Q.1 Give brief note on compilers

Ans.

- **Compiler**: is a computer program that reads a program from written in one language called the source language and translates it into an equivalent program in another language which called target language.
- In other word compiler are a type of translator primarily work on computers or other digital devices.
- And The name compiler is primarily used for programs that translate source code from a high-level programming language to a lower level programming language such as like assembly, object code, machine code.
- A compiler is likely to perform many or all of the following operations:

  1. Pre-processing
  2. lexical analysis
  3. parsing
  4. semantic analysis (syntax-directed translation)
  5. conversion of input programs to an intermediate representation
  6. code optimization and code generation.

- Compilers implement these operations in phases that promote efficient design and correct transformations of source input to target output.

- During this important part of translation process, the compiler reports to its user this presence of errors or incorrect compiler behavior in the source program
- And compiler implementers(maker) invest significant effort to ensure compiler correctness.
- In Compiler writing it includes programming language, machine architecture, language theory, algorithms and software engineering.

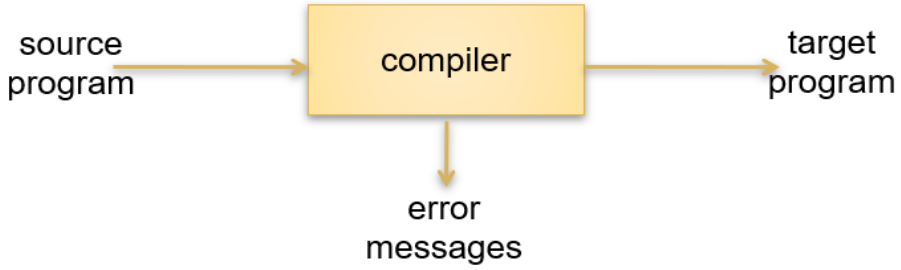


Fig: A compiler

- Compilers are not the only translators used to transform source programs.
- Compilers are classified as follows depending on how they are constructed or on what function they are supposed to perform.

⇨ Types of compiler:
  - Single pass
  - Multi pass
  - Load and go
  - Debugging
  - Optimizing

Q.2 compilers History.

Ans.

- Software for early computers was primarily written in assembly language for many years.
- Higher level programming languages were not invented until the benefits of being able to reuse software on different kinds of CPUs started to become significantly greater than the cost of writing a compiler.
- The very limited memory capacity of early computers also created many technical problems when implementing a compiler.
- Towards the end of the 1950s, machine-independent programming languages were first proposed.
- The first compilers started to appear in the early 1950s.
- We can’t give exact date of release because initially an experimentation and implementation were done independently by several group.
- Most of them working on translation of arithmetic formulas into machine code.
- At that time compilers were considered as difficulty programs to write.
- The first compiler was written by Grace Hopper, in 1952, for the A-0 programming language.
- COBOL was an early language to be compiled on multiple architectures, in 1960.
- The first complete compiler is for Fortran.
- And it took 18 staff-year to implement by Backus lead by IBM in 1957.

- Compiler construction and compiler optimization are taught at universities and schools as part of the computer science curriculum.
- Such courses are usually supplemented with the implementation of a compiler for an educational programming language.

Q.3 Give brief not on translators, Compilers, interpreters and byte-code.

Ans.

⇨ Translator:

- it is most general term for a software code converting tool is “translator.”
- A translator, in software programming terms, is a generic term that could refer to a compiler, assembler or interpreter, etc. anything that converts higher level code into another high-level code or lower-level such as assembly language or machine code.
- If you don’t know what the tool actually does other than that it accomplishes some level of code conversion to a specific target language, then you can safely call it a translator.

⇨ **Compilers:**

- It converts high-level language code to machine (object) code in one session.
- Compilers can take a while, because they have to translate high-level code to lower-level machine

language all at once and then save the executable object code to memory.

- A compiler creates machine code that runs on a processor with a processor specific Instruction Set Architecture (ISA).
- Compilers are also platform-dependent.

⇨ Compiler operates on phases and various stages can be grouped into two parts that are:

- **Analysis Phase:** this phase is also referred to as the front end in which program is divided into fundamental constituent parts.
  - ✓ In this it checks grammar, semantic and syntax of the code after which intermediate code is generated.
  - ✓ Analysis phase includes lexical analyzer, semantic analyzer and syntax analyzer.

- **Synthesis phase:** this phase is also known as the back end in which intermediate code is optimized and target code is generated.
  - ✓ Synthesis phase includes code optimizer and code generator.

⇨ **Interpreter:**

- An interpreter is a computer program that is used to directly execute program instructions written using one of the many high-level programming languages.
- It transforms the high-level program into an intermediate language for executes or it could parse the high-level source code and then performs the commands directly.

- In this conversion is done line by line or statement by statement.
- In simple word an interpreter transforms or interprets a high-level programming code into code that can be understood by the machine (machine code) or into an intermediate language that can be easily executed as well.
- In contrast, an assembler or a compiler converts a high-level source code into native (compiled) code that can be executed directly by the operating system.
- It has advantage that It can be stopped in the middle of execution to allow for either code modification or debugging.
- Since an interpreter reads and then executes code in a single process, it very useful for scripting and other small programs. As such, it is commonly installed on Web servers, which run a lot of executable scripts.
- If an error occurs, an interpreter stops execution and reports it; whereas a compiler reads the whole program even if it encounters several errors.

⇨ **Bytecode:**

- Bytecode is computer object code that is processed by a program, usually referred to as a virtual machine rather than by the "real" computer machine or processor.
- The virtual machine converts each generalized machine instruction into a specific machine instruction or instructions that this computer's processor will understand.
- Bytecode is the result of compiling source code written in a language that supports this approach.

- Using a language that comes with a virtual machine for each platform, your source language statements need to be compiled only once and will then run on any platform.

Q.4 Difference between: Compiler and interpreter

Ans.

BASIS FOR COMPARISON	COMPILER	INTERPRETER
Input	It takes an entire program at a time.	It takes a single line of code or instruction at a time.
Output	It generates intermediate object code.	It does not produce any intermediate object code.
Working mechanism	The compilation is done before execution.	Compilation and execution take place simultaneously.
Speed	Comparatively faster	Slower
Memory	Memory requirement is more due to the creation of object code.	It requires less memory as it does not create intermediate object code.
Errors	Display all errors after compilation, all at the same time.	Displays error of each line one by one.
Error detection	Difficult	Easier comparatively
Pertaining Programming languages	C, C++, C#, Scala, typescript uses compiler.	Java, PHP, Perl, Python, Ruby uses an interpreter.

Q.5 Importance of compiler design techniques

Ans.

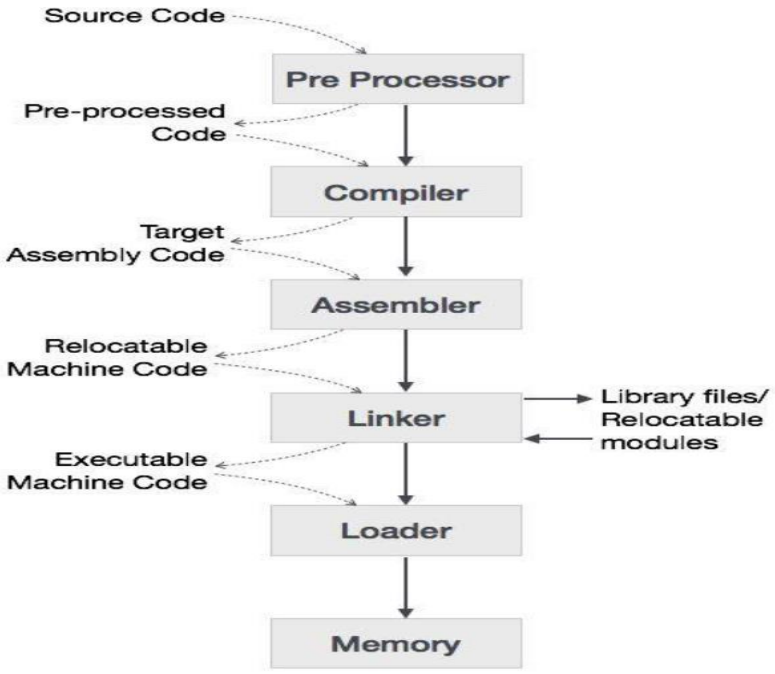
- A compiler translates the code written in one language to some other language without changing the meaning of the program.
- It is also expected that a compiler should make the target code efficient and optimized in terms of time and space.
- Compiler design principles provide an in-depth view of translation and optimization process.
- Compiler design covers basic translation mechanism and error detection & recovery.
- It includes lexical, syntax and semantic analysis as front end and code generation and optimization as back-end.

Q.6 Cousins of compilers and explain how compiler operate.

Or

Q.6 Skeleton of compilers.

Ans.



- As we can see in this diagram that input to a compiler may be produced by one or more pre-processors and further processing of the compiler's output may be needed before running machine code is obtained.
- With help of cousins of compiler typically compiler operates.
- In addition to a compiler, several other programs may be required to create an executable target program.
- A source program may be divided into modules stored in separate files.
- The task of collecting the source program is sometimes entrusted to a distinct program, called preprocessor.
- The preprocessor may also expand shorthand's, called macros, into source language statements.
- The target program creates by the compiler may require further processing before it can be run.

- Cousins of compiler are given bellow:

⇨ Preprocessors:
- It is tool which produces input to compilers.
- generally considered as a part of compiler.
- It deals with macro-processing, augmentation, file inclusion, language extension, etc.

- They perform the following functions:

1. Macro processing:
   - A preprocessor may allow a user to define macros that are shorthand's for longer constructs.

- And it deals with two kind of statement macro definition and macro use.
- Definitions are normally indicated by some unique character or keywords like define or macro.
- A macro name is any string of letters precede by backslash.

2. File inclusion:
   - A preprocessor may include header files into the program text.
   - E.g. #include <stdio.h> in C.
3. “Rational” preprocessors:
   - These processors augment older languages with more modern flow-of-control and data-structuring facilities.
   - E.g. it might provide the user with built-in macros for constructs like while-statements or if-statements where none exist in the programming language itself.
4. Language extensions:
   - These processors attempt to add capabilities to the language by what amounts to built-in macros.
   - E.g. In Equel(database query lang.) embedded with C, statements beginning with ## are taken by the preprocessor to be database-access statements.

⇨ Assemblers:

- Assembly code is a mnemonic version of machine code, in which names are used instead of binary codes for operations and names are also given to memory addresses.
- And that mnemonic is passed directly to the loader/linker.
- E.G:

```
MOV A,R1;
ADD #2,R1;
MOV R1,B;
```

⇨ Two-Pass Assembly

- The simplest form of assembler makes two passes over the input, where a pass consists of reading an input file once.
- In the first pass, all the identifiers that denote storage locations are found and stored in a table which called symbol table.
- Identifiers are assigned storage locations as they are encountered for the first time.

- In the second pass, the assembler scans the input again and translates each operation code into the sequence of bits representing that operation in machine language.
- It also translates each identifier representing a location into the address given for that identifier in the symbol table.
- The output of the second pass assembly is usually relocatable machine code which can be loaded starting at any location L in memory, i.e. if L is added to all addresses in the code, then all references will be correct.

⇨ Loaders and Link-Editors

- Usually, a program called a loader performs the two functions of loading and link-editing.
- The process of loading consists of taking relocatable machine code, altering the relocatable addresses and placing the altered instructions and data in memory at proper locations.
- The link-editor allows us to make a single program from several files of relocatable machine code.
- These files may have been the result of several different compilations and one or more may be library files of routines provided by the system and available to any program that need them.
- If the files are to be used together, there may be some external references, in which the code of one file refers to a location in another file.
- This reference may be to a data location defined in one file and used in another, or it may be to the entry point of a procedure that appears in the code for one file and is called from another file.
- The relocatable machine code file must retain the information in the symbol table for each data location or instruction label that is referred to externally.

Q.7 Model of compilation and give brief note on each phase of compilation

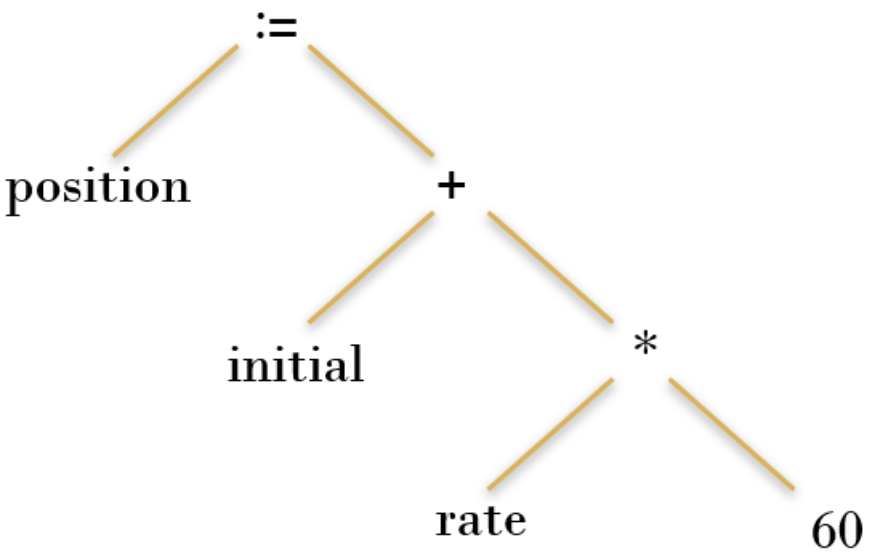
Ans.

➢ The Analysis-Synthesis Model of Compilation given below and There are two parts to compilation:

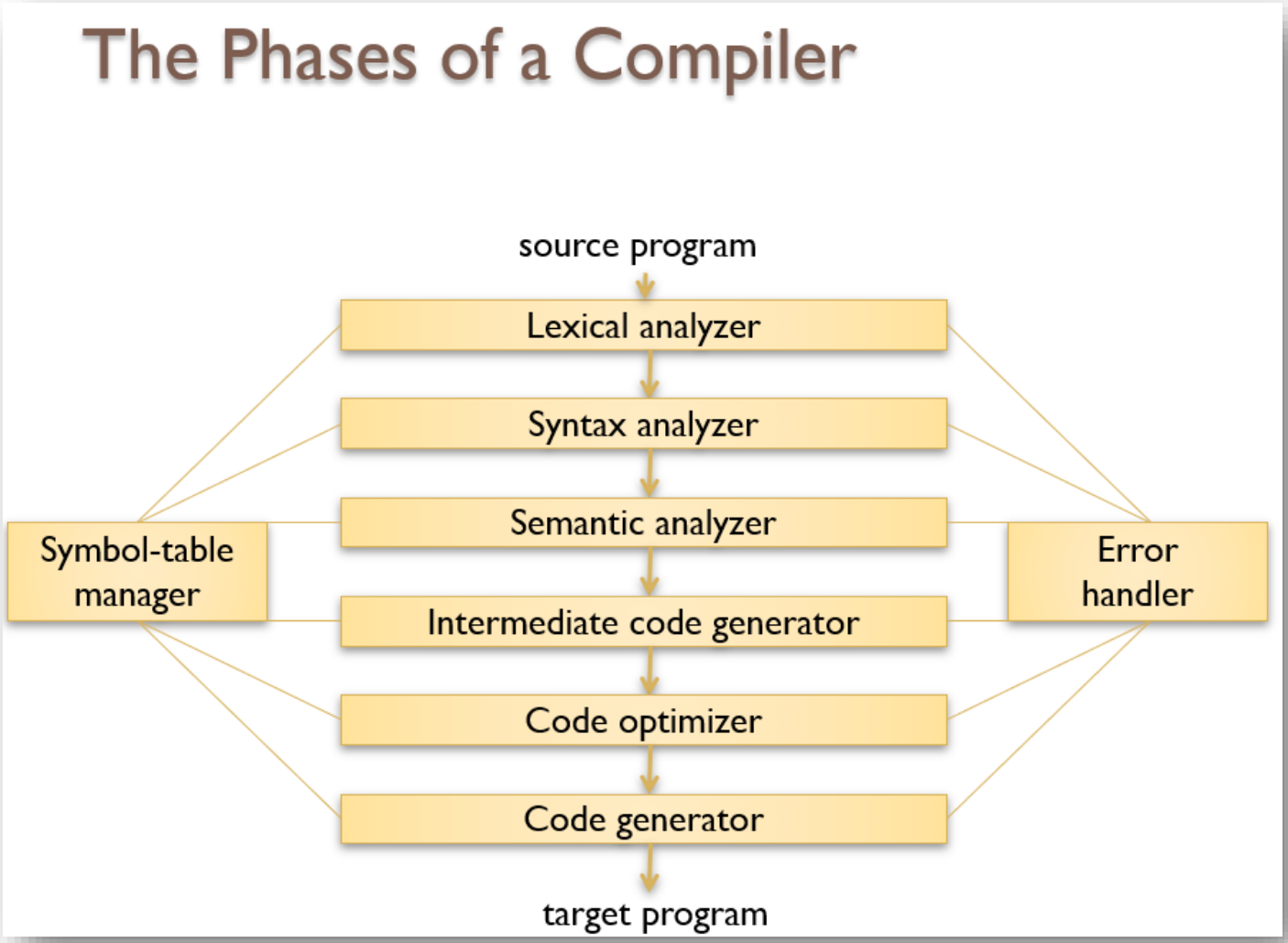
1. Analysis: The analysis part breaks up the source program into constituent pieces and creates an intermediate representation of the source program.
2. Synthesis: The synthesis part constructs the desired target program from the intermediate representation.

- During analysis, the operations implied by the source program are determined and recorded in a hierarchical structure called a tree.
- And that special kind of tree called syntax tree
- It represents each node of tree represents an operation and the children of a node represent the arguments of the operation.
  - The following is the syntax tree for the statement

position := initial + rate * 60



- Synthesis requires the most specialized techniques.



- A compiler operates in each phase and transforms the source program from one representation to another.
- A typical decomposition of a compiler is shown in above figure.

⇨ Analysis consists of three phases:

1. Linear analysis:

- In a compiler, linear analysis is called lexical analysis or scanning.
- The first phase of scanner works as a text scanner.

- This phase scans the source code as a stream of characters and converts/group them into a stream of token which also called lexemes.
- In other word the character sequence forming a token is called the lexeme for token.
- This phase not only create tokens but also add (augmented) to a “lexical value”.
- the source program is read from left-to-right and grouped into tokens that are sequences of characters having a collective meaning.
- Lexical analyzer represents these lexemes in the form of tokens as:

  <token-name, attribute-value>
- E.g. in lexical analysis the characters in the assignment statement

  position := initial + rate * 60

  would be grouped into the following tokens:

1. The identifier position
2. The assignment symbol :=
3. The identifier initial
4. The plus sign
5. The identifier rate
6. The multiplication sign
7. The number 60

- The blank separating the characters of these tokens would normally be eliminated during lexical analysis.

2. Hierarchical analysis:

- Hierarchical analysis is called parsing or syntax analysis.
- In this characters or tokens are grouped hierarchical into nested collections with collective meaning.
- Or in other word It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree).
- In this phase, tokens and arrangements are checked against the source code grammar i.e., the parser checks if the expression made by the tokens is syntactically correct.
- And that are used by the compiler to synthesize output.
- Usually the grammatical phrases of the source program are represented by a parse tree like one as below.
- In the expression (initial + rate * 60), the phrase rate * 60 is a logical unit because the usual conventions of arithmetic expressions tell us that multiplication is performed before addition.
- Because the expression initial + rate is followed by a *, it is not grouped into a single phrase by itself.
- In short, this phase work is checking the grammar of code and make parser tree using grammar of code.

- The hierarchical structure of a program is usually expressed by recursive rules.
- E.g. the following rules is the part of the definition of expressions:

    1. Any *identifier* is an expression.
    2. Any *number* is an expression.
    3. If expression_1 and expression_2 are expressions, then so are

        $\text{expression}_1 + \text{expression}_2$

        $\text{expression}_1 * \text{expression}_2$

        (expression_1)

- Rules (1) and (2) are non-recursive basis rules, while (3) defines expressions in terms of operators applied to other expressions.
- Thus by rule (2), 60 is an expression, while by rule (3), first it can be inferred that rate * 60 is an expression and finally that initial + rate * 60 is an expression.
- Similarly many languages define statements recursively by rules such as:

1. If *identifier1* is an identifier, and *expression2* is an expression then

    identifier1 := expression 2

    is a statement.
2. If *expression1* is an expression and *statement2* is a statement, then

while (*expression1*) do *statement2*

if (*expression 1*) then *statement2*

are statements.

3. Semantic analysis:

- Semantic analysis checks whether the parse tree constructed follows the rules of language.
- An important component of the semantic analysis is types checking.
- For example, assignment of values is between compatible data types, and adding string to an integer.
- Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not, etc.
- In this phase the compiler checks that each operator has operands that are permitted by the source language specification.
- The semantic analyzer produces an annotated syntax tree as an output.
- It checks that the components of a program fit together meaningfully.
- The semantic analysis phase checks the source program for semantic errors and gathers type information for the subsequent code-generation phase.

- It uses the hierarchical structure determined by the syntax-analysis phase to identify the operators and operands of expressions and statements.
- In short Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Semantics help interpret symbols, their types, and their relations with each other. Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not

4. Intermediate code generation:

- The intermediate representation should have two properties:
    1. It should be easy to produce
    2. And easy to translate into the target program.
- The intermediate representation can have a variety of forms means it can be written in various way.
- The intermediate form called “three-address code”.
- And it is used for is represent code like the assembly language for a machine in which every memory location can act as a register.
- Three-address code consists of a sequence of instructions and each of has at most three operands.
- So, the statement would be as follows using the three-address code:

    temp1 := inttoreal(60)

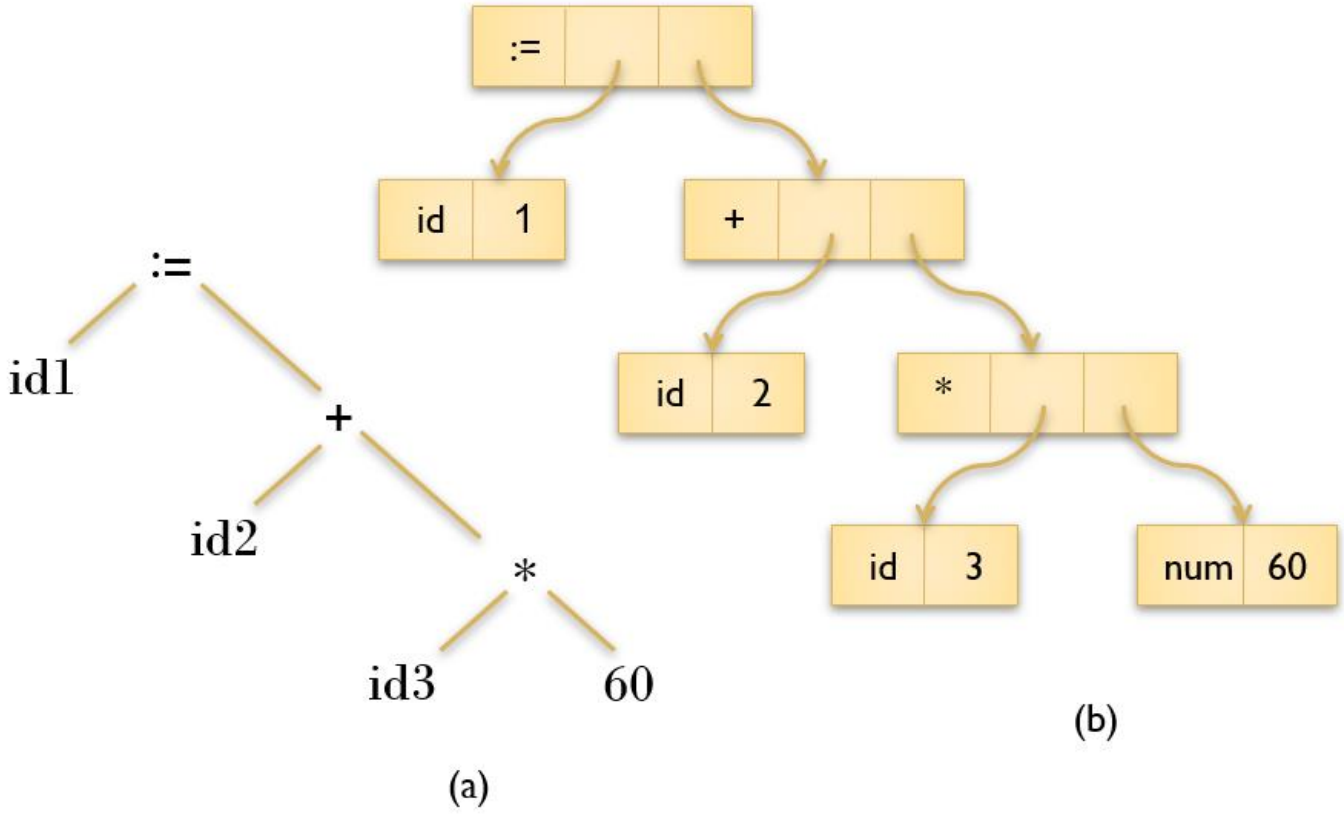
temp2 := id3 * temp1

temp3 := id2 + temp2

id1 := temp3

- This intermediate form has the following properties.

1. Each three-address instruction has at most one operator in addition to the assignment.
   - Thus, when generating these instructions, the compile has to decide on the order in which operations are to be done.
2. The compiler must generate a temporary name to hold the value computed by each instruction.
3. Some three-address instructions have fewer than three operands.



- After semantic analysis, the compiler generates an intermediate code of the source code for the target machine.
- It represents a program for some abstract machine.
- It is in between the high-level language and the machine language.
- This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

5. Code Optimizer:

- The next phase does code optimization of the intermediate code.
- Optimization is a program transformation technique.
- which tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed.
- By removing unnecessary code lines and arranges the sequence of statements in order to speed up the program execution.
- It improves the intermediate code, so that faster-running machine code will result.
- In optimization, high-level general programs are replaced by very efficient low-level programming codes.
- The example statement would be as follows after performing code optimization.

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```

- There is great variation in the amount of code optimization different compilers perform.
- In those that do the most, called “optimizing compilers”, a significant fraction of the time of the compiler is spent on this phase.
- A code optimizing process must follow the three rules given below:
  1. The output code must not, in any way, change the meaning of the program.
  2. Optimization should increase the speed of the program and if possible, the program should demand les number of resources.
  3. Optimization should itself be fast and should not delay the overall compiling process.
- Optimization can be categorized broadly into two types:
  1. machine independent.
  2. machine dependent.

6. Code Generator:

- This is the final phase of the compiler.
- In this phase, it takes the optimized code.
- Above phase generated code and in this phase maps code to the target machine language.
- Generally, it translates the intermediate code into a sequence of re-locatable machine code.
- And it performs same the task as the intermediate code would do.
- Memory locations are selected for each of the variables used by the program.

- A crucial aspect is the assignment of variables to registers.
- Above phase can be seen as a part of code generation phase itself.
- It should follow minimum properties given below:
  1. It should carry the exact meaning of the source code.
  2. It should be efficient in terms of CPU usage and memory management.

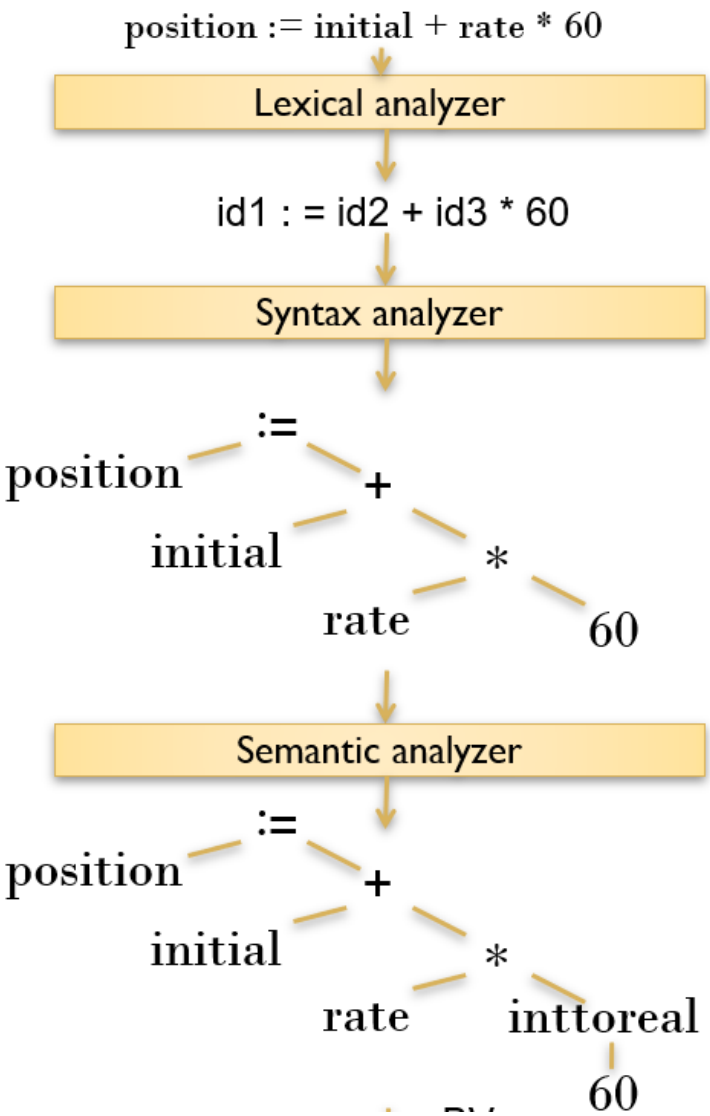
⇨ Symbol-Table Management:

- It is a data-structure containing a record with fields for the attributes of the identifier.
- And it is maintained throughout all the phases of a compiler.
- All the identifiers' names along with their types are stored here.
- The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it.
- The symbol table is also used for scope management
- And this is essential function of a compiler.
- Symbol table work is to record the identifiers used in the source program and also collect information about various attributes of each identifier.
- These attributes may provide information about the storage allocated for an identifier, 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, if any.

- When an identifier in the source program is detected by the lexical analyzer, the identifier is entered into the symbol table.
- However, the attributes of an identifier cannot be determined during lexical analysis.
- The remaining phases enter information about identifiers into the symbol table and then use this information in various ways.

⇨ Error Detection and Reporting:

- Each phase can encounter errors.
- After detecting error, a phase must somehow deal with that error, so that compilation can proceed, allowing further errors in the source program to be detected.
- A compiler that stops when it finds the first error.
- The syntax and semantic analysis phases usually handle a large fraction of the errors detectable by the compiler.
- The lexical phase may detect errors where the characters remaining in the input do not form any token of the language.
- Errors where the token stream violates the structure rules (syntax) of the language are determined by the syntax analysis phase.
- During semantic analysis the compiler tries to detect constructs that have the right syntactic structure but no meaning to the operation involved.
- E.g. adding two identifiers, one of which is the name of an array and the other the name of a procedure has no meaning.


position := initial + rate * 60
Lexical analyzer
id1 : = id2 + id3 * 60
Syntax analyzer
:=
position
+
initial
*
rate
60
Semantic analyzer
:=
position
+
initial
*
rate
inttoreal
60

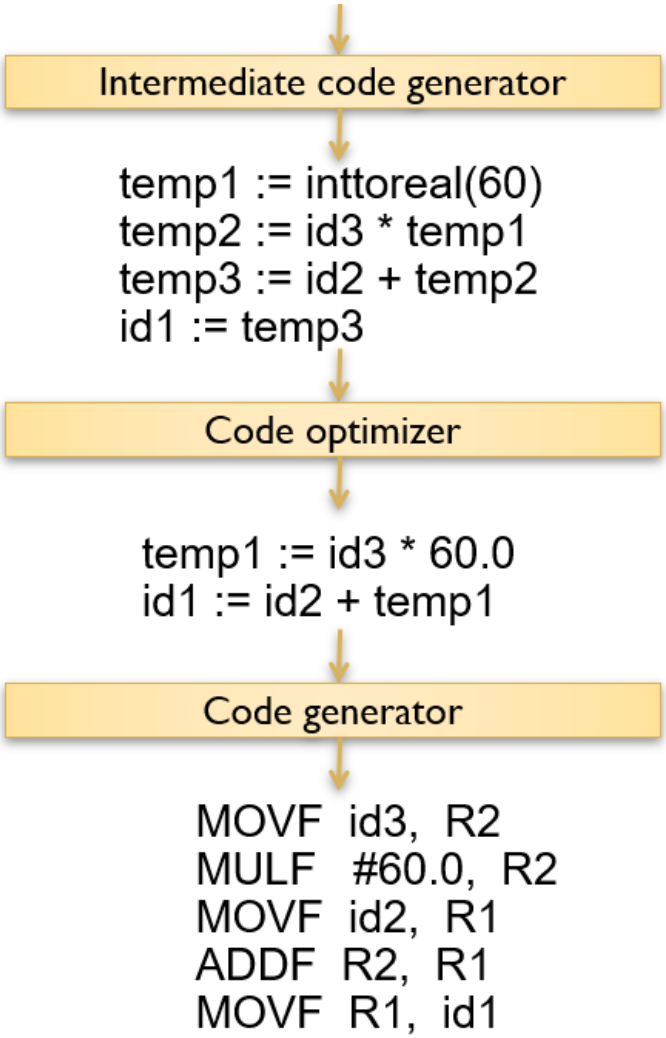


Fig : Translation of a statement through different phases of compiler

Q.8 Explain Front and Back Ends of compiler and passes.

Ans.

- Front and Back Ends

- The front end consists of the following: phases or parts of phases that depend primarily on the source language and are largely independent of the target machine.
- Normally it includes lexical and syntactic analysis.
- The creation of the symbol table, semantic analysis and the generation of intermediate code.

- A certain amount of code optimization can be done by the front end as well.
- The front end also includes the error handling that goes along with each of these phases.

- Back end:
  - The back end includes those portions of the compiler that depend on the target machine.
  - Generally, these portions do not depend on the source language, just the intermediate language.
  - In the back end, code optimization phase and code generation phase.
  - In this phase also necessary error handling and symbol-table operations are done.

- Passes
  - Several phases of compilation are usually implemented in a single pass consisting of reading an input file and writing an output file.
  - The activity of these phases is to be interleaved or done during the pass.
  - E.g. lexical analysis, semantic analysis, syntax analysis and intermediate code generation might be grouped into one pass.
  - If so, the token stream after lexical analysis may be translated directly into intermediate code.
  - As the grammatical structure is discovered, the parser calls the intermediate code generator to perform semantic analysis and generate a portion of the code.

- <u>Reducing the Number of Passes</u>

- It is desirable to have relatively few passes, since it takes time to read and write intermediate files.
- On the other hand, if several phases are grouped into one pass, it is required to keep the entire program in memory, because one phase may need information in a different order than a previous phase produces it.
- The internal form of the program may be considerably larger than either the source program or the target program, so this space may not be a trivial matter.
- For some phases, grouping into one pass presents few problems.
- E.g. the interface between the lexical and syntactic analyzers can often be limited to a single token.
- On the other hand, it is often very hard to perform code generation until the intermediate representation has been completely generated.
- The target code cannot be generated for a construct if we do not know the types of variables involved in that construct.
- Similarly, most languages allow goto's that jump forward in the code.
- The target address for such a jump cannot be determined.
- In some cases, it is possible to leave a blank slot for missing information, and fill in the slot when the information becomes available.
- In particular, intermediate and target code generation can often be merged into one pass using a technique called "backpatching".

Q.8 Explain Front and Back Ends of compiler and passes.

Ans.

- Difference between lexical and syntactic analysis:

- One factor in determining the division is whether a source language construct is inherently recursive or not.
- Lexical construct does not require recursion, while syntactic construct do require recursion.
- Context free grammars are a formalization of recursive rules that can be used to guide syntactic analysis.
- E.g. recursion is not required to recognize identifiers, which are typically strings of letters and digits beginning with a letter.
- Identifiers are normally recognized by a simple scan of the input stream, waiting until a character that was neither a letter nor a digit was found, and then grouping all the letters and digits found up to that point into an identifier token.
- The characters so grouped are recorded in a table, called symbol table, and removed from the input so that processing of the next token can begin.
- On the other hand, this kind of linear scan is not powerful enough to analyze expressions or statements.
- E.g. without putting some kind of hierarchical or nesting structure on the input, we cannot properly match parentheses in expressions, or begin and end in statements.

- The above figure is a more common internal representation of the syntactic structure in fig 1.4.

- Syntax tree:
  - A syntax tree is a compressed representation of the parse tree in which the operators appear as the interior nodes, and the operands of an operator are the children of the node for that operator.

Q. Compiler construction tools.

Ans.

- Shortly after the first compilers were written after that software/tools appeared to help with the compiler-writing process.
- These systems have often referred to as:
  - Compiler-compilers
  - Compiler-generators
  - Translator-writing systems
  - Some general tools have been created for the automatic design of specific compiler components.

- These tools use specialized languages for specifying and implementing the components and many used algorithms that are quite sophisticated.
- The most successful tools are those that hide the details of the generation algorithm
- And produce components that can be easily integrated into the remainder of s compiler.

⇨ Some compiler-construction tools are as follows:

1. Scanner generators:

- These automatically generates lexical analyzers, from regular expressions.
- The basic organization of the resulting lexical analyzer is in effect a finite automaton.

2. Parser generators:

- It produces syntax analyzers which is take input based on a context-free grammar.
- In early compilers, syntax analysis consumed not only a large fraction of the running time of a compiler.
- But a large fraction of the intellectual effort of writing a compiler.
- This phase is now considered one of the easiest to implement.
- Many of the little languages like PIC and EQN are developed in a few days using the parser generator.

3. Syntax-directed translation engines:

- These produce collection of routines that walk the parse tree generating intermediate code.
- The basic idea is that one or more “translations” are associated with each node of the parse tree and each translation is defined in terms of translations at its neighbor nodes in the tree.

4. Automatic Code generators:

- Such a tool takes a collection of rules that define the translation of each operation of the intermediate

language into the machine language for the target machine.

- The rules must include sufficient detail that we can handle the different possible access methods for data.
- The basic technique is “template matching”.
- The intermediate code statements are replaced by “templates” that represent sequences of variables match from template to template.

5. Data-flow engines:

- Much of the information needed to perform good code optimization involves “data-flow analysis” that gathers the information about how values are transmitted from one part of a program to each other part.
- Different parts of this nature can be performed by essentially the same routine, with the user supplying details of the relationship between intermediate code statements and the information being gathered.

- The following are some software tools that perform some kind of analysis.

1. Structure editors:

- A structure editor takes as input a sequence of commands to build a source program.
- It not only performs the text-creation and modification functions of an ordinary text editor, but it also analyses the program text, putting an appropriate hierarchical structure on the source program.

- So, the structure editor can perform additional tasks that are useful in the preparation of program.
- The output of such an editor is often similar to the output of the analysis of a compiler.

2. Pretty printers

- It analyzes a program and printed it in such a way that the structure of the program becomes clearly visible.
- E.g. comments may appear in a special form, and statements may appear with an amount of indentation.

3. Static checkers

- It reads a program, analyse it and attempts to discover potential bugs without running the program.
- It may detect that parts of the source program can never be executed, or that a certain variable might be used before being defined.
- It can catch logical error such as trying to use real number as a pointer.

4. Interpreters

- Instead of producing a target program as a translation, an interpreter performs the operations implied by the source program.
- Interpreters are frequently used to execute command languages, since each operator executed in a command language is usually an invocation of a complex routine such as an editor or compiler.

Q. Give unrelated places where compiler technology is regularly used.

Ans.

⇨ There are certain unrelated places where compiler technology is regularly used.

1. Text formatters: A text formatter takes input that is a stream of character, most of which is text to be typeset, but some of which includes commands to indicate paragraphs, figures or mathematical structures like subscripts and superscripts.

2. Silicon compilers: A silicon compiler has a source language that is similar or identical to a conventional programming language.
   - However, the variables of the language represent, not locations in memory, but logical signals (0 or 1) or groups of signals in a switching circuit.
   - The output is a circuit design in an appropriate language.

3. Query interpreters: A query interpreter translate a predicate containing relational and Boolean operators into commands to search a database for records satisfying that predicate.