1. Introduction

Q.4.6.1.1 What do you mean by a language translator?
Answer: It is a software which bridges an execution gap. The input program of a language translator is a source program and the output program is the target program. Different types of language translators are mentioned here.
◦ *Assembler* is a language translator whose source language is an assembly language.
◦ *Compiler* is a language translator whose source language is a high level language.
◦ *Detranslator* is a language translator which converts machine language to an assembly level language.

Q.4.6.1.2 How does a compiler differ from a translator?
Answer: A language translator for a programming language (source language) L bridges an execution gap between L and the machine language or, assembly language, for a given computer. In particular, a language translator is called a compiler, if L is a high level programming language. In other words, if the source language is a high-level language and the object language is a low-level language such as an assembly language or machine language, then such a translator is called a compiler.

Q.4.6.1.3 State characteristics of an interpreter.
Answer: Some important characteristics of an interpreter are given below:
$*$ It bridges execution gap between programming language domain and execution domain without generating a target program.
$*$ It interprets one statement at a time, find its meaning and then performs actions that implement the meaning.
$*$ There is a substantial overhead during interpretation, which makes interpretation slow.

Q.4.6.1.4 Explain the purposes of the following software tools: cross-compiler, static checker and pretty printer.
Answer: A compiler that runs on one machine and produces object code for another machine is called a *cross compiler*.
A *static checker* reads a program, analyses it, and attempts to dis-

cover potential bugs without running the program. For example, a static checker may detect that parts of the source program can never be executed or that a certain variable might be used before being defined.
A *pretty printer* analyzes a program and prints it in such a way that the structure of the program becomes clearly visible. For example, comments may appear in a special font, and statements may appear with an amount of indentation proportional to the depth of their nesting in the hierarchical organisation of the statement.

Q.4.6.1.5 Explain main phases of compilation process.
Answer: Compilation of a program can be performed using two main phases: analysis and synthesis.
During analysis phase, the source program is broken down into pieces, and an intermediate representation is created.
In the synthesis phase, the desired target program is created from the intermediate representation.

Q.4.6.1.6 What is dynamic binding? Compare it with static binding.
Answer: Dynamic binding, also called late binding, is a computer programming mechanism in which the method being called upon by an object, or the function being called with arguments, is looked up by name at runtime. In other words, a name is associated with a particular operation or object at runtime, rather than during compilation.
The primary advantage of using late binding in component object model (COM) programming is that it does not require the compiler to reference the libraries that contain the object at compile time. This makes the compilation process more resistant to version conflicts, in which the class's virtual method table may be accidentally modified.
With early binding, or static binding, in an object-oriented language, the compilation phase fixes all types of variables and expressions. This is usually stored in the compiled program as an offset in a virtual method table.

Q.4.6.1.7 What is associativity?
Answer: An operator in an expression is left associative if operands are grouped from left to right. For example, if operator + is left associative then $a + b + c$ can be interpreted as $((a + b) + c)$. In the same way, right associativity can be defined.

Q.4.6.1.8 Why do we need a precedence level of an operator?

Answer: In an unparenthesized expression, or sub-expression, we need to tell the evaluator which operands are allowed to group their operands first. For example, $*$ is given higher precedence than $+$, then the expression $a + b * c$ is equivalent to $(a + (b * c))$.

Q.4.6.1.9 What is returned to the parser by the lexical analyzer, for each token?
Answer: For each token, the lexical analyser returns token type (code) and an index value of symbol table. At this index position, the details of the symbol is stored. In case of token like if, there is no value returned. In case of token like $<$, value 1 may be returned.

Q.4.6.1.10 Find the output of the following program using call-by-reference.

```
procedure $P$ $(X, Y, Z)$
begin
        $Y \leftarrow Y + 1$;
        $Z \leftarrow Z + X$;
end
begin
        $A \leftarrow 2$; $B \leftarrow 3$;
        $P$ $(A + B, A, A)$;
        $print$ $A$;
end
```

Answer: The above program has a procedure $P()$. The output of the above program is explained using Fig. 1.1. The output, i.e., the value of A, is 8.

Q.4.6.1.11 What are the purposes of compiler?
Answer: There are two purposes of a general compiler.
i. It transfers a program written in a language into an equivalent program in another language.
ii. It reports errors present in the source program.

Q.4.6.1.12 Generate intermediate code from the parse tree given in Fig. 1.2.
Answer: Corresponding intermediate code for the parse tree given in Fig. 1.2 is given below:
$temp1 = 100$

2. Basic Compiling Techniques

Q.4.6.2.1 Explain the terms: syntax, semantics.
Answer: The *syntax* of a programming language refers to the structure of programs without considering their meaning. It emphasizes the structure, layout of a program with their appearance. It involves a collection of rules which validates the sequence of symbols and instructions used in a program. The pragmatic and computation model figure these syntactic components of a programming language. Some tools evolved for the specification of the syntax of the programming languages are regular, context-free and attribute grammars.
The term *semantics* in a programming language is used to figure out the relationship between the syntax and the model of computation. It emphasizes the interpretation of a program so that the programmer could understand it in an easy way and / or predict the outcome of program execution. An approach known as syntax-directed semantics is used to map syntactical constructs to the computational model with the help of a function. The programming language semantics can be described by various techniques such as algebraic semantics, axiomatic semantics, operational semantics, denotational semantics, and translation semantics. See Table 2.2 for finding the differences between syntax and semantics as used in programming languages.

Table 2.2: Comparison between syntax and semantics

1. [Defitition] Syntax refers to grammar of writing expression / sentence. Semantics refers to meaning associated with it.
2. [Error handling] Syntax errors are handled during compile time. Semantics errors are encountered during runtime.
3. [Relation] Interpretation of a syntactic expression / structure has a distintive meaning. Semantics of an expression / structure is associated with its syntax.

Q.4.6.2.2 Explain the notion of syntax directed translation scheme.
Answer: Syntax-directed translation (SDT) refers to a method of compiler implementation, where the source language translation is completely driven by the parser.
A common method of syntax-directed translation is translating a string into a sequence of actions by attaching one such action to each rule of

a grammar. Thus, parsing a string of the grammar produces a sequence of rule applications. SDT provides a simple way to attach semantics to any such syntax.
Syntax-directed translation fundamentally works by adding actions to the productions in a context-free grammar, resulting in a syntax-directed definition (SDD). Actions are steps or procedures that will be carried out when that production is used in a derivation. A grammar specification embedded with actions to be performed is called a syntax-directed translation scheme.
Each symbol in the grammar can have an attribute, which is a value that is to be associated with the symbol. Common attributes could include a variable type, the value of an expression, etc. Given a symbol X, with an attribute t, that attribute is referred to as $X.t$. Thus, given actions and attributes, the grammar can be used for translating strings from its language by applying the actions and carrying information through each symbol's attribute.

Q.4.6.2.3 Fill in the blanks.
Method of syntax-directed translation combines both —— and ——.
Answer: syntax analysis, intermediate code generation

Q.4.6.2.4 Define context-free grammar and context-free language.
Answer: A context-free grammar G is defined by the 4-tuple

$$G = (V, \Sigma, R, S)$$

where, V is a finite set; each element $v \in V$ is called a nonterminal character or a variable. Σ is a finite set of terminals, disjoint from V, which makes up the actual content of the sentence. The set of terminals is the alphabet of the language defined by the grammar G. R is a finite relation in $V \times (V \cup \Sigma)^*$, where the asterisk represents the Kleene star operation. The members of R are called the productions of the grammar. S is the start symbol, used to represent the whole sentence (or program). S is an element of V.
The language of a grammar $G = (V, \Sigma, R, S)$ is the set

$$L(G) = \{w \in \Sigma^* : S \overset{*}{\Rightarrow} w\}$$

of all terminal-symbol strings derivable from the start symbol.
A language L is said to be a context-free language (CFL), if there exists a CFG G, such that $L = L(G)$.

Q.4.6.2.5 What is ambiguous grammar? Give an example.
Answer: An ambiguous grammar is a context-free grammar for which there exists a string that can have more than one leftmost derivation, also known as a parse tree. Consider the following grammar, as given by its production rules:
$S \rightarrow S + S \mid S - S \mid a$
The grammar is ambiguous, since there are two leftmost derivation for string $a + a - a$, as given below.

$$S \Rightarrow S + S \Rightarrow a + S \Rightarrow a + S - S \Rightarrow a + a - S \Rightarrow a + a - a \quad (2.1)$$
$$S \Rightarrow S - S \Rightarrow S + S - S \Rightarrow a + S - S \Rightarrow a + a - S \Rightarrow a + a - a \quad (2.2)$$

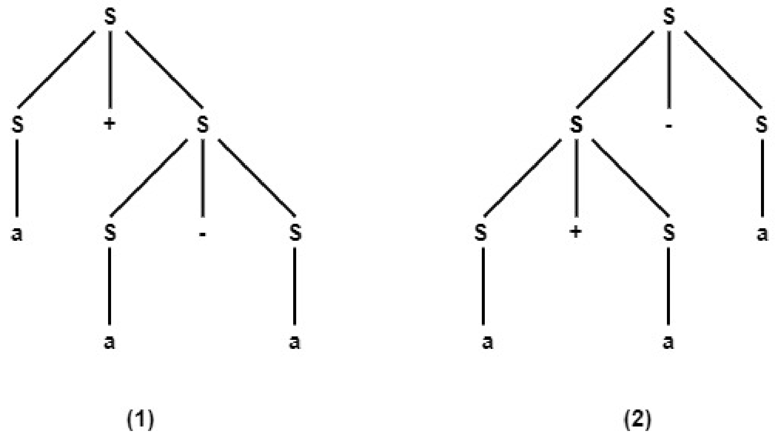

Figure 2.1: Parse trees for derivations (2.1) and (2.2)

Derivations (2.1) and (2.2) generate parse trees Fig. 2.1 (1) and Fig. 2.1 (2) respectively.

Q.4.6.2.6 Consider the following context-free grammar.
$S \rightarrow iCtS$
$S \rightarrow iCtSeS$
$S \rightarrow a$
$S \rightarrow b$
Construct a leftmost derivation for the sentence $(w) = ibtibtaea$. Also construct the parse tree at each phase of left-most derivation process.
Answer: Parse tree for a leftmost derivation of the sentence $(w) = ibtibtaea$ is given in Fig. 2.2.
$S \Rightarrow iCtS \Rightarrow ibtS \Rightarrow ibtiCtSeS \Rightarrow ibtibtSeS \Rightarrow ibtibtaea \Rightarrow ibtibtaeS$
Construction of parse tree is shown below.

3. Lexical Analysis

Q.4.6.3.1 Discuss techniques of writing a computer program for a lexical analyser.
Answer: We discuss here two approaches for the task. A simple technique is to draw a deterministic finite automaton (DFA) that recognizes a given language. Then the diagram could be translated into a computer program that recognizes tokens in the given language. Another approach is to design patterns using pattern-directed programming language Lex. Then the compiler of Lex generates a finite-automaton from the given design patterns that recognizes the regular expression.

Q.4.6.3.2 Discuss important tasks performed by a lexical analyzer.
Answer: Lexical analyzer plays an important role in translating a source program into object program. In this regard, some important tasks performed are given below.
∗ Reading input characters in the source program
∗ Identifying tokens from the source program and insert into the symbol table
∗ Removing white spaces and comments from the source program
∗ Generating appropriate error messages, whenever they are required
∗ Expanding the macros, if it is found in the source program
Also, lexical analyzers can be applied to areas such as query language and information retrieval systems.

Q.4.6.3.3 Find the appropriate one.
If the lexical analyzer finds a token invalid then it (i) generates an exception, (ii) generates an error, (iii) generates an warning, (iv) continues reading the program.
Answer: (ii)

Q.4.6.3.4 Give examples of tokens that are accepted by most programming languages.
Answer: Most programming languages accept the following constructs as tokens: operators, keywords, identifier, constant, string literal and punction symbol.

Q.4.6.3.5 Distinguish between keyword and reserved word.

Answer: In a computer programming language, a reserved word, i.e., a reserved identifier, is a word that cannot be used as an identifier, such as the name of a variable, function, or label.
A keyword has a special meaning in a particular context. In some cases, names in a standard library are not built into the language as reserved words or keywords.
In general, reserved words and keywords need not coincide. In most modern programming languages, keywords are a subset of reserved words. This makes parsing easier, since keywords cannot be confused with identifiers. In some languages, like C or Python, reserved words and keywords coincide. In other languages, like Java, all keywords are reserved words, but some reserved words are not keywords - these are "reserved for future use". Again, in some older languages such as ALGOL, FORTRAN and PL/I, there are keywords but no reserved words. Hence, keywords being distinguished from identifiers by other means. Such generality makes parsing more difficult and hence look-ahead parsers are required.

Q.4.6.3.6 Give some examples of attributes of a token.
Answer: When lexical analyser receives i as an identifier, $<$ **id**, *pointer to symbol-table entry for i* $>$ is returned to parser. Here, the *pointer to symbol-table entry for i* is an attribute for token **id**.
When lexical analyser receives "=" as an assignment operator, $<=,>$ is returned to parser. In this case, there is no attribute value for token = operator.
When lexical analyser receives 23 as integer constant, $<$ **num**, $23 >$ is returned to parser. Here, integer value 23 is an attribute for token **num**.

Q.4.6.3.7 Consider the following code in C:
fi (x = f1(3)) . . .
What error (s) the lexical analyzer generates, when it encounters **fi** for the first time?
Answer: There are many possible cases:
→ At this stage, it is difficult for the lexical analyzer to decide whether it is a mispelling of keyword **if**.
→ Also, it is difficult to say whether **fi** is an undeclared function.
→ If **fi** is a valid identifier, then it returns a token for it, and later stage of compilation addresses the error.

Q.4.6.3.8 What is the difference between null and empty strings?
Answer: Many programming languages distinguish between null and

empty strings. An empty string is a string instance of zero length, whereas a null string has no value at all.
An empty string in C is represented as "". It is a character sequence of zero character. We create here the empty string in C.
$char * pStr = ""$;
Here, pointer variable $pStr$ points to empty string.
Suppose that we have a pointer to character which is set to NULL. We create such a pointer in C as follows.
$char * pStr = NULL$;
Here, pointer variable $pStr$ points to null string.
NOTE: NULL value is compatible to any type of pointer variable in C. For example, one can declare pointer variable $pInt$ to integer, and set NULL value to it.
$int * pInt = NULL$;

Q.4.6.3.9 Define the following terms: regular expression, regular language, regular set.
Answer: Let r be a regular expression. The language generated by r is a regular language, denoted by $L(r)$. Regular expression / language over the alphabet Σ is defined recursively as follows:
1. Null string, denoted by ϕ is a regular expression representing regular language $\Phi = \{\}$.
2. Empty string, denoted by ϵ is a regular expression. The regular expression ϵ represents regular language $\{\epsilon\}$, the set containing empty string.
3. Let a be a symbol of Σ. a is a regular expression that represents regular language $\{a\}$.
4. Suppose r and s be two regular expressions representing regular languages $L(r)$ and $L(s)$ respectively. Then,
(i) (r) is a regular expression, representing regular language $L(r)$.
(ii) $(r|s)$ is a regular expression, representing regular language $L(r) \cup L(s)$.
(iii) (rs) is a regular expression, representing regular language $L(r)L(s)$.
(iv) (r^*) is a regular expression, representing regular language $(L(r))^*$.
A regular language is also called a regular set.

Q.4.6.3.10 How can you avoid extra parenthesis as part of definitions given in Q.4.6.3.9?
Answer: First, we put precedence levels of the operators from highest to lowest, given as follows: Kleene star (*), concatenation, |. Then we as-