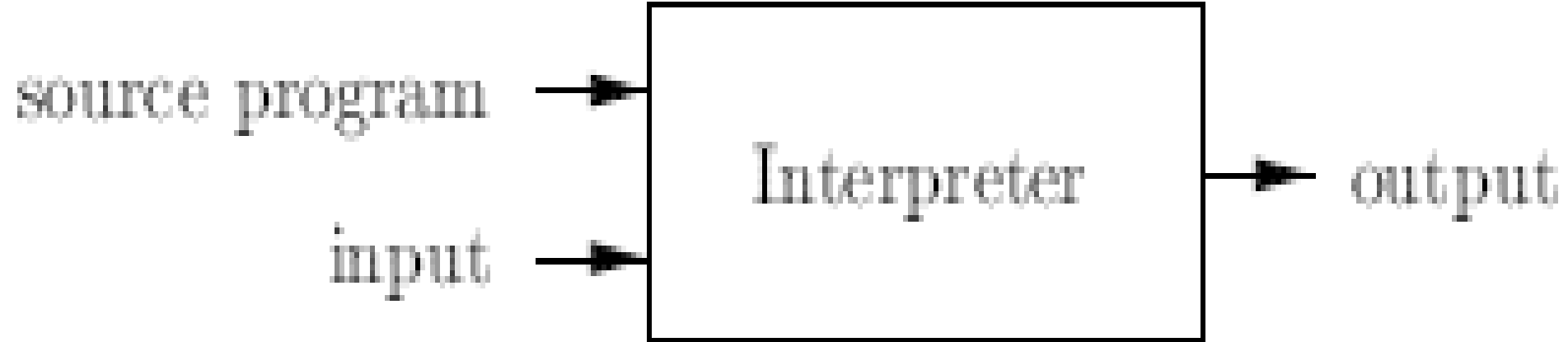
RUNNING THE TARGET PROGRAM

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



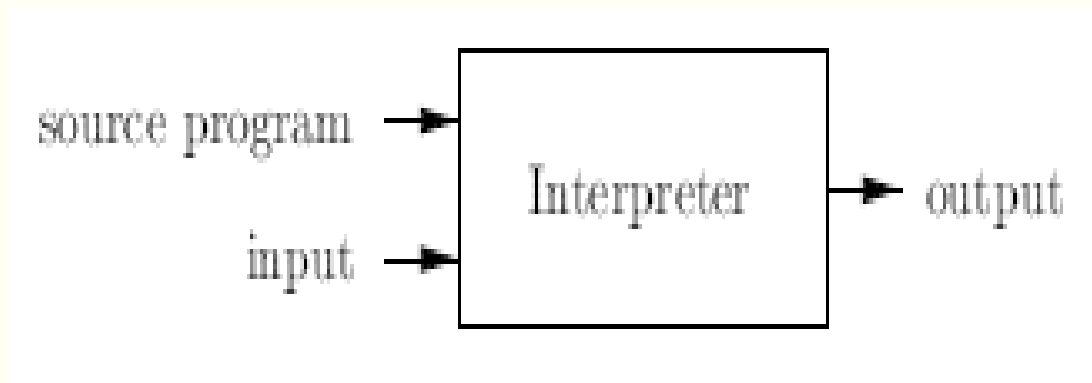
INTERPRETER

- 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



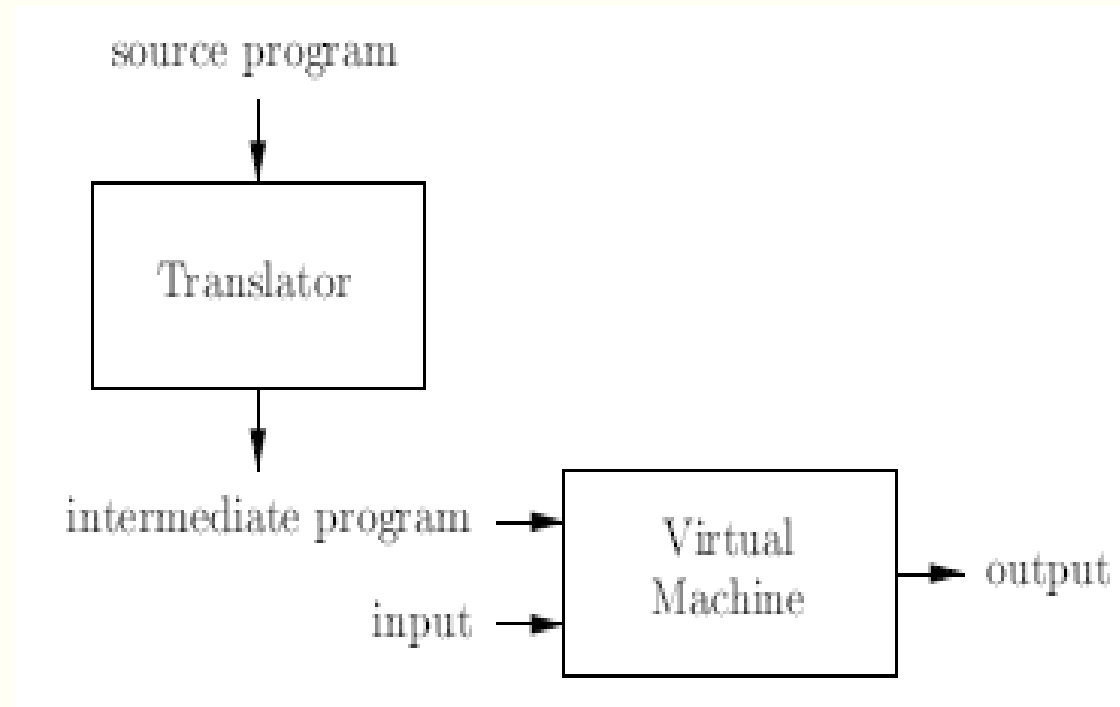
INTERPRETER

- 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.



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.
- A benefit of this arrangement is that bytecodes compiled on one machine can be interpreted on another machine, perhaps across a network.

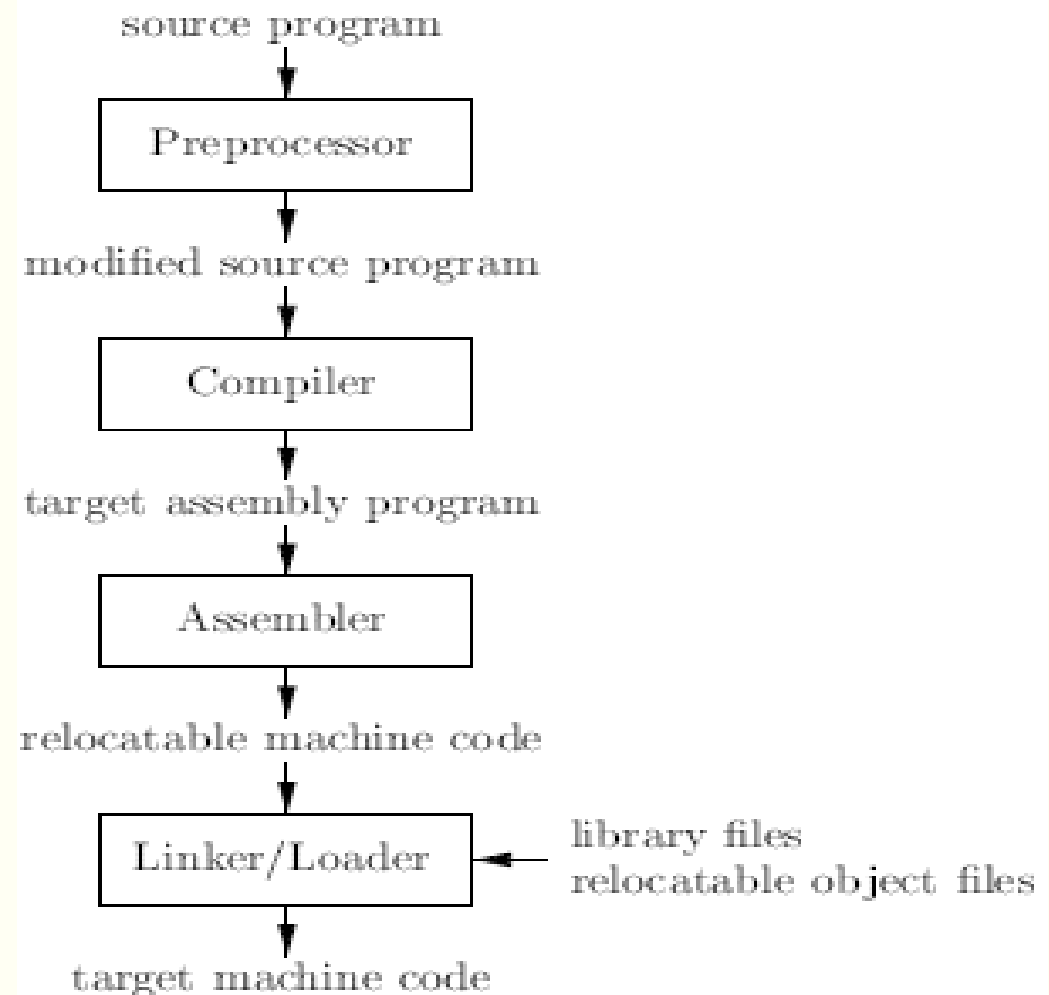


LANGUAGE-PROCESSING SYSTEM

- A source program may be divided into modules stored in separate files.
- The task of collecting the source program is sometimes entrusted to a separate program, called a pre-processor.
- The pre-processor may also expand short hands, called macros, into source language statements.
- The modified source program is then fed to a compiler.
- The 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.

LANGUAGE-PROCESSING SYSTEM

- Large programs are often compiled in pieces, so the relocatable machine code may have to be linked together with other relocatable object files and library files into the code that actually runs on the machine.
- 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.



STRUCTURE OF A COMPILER/PHASES OF A COMPILER

- Two Parts
 - Analysis
 - Synthesis

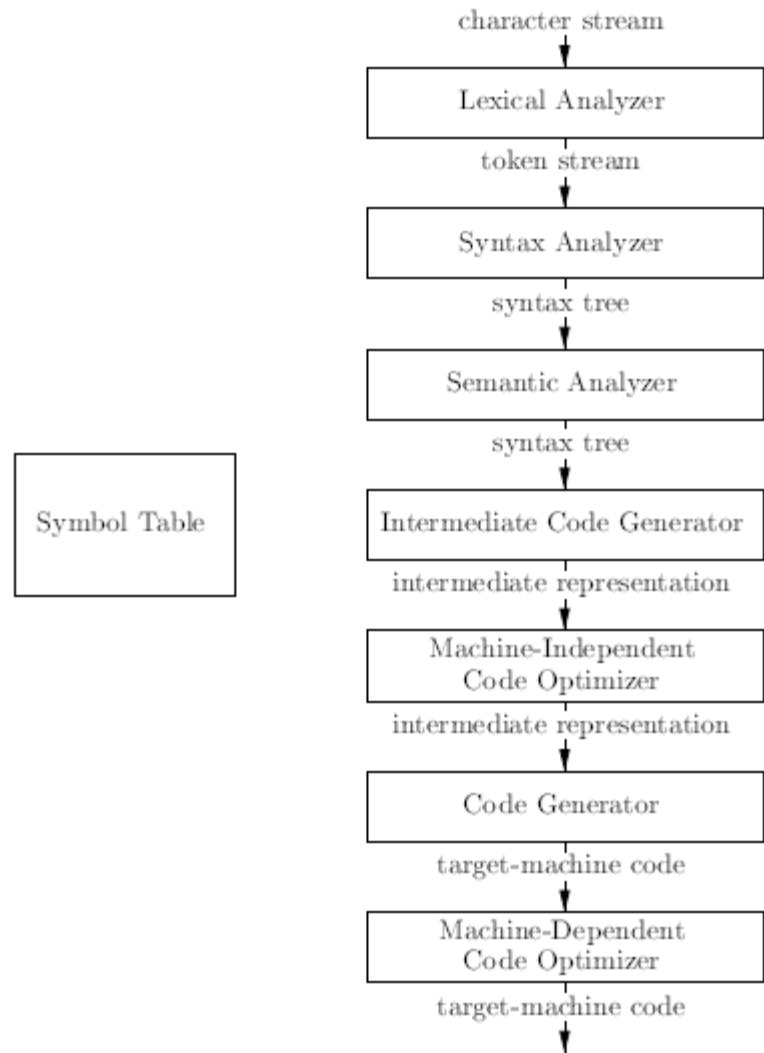
ANALYSIS

- 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.

SYNTHESIS

- 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.
- It operates as a sequence of phases, several phases may be grouped together, and the intermediate representations between the grouped phases need not be constructed explicitly.
- The symbol table, which stores information about the entire source program, is used by all phases of the compiler.
- Some compilers have a machine-independent optimization phase between the front end and the back end.
- The purpose of this optimization phase is to perform transformations on the intermediate representation, so that the back end can produce a better target program than it would have otherwise produced from an unoptimized intermediate representation.

PHASES OF A COMPILER



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. For each lexeme, the lexical analyzer produces as output a token of the form

<token-name; attribute-value>

- that it passes on to the subsequent phase, syntax analysis.
- In the token, the first component token-name is an 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
- $\text{position} = \text{initial} + \text{rate} * 60$

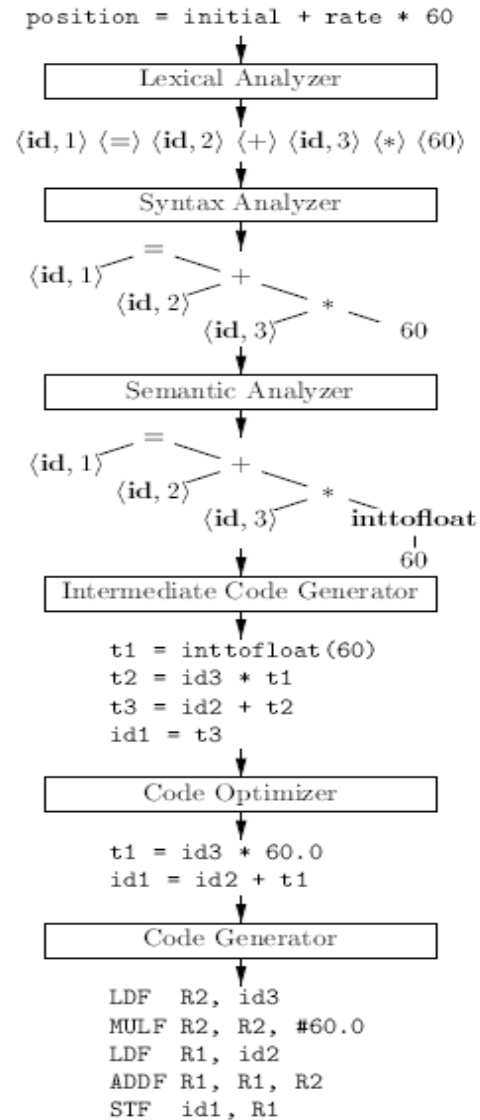
LEXICAL ANALYSIS

- The first phase of a compiler is called lexical analysis or scanning. The lexical
For example, suppose a source program contains the assignment statement
- `position = initial + rate * 60`
- `Position` \rightarrow `id1`
- `=` \rightarrow token (assignment symbol)
- `Initial` \rightarrow `id2`
- `+` \rightarrow token (plus operator)
- `Rate` \rightarrow `id3`
- `*` \rightarrow token (multiplication operator)
- `60` \rightarrow token (number 60)
- `<id1> <=> <id2> <+> <id3> <*> <60>`

LEXICAL ANALYSIS

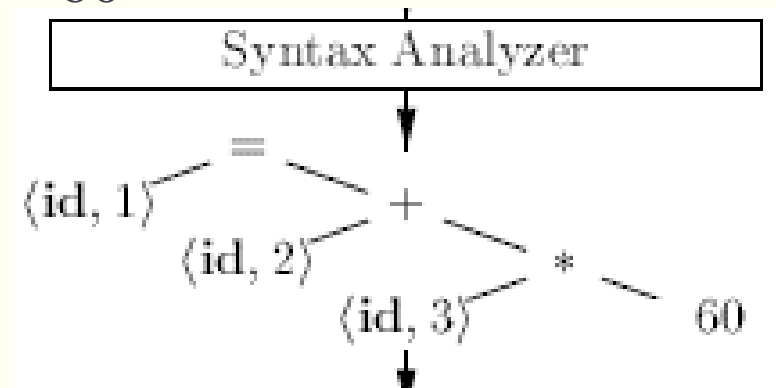
1	position	...
2	initial	...
3	rate	...

SYMBOL TABLE



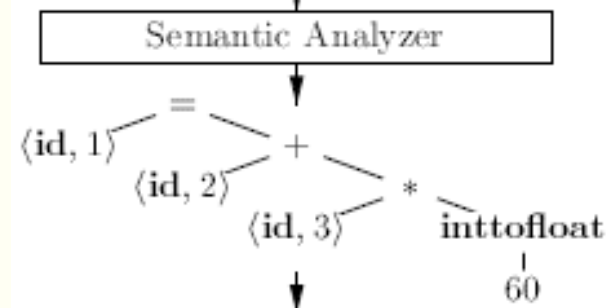
SYNTAX ANALYSIS

- 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.
- $\text{position} = \text{initial} + \text{rate} * 60$



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.
- 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.



INTERMEDIATE CODE GENERATION

- 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 have two important properties: it should be easy to produce and it should be easy to translate into the target machine.


Intermediate Code Generator

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

INTERMEDIATE CODE GENERATION

- 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.
- `t1 = inttofloat(60)`
- `t2 = id3 * t1`
- `t3 = id2 + t2`
- `id1 = t3`

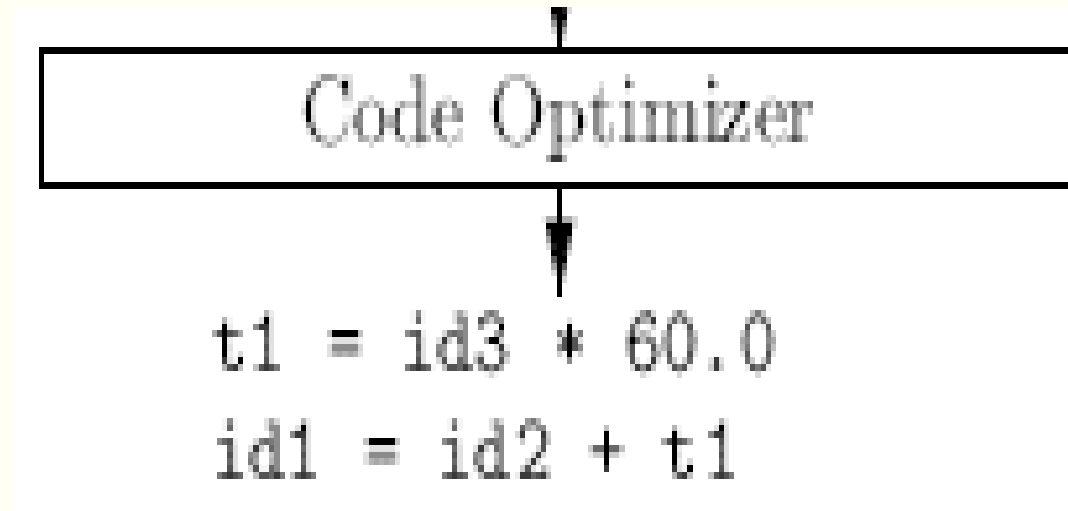
Intermediate Code Generator



```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

CODE OPTIMIZATION

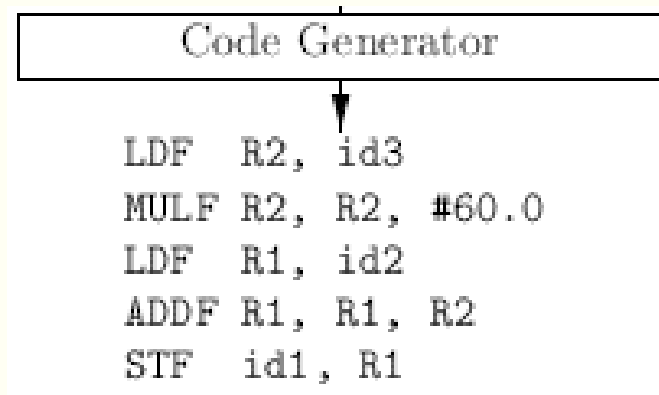
- The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result.
- Usually better means faster, but other objectives may be desired, such as shorter code, or target code that consumes less power.
- `t1 = id3 * 60.0`
- `id1 = id2 + t1`



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.
- 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



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 (where in the program its value may be used), and in the case of procedure names, such things as the number and types of its arguments, the method of passing each argument (for example, by value or by reference), 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.
- 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

COMPILER CONSTRUCTION TOOLS

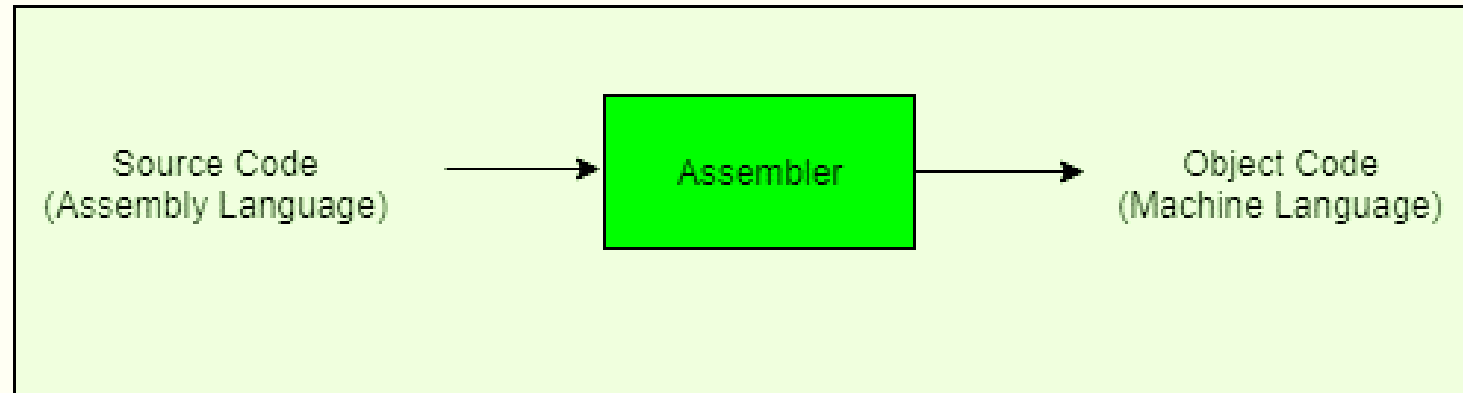
- The compiler writer, like any software developer, can profitably use modern software development environments containing tools such as language editors, debuggers, version managers, profilers, test harnesses, and so on.
- In addition to these general software-development tools, other more specialized tools have been created to help implement various phases of a compiler.
 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.

COMPILER CONSTRUCTION TOOLS

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.

ROLE OF ASSEMBLERS

- Assembler is a program for converting instructions written in low-level assembly code into relocatable machine code and generating along information for the loader.



- An assembler translates assembly language programs into machine code.
- The output of an assembler is called an object file, which contains a combination of machine instructions as well as the data required to place these instructions in memory.

ROLE OF ASSEMBLERS

- **Assembler divide these tasks in two passes:**
- **Pass-1:**
 - Define symbols and literals and remember them in symbol table and literal table respectively.
 - Keep track of location counter
 - Process pseudo-operations
- **Pass-2:**
 - Generate object code by converting symbolic opcode into respective numeric opcode
 - Generate data for literals and look for values of symbols

MACRO-PROCESSORS

- A macro processor is a system software.
- Macro is that the Section of code that the programmer writes (defines) once, and then can use or invokes many times.
- A Macro instruction is the notational convenience for the programmer.
- For every occurrence of macro the whole macro body or macro block of statements gets expanded in the main source code.
- Thus Macro instructions make writing code more convenient.

MACRO-PROCESSORS

- **Salient features of Macro Processor:**
- **Macro** represents a group of commonly used statements in the source programming language.
- Macro Processor replaces each macro instruction with the corresponding group of source language statements. This is known as the expansion of macros.
- Using Macro instructions programmer can leave the mechanical details to be handled by the macro processor.
- Macro Processor designs are not directly related to the computer architecture on which it runs.
- Macro Processor involves definition, invocation, and expansion.

MACRO-PROCESSORS

- **Macro Definition and Expansion:**

Line	Label	Opcode	Operand
▪ 5	COPY	START	0
▪ 10	RDBUFF	MACRO	&INDEV, &BUFADR
▪ 15			
▪ .			
▪ .			
▪ 90			
▪ 95		MEND	

MACRO-PROCESSORS

- **Macro Definition and Expansion:**
- **Line 10:**
- RDBUFF (Read Buffer) in the Label part is the name of the Macro or definition of the Macro. &INDEV and &BUFADR are the parameters present in the Operand part. Each parameter begins with the character &.
- **Line 15 – Line 90:**
- From Line 15 to Line 90 Macro Body is present. Macro directives are the statements that make up the body of the macro definition.
- **Line 95:**
- MEND is the assembler directive that means the end of the macro definition.

MACRO-PROCESSORS

- **Macro Invocation:**

Line	Label	Opcode	Operand
▪ 180	FIRST	STL	RETADR
▪ 190	CLOOP	RDBUFF	F1, BUFFER
▪ 15			
▪ .			
▪ .			
▪ 255		END	FIRST

- Line 190:

- RDBUFF is the Macro invocation or Macro Call that gives the name of the macro instruction being invoked and F1, BUFFER are the arguments to be used in expanding the macro. The statement that form the expansion of a macro are generated each time the macro is invoked.

LOADERS

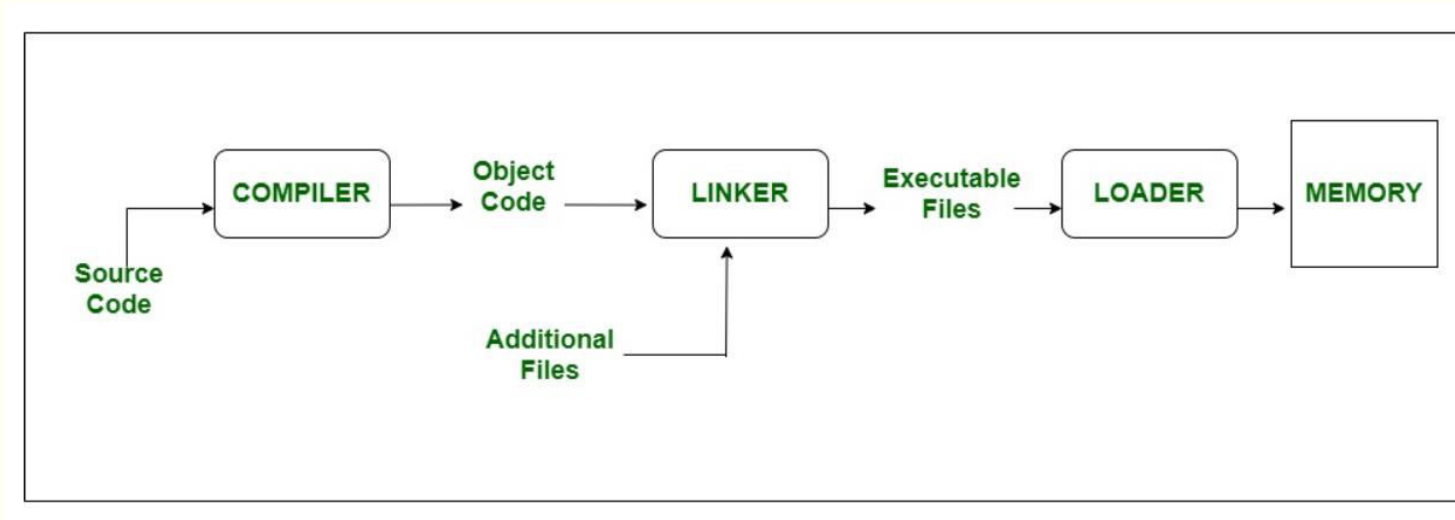
- Loader is a part of operating system and is responsible for loading executable files into memory and execute them.
- It calculates the size of a program (instructions and data) and creates memory space for it.
- It initializes various registers to initiate execution.

LOADERS

- The loader is special program that takes input of object code from linker, loads it to main memory, and prepares this code for execution by computer.
- Loader allocates memory space to program.
- Even it settles down symbolic reference between objects.
- It is in charge of loading programs and libraries in operating system.
- The embedded computer systems don't have loaders. In them, code is executed through ROM.
- There are following various loading schemes:
 1. Absolute Loaders
 2. Reloacting Loaders
 3. Direct Linking Loaders
 4. Bootstrap Loaders

LINKERS

- Linker is a computer program that links and merges various object files together in order to make an executable file.
- All these files might have been compiled by separate assemblers.
- The major task of a linker is to search and locate referenced module/routines in a program and to determine the memory location where these codes will be loaded, making the program instruction to have absolute references.



LINKERS

- A linker is special program that combines the object files, generated by compiler/assembler, and other pieces of codes to originate an executable file have. exe extension.
- In the object file, linker searches and append all libraries needed for execution of file.
- It regulates the memory space that will hold the code from each module.
- It also merges two or more separate object programs and establishes link among them.
- Generally, linkers are of two types :
 1. Linkage Editor
 2. Dynamic Linker

DIFFERENCE BETWEEN LINKER AND LOADER

S. No.	LINKER	LOADER
1	The main function of Linker is to generate executable files.	Whereas main objective of Loader is to load executable files to main memory.
2	The linker takes input of object code generated by compiler/assembler.	And the loader takes input of executable files generated by linker.
3	Linking can be defined as process of combining various pieces of codes and source code to obtain executable code.	Loading can be defined as process of loading executable codes to main memory for further execution.
4	Linkers are of 2 types: Linkage Editor and Dynamic Linker.	Loaders are of 4 types: Absolute, Relocating, Direct Linking, Bootstrap.
5	Another use of linker is to combine all object modules.	It helps in allocating the address to executable codes/files.
6	Linker is also responsible for arranging objects in program's address space.	Loader is also responsible for adjusting references which are used within the program.

REFERENCE

1. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, “Compilers: Principles, Techniques and Tools”, Second Edition, Pearson Education Limited, 2014.
2. Randy Allen, Ken Kennedy, “Optimizing Compilers for Modern Architectures: A Dependence-based Approach”, Morgan Kaufmann Publishers, 2002.
3. Steven S. Muchnick, “Advanced Compiler Design and Implementation”, Morgan Kaufmann Publishers - Elsevier Science, India, Indian Reprint, 2003.
4. Keith D Cooper and Linda Torczon, “Engineering a Compiler”, Morgan Kaufmann Publishers, Elsevier Science, 2004.
5. V. Raghavan, “Principles of Compiler Design”, Tata McGraw Hill Education Publishers, 2010.
6. Allen I. Holub, “Compiler Design in C”, Prentice-Hall Software Series, 1993.

