B.Tech.

FIFTH SEMESTER EXAMINATION, 2008-09

COMPILER DESIGN

Time : 3 Hours] *[Total Marks : **100***

Note : (i) Attempt **all** questions.

(ii) All questions carry **equal** marks.

Q. 1. Attempt any four of the following sections. $5 \times 4 = 20$

Q. 1. (a) How boot strapping of compiler to more than one machine is done ? Discuss.

Ans. Using a facilities offered by a language to compile itself is the essence of boot strapping. For boot straping, a compiler is characterized by three languages :

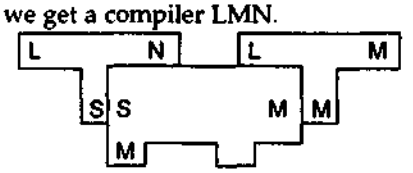
(1) The source language S that it compiles.

(2) The target language T that it generates code for.

(3) The implementation language I that it is written in.

The three language may all be quite different. A compiler may run on one machine and produce target code for another machine. Such a compiler is often called cross compiler.

Suppose for a new language L in implementation language S to generate code for machine N i.e., we create LSN. If an existing compiler for S runs on machine M and generate code for M *i.e.* SMM. If LSN is run through SMM, we get a compiler LMN.



LSN + SMM = LMN

Q. 1. (b) What do you understand by pass ? Discuss merits and demerits of multi-pass and single-pass compiler.

Ans. Pass in a procedure at which compiler, compiles the source program. Several phases of compilation are usually implemented in a single pass, consisting of reading an input file and writing an output file. Several phases of compilers are grouped and performed various passes, this is called multipasses.

Advantages & Disadvantages

(1) It is desirable to have relatively few passes, since it takes time to read and write intermediate files.

(2) If we group several phases into one pass, we may be forced to keep the entire program in memory.

(3) For some phases, grouping into one pass presents few problems. For example the interface between the lexical and syntatic analyzers can often be limited to a single token.

(4) In a single pass it is very difficult to perform code generation untill the intermediate representation has been completely generated.

(5) In a single pass all the errors are encountered after the compilation process is complete.

(6) In a multipass, errors are displayed according to the phases that are combined into different pass.

Q. 1. (c) Why do translators are needed ?

Ans. Translaters are the system software which translate are language into another language i.e. that translate source language to

www.btechsamplepapers.in

target language Where source language is either high level, assembly or low level language and target language is either high level or law level language.

Different type of translaters

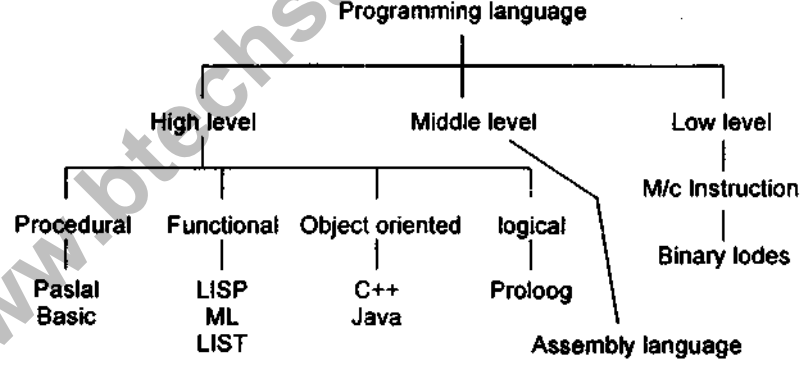
(1) Compiler : Compiler is a translator which converts source langauge to target language where source language must be high level language and target language is either high or low level language.

(2) Assembler : Assembler is a translator that translate the assembly language into machine language.

The simplest form of assembler makes two passes over the input. In first pass, all the identifiers that denote storage locations are found and stored in symbol table. In the second pass the assembler scans the input again, this time it translates each operation code into the sequence of bits representing that operation in machine language and it translates each identifiers representing a location into the address given for that identifiers in the symbol table.

Q. 1. (d) Why do translators are needed ?

Ans. Hierarchical Structure of Programming Language



Q. 1. (e) Discuss the role of different data structures in compiler design.

Ans. Role of Different Data Structure in Compiler Design

1. Stack : Stack is used in constructing the passing table and also used in scanning the input. It is used to calculate infix, prefix and postfix notation.

2. Tree : Tree is used in the second phase of compiler design, where tokens are grouped into hierarchical representation that is called parse tree. It is also used to check the ambiguity of any grammar. If for any grammar there are more than are parse tree then the grammar is ambiguous.

Otherwise the grammar is anambiguous tree is also used to check the data type of the expression. In the third phase of compiler known as syntax directed translation.

www.btechsamplepapers.in

3. Graph : Graph is used for code optimization. DAG (Directed acyclic graph) is used to optimize the intermediate code generated in the fourth phase of compiler.

Another graph known as flow graph is used to decide the flow of basic blocks. Basic block contains the group of statements of similar type. How basic blocks are executed will be decided by the flow graph.

Q. 2. Attempt any two of the following sections : $10 \times 2 = 20$

Q. 2. (a) What do you understand by preliminary scanning ? Describe the ways how lexical analyzer is grouped to make a pass.

Ans. Prelimary scanning, is usually based on a **finite state machine.** It has encoded within it information on the possible sequences of characters that can be contained within any of the tokens it handles (individual instances of these character sequences are known as **lexemes**). For instance, an *integer* token may contain nay sequence of **numerical digit** characters. In many cases, the first non-whitespace character can be used to deduce the kind of token that follows and subsequent input characters are then processed one at a time until reaching a character that is not in the set of characters acceptable for that token (this is known as the **maximal munch** rule). In some languages the lexeme creation rules are more complicated and may involve **backtracking** over previously read characters **lexical analysis** is the process of converting a sequence of characters into a sequence of token. Programs performing lexical analysis are called **lexical analyzers** or **lexers**. A lexer is often organized as separate **scanner** and **tokenizer** functions, though the boundaries may not be clearly defined.

A **token** is a categorized block of text. The block of text corresponding to the token is known as a **lexeme**. A lexical analyzer processes *lexemes* to categorize them according to function, giving them **meaning**. This assignment of meaning is known as **tokenization**. A token can look like anything; it just needs to be a useful part of the structured text.

Consider this expression in the **C programming language** :

sum = 3 + 2;

Tokenized in the following table :

lexeme	token type
sum	IDENT
=	ASSIGN_OP
3	NUMBER
+	ADD_OP
2	NUMBER
;	SEMICOLON

Tokens are frequently defined by **regular expressions**, which are understood by a lexical analyzer generator such as **lex**. The lexical analyzer (either generated automatically by a tool like lex, or hand-crafted) reads in a stream of characters, identifies the lexemes in the stream, and categorizes them into tokens. This is called "tokenizing". If the lexer finds an invalid token, it will report an error.

Q. 2. (b) Why it is difficult to simulate NFA ? Discuss a method for constructing an NFA from a regular expression.

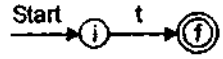
Ans. While we have a situation where we could choose a transition on E or on a seal input symbol causes ambiguity. These situations, in which the transition function is multivalued, make it hard to simulate an MFA with a computer program.

www.btechsamplepapers.in

Method for constructing an NFA from a regular expression

We first parse regular expression r into its contituent subexpression then follow the following steps :

1. For t, construct the NFA

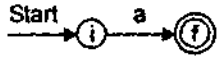


i → new start state

F → new accepting state.

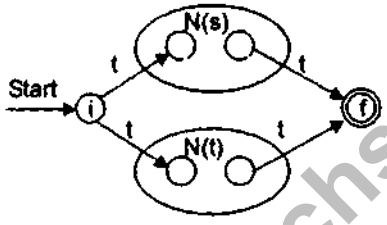
This recognizes {t}

2. For a in Σ, construct the NFA



this recognizes {a}

3. Suppose N(S) and N(t) are NFA's for regular expressions is S and t.



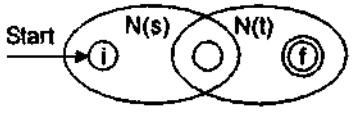
(i) For regular expression S/t construct the composite NFA N(S/t)

i → new start state

F → new accepting state

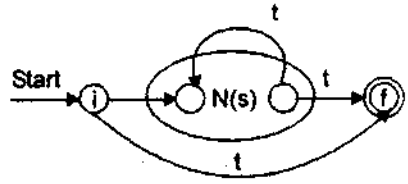
there is a transition t from i to start state of N(s) and N(t) there is a transition t from accepting state of N(s) and N(t) to F. This recognizes L(s) U L(t)

(b) For regular expression st



Starting state of N(s) is starting state of composite NFA and accepting state of N(t) is the accepting state of composite NFA.

(c) For regular experssion S*.



(d) For parentherized regular expression (S) use N(S) itself as the NFA.

Q. 2. (c) What do you understand by ambiguity in grammar ? How the grammar is made unambiguous using precedence order and associativity among arithmetic operators.

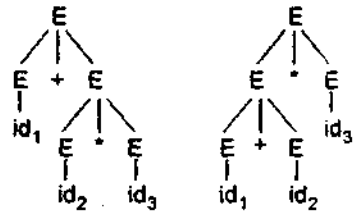
Ans. Ambiguity of Grammar : A grammar is said to be ambiguous if it produces more than one parse tree or if it produces more than one left or right derivative.

For example consider the grammer.

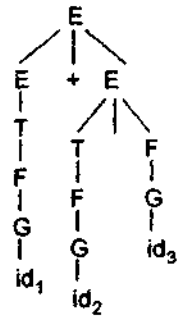
$E \rightarrow E + E \mid E * E \mid E \wedge E \mid id \mid (E)$

for a strid. $id_1 + id_2 \times id_3$

(1)



(2)



there is two parse tree for string $id_1 + id_2 * id_3$ so the above grammar is ambiguous.

For the most parsers, the grammar must be unambiguous. We should climinate the

www.btechsamplepapers.in

ambiguity in the grammar during the design phase of the compiler, we have to prefer one of the parse tree of a sentence to disambiguate the grammer. Ambiguous grammar can be disambiguate according to the precedence and associativity rule.

Example :

To disambiguate the grammar

$E \rightarrow E + E \mid E * E \mid E \wedge E \mid id \mid (E)$

We can use the precedence and associativity as follows :

^ (right to left)

* (left to right)

+ (left to right)

we get the following unambiguous grammar

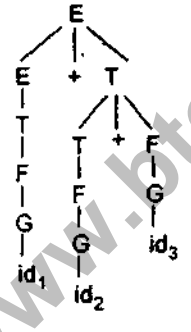
$E \rightarrow E + T/T$

$T \rightarrow T * F/F$

$F \rightarrow G \wedge F/G$

$G \rightarrow id/(E)$

now for string $id_1 + id_2 * id_3$



Only one parse tree is possible. So new grammar is unambiguous.

Q. 3. Attempt any two of the following sections. $10 \times 2 = 20$

Q. 3. (a) Explain how stack implementation of shift reduce parsing is done, considering the grammar :

$E \longrightarrow E + E$

$E \longrightarrow E * E$

$E \longrightarrow (E)$

$E \longrightarrow id$

and input string as $id_1 + id_2 * id_3$

Ans.

Stack	Input	Action
1. $	$id_1 + id_2 * id_3$ $	Shift
2. $id_1	$+ id_2 * id_3$ $	reduce by E → id
3. $E	$+ id_2 * id_3$ $	Shift
4. $E +	$id_2 * id_3$ $	Shift
5. $E + id_2	$* id_3$ $	reduce by E → id
6. $E + E	$* id_3$ $	Shift
7.$E + E*	id_3 $	Shift
8. $E + E*id_2	$	reduce by E → id
9. $E + E*E	$	reduce by E → E*E
10. $E + E	$	reduce by E → E + E
11. $E	$	accept

Q. 3. (b) What do you understand by left recursion and how it is eliminated ?

Ans. Left recursion : A grammar is left recursive if it has a non terminal. A such that there is a derivation $A \xrightarrow{+} A\alpha$ for some string α.

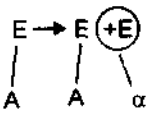
Example : A grammar

$E \rightarrow E + E$

$E \rightarrow E * E$

www.btechsamplepapers.in

is left recursive because there is a production



Elimination of Left Recursion

A left recursive pair fo production $A \rightarrow A\alpha/\beta$ could be replaced by non-left recursive productions.

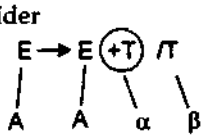
$$A \rightarrow \beta A^1$$
$$A^1 \rightarrow \alpha A^1/t$$

Example : Consider the grammar

$E \rightarrow E+T/T$
$T \rightarrow T*F/F$
} — left recursive production

$F \rightarrow (E)/id$

Consider



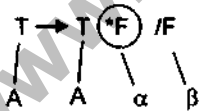
$A \rightarrow \beta A^1$ becomes

$E \rightarrow TE^1$

$A^1 \rightarrow \alpha A^1/E$

$E^1 \rightarrow +TE^1/t$

Consider



$T \rightarrow FT^1$

$T^1 \rightarrow *FT^1/t$

now non-left recorsive grammar is

$E \rightarrow TE^1$

$E^1 \rightarrow +TE^1/t$

$T \rightarrow FT^1$

$T^1 \rightarrow *FT^1/t$

$F \rightarrow (E)/id$

Q. 3. (c) Discuss the role of syntax directed translation scheme.

Ans. Syntax Directed Translation

Syntax-directed translation refers to a method of compiler implementation where the source language translation is completely driven by the parser. In other words, the parsing process and parse trees are used to direct semantic analysis and the translation of the source program. This can be a separate phase of a compiler or we can augment our conventional grammar with information to control the semantic analysis and translation. Such grammars are called *attribute grammars*.

We augment a grammar by associating *attributes* with each grammar symbol that describes its properties. As attribute has a name and an associated value : a string, a number, a type, a memory location, an assigned register—whatever information we need. For example, variables may have an attribute "type" (which records the declared type of a variable, useful later in type-checking) or an integer constant may have an attribute "value" (which we will later need to generate code).

With each production in a grammar, we give semantic rules or *actions*, which describe how to compute the attribute values associated with each grammar symbol in a production. The attribute value for a parse node may depend on information from its children nodes below or its siblings and parent node above.

Conider this production, augmented with a set of actions that use the "value" attribute for a digit node to store the apropriate numeric value. Below, we use the syntax x. a to refer to the attribute a associated with symbol x.

digit → 0 {digit . value =0}

1 {digit . value =1}

2 {digit . value =2}

.....

9 {digit . value =9}

Attributes may be passed up a parse tree to be used by other productions :

www.btechsamplepapers.in

1 int 1 → digit {int 1 . value = digit . value} int 2 digit {int 1 . value = int 2 . value * 10 + digit . value}

We are using subscripts in this example to clarify which attribute we are referring to, so int1 and int 2 are different instances of the same non-terminal symbol. There are two types of attributes we might encounter : synthesized or inherited.

(1) *Synthesized* attributes are those attributes that are passed up a parse tree, *i.e.*, the left 2 side attribute is computed from the right-side attributes.

(2) *Inherited* attributes are those that are passed down a parse tree, *i.e.*, the right-side attributes are derived from the left-side attributes (or other right-side attributes). These attributes are used for passing information about the context to nodes further down the tree.

$X \rightarrow Y_1 Y_2 \ldots . y_n$

Yk . a = f (X . a, Y1 . a, Y2 . a, ..., Yk-1 . a, Yk +1 . a, Yk + 1 . a, ..., Yn . a

Top-Down SDT

We can implement syntax-directed translation in either a top-down or a bottom-up parser and we'll briefly investigate each approach. First, let's look at adding attribute information to a hand- constructed top-down recursive-descent parser. Our example will be a very simple FTP client, where the parser accepts user commands and uses a syntax-directecd translation to act upon those requests. Here's in the grammar we'll use, already in an LL(1) form :

Session → CommandList T_QUIT

CommandList → Command CommandList | ε

Command → Login | Get | Logout

Login → User Pass

User → T_USER T_IDENT

Pass → T_PASS T_IDENT

Get → T_GET T_IDENT More Files

More Files → T_IDENT MoreFiles | ε

Logout → T_LOGOUT

Bottom-Up SDT

Here is a simple expression grammar that has associativity and precedence already built in.

E' → E

E → T | EAT

T → F | TMF

F → (E) | int

A → + | –

M → * | /

During the bottom-up parse, as we push symbols on to the parse stack, we will associate with each operand/expression symbol (E, T, F, etc.) an integer value. For each operator (A, M) we will store the operator code. When performing a reduction, we will synthesize the attribute for the left-side nonterminal from the attributes of the right side symbol, the handle that is currently on top of the stack.

Q. 4. Attempt any two of the following sections. $10 \times 2 = 20$

Q. 4. (a) Consider the following grammar :

S' = S#

S ⟶ ABC

A ⟶ A / bbD

B ⟶ a | ∈

C ⟶ b | ∈

D ⟶ d | ∈

Consruct the first and follow sets for the grammar also design LL(1) parsing table for the grammar.

Ans.

$S^1 \rightarrow S\ \#$

S → ABC

A → a/bbD

www.btechsamplepapers.in

B → a/t

C → b/t

D → c/t

First (S^1) = First (S) = First (A) = {a, b}

First (B) = {a, t}

First (C) = {b, t}

First (D) = {c, t}

Follow (S) = {#, $}

Follow (A) = {a, #, $}

Follow (B) = {b, #, $}

Follow (C) = {#, $}

Follow (D) = {a, #, $}

Parsing table

	a	b	c	#	$
S^1	$S^1 \rightarrow S\#$	$S^1 \rightarrow S\#$			
S	$S \rightarrow ABC$	$S \rightarrow ABC$			
A	$A \rightarrow a$	$A \rightarrow bbD$			
B		$B \rightarrow t$ $B \rightarrow b$		$B \rightarrow t$	$B \rightarrow E$
C		$C \rightarrow b$		$C \rightarrow \varepsilon$	$C \rightarrow t$
D	$D \rightarrow E$		$D \rightarrow c$	$D \rightarrow b$	$D \rightarrow E$

Q. 4. (b) Write the quadruples, triples and indirct triples for the following expression :

$(x + 4) * (y + z) + (x + y + z)$

Ans. $(x + 4) * (y + z) + (x + y + z)$

Quadruples :

	op	arg1	arg2	result
(0)	+	x	4	t_1
(1)	+	4	z	t_2
(2)	*	t_1	t_2	t_3
(3)	+	t_1	z	t_4
(4)	+	t_3	t_4	t_5

www.btechsamplepapers.in

triples

	op	arg1	arg2	
(10)	+	x	y	
(11)	+	y	z	
(12)	*	(10)	(11)	
(13)	+	(10)	z	
(14)	+	(12)	(13)	

Indirect triples

	Statements
(0)	(10)
(1)	(11)
(2)	(12)
(3)	(13)
(4)	(14)

Q. 4. (c) Discuss the types of errors with example which can be encountered by all the phases of the complier.

Ans. Different types of errors in compiler phase

(1) Lexical : Lexical analyzer creates tokens. When tokens are not created it reports errors.

Ex. Consider the statement

$c = a + b$

The different tokens are *c*, =, *a*, + , and b. If any one of the token is not created, compiler reports error.

(2) Syntax : When there is any problem regarding the construction of parse tree or any ambiguity then compiler reports errors.

(3) Syntatic : Syntatic phase is used to check the type of variable used in any expression. If the type of variable is not same this phase reports error.

Ex. $c = a + b * 5.0$

If a, b, c are of real type then there is no error but if any are of them or all of them is an integer then it reports error.

(4) Intermediate Code Generator

Intermediate code generates the 3 address codes for any expression.

Ex. $c = a + b$ is represented by 3 address code as

$t_1 = a$

www.btechsamplepapers.in

$t_2 = b$

$t_3 = t_1 + t_2$

$c = t_3$

If there is any problem regarding the construction of 3 address code, it reports error.

(5) Code Optimization

Code optimization is used to reduce the number of 3 address code generated in Intermediate code generation.

Ex. $c = a + b$ is reduced as

$t_1 = a + b$

$c = t_1$

If after the completion of this phase, intermediate code is not reduced then an error is reported.

Q. 5. Write short notes on any two of the following : **10 × 2 = 20**

Q. 5. (a) Induction variable eleminaion

Ans. Induction Variable Elimination : Some loops contain two or more induction variables that can be combined into one induction variable.

Example : The code fragment below has three induction variables (i1, i2, and i3) that can be replaced with one induction variable, thus eliminating two induction variables.

```
int a [SIZE];
int b [SIZE];
void f (void)
{
int i1, i2, i3;
for (i1 = 0, i2 = 0, i3 = 0; i1 < SIZE; i1+ +)
a [i2+ +] = b [i3+ +];
return;
}
```

The code fragment below shows the loop after induction variable elimination.

```
int a [SIZE];
int b [SIZE];
void f (void)
{
int i1;
for (i1 = 0; i1 < SIZE; i1+ +)
a [i1] = b [i1];
return;
}
```

Induction variable elimination can reduce the number of additions (or subtractions) in a loop, and improve both run-time performance and code space.

Some architectures have auto-increment and auto-decrement instructions that can sometimes be used instead of induction variable elimination.

Q. 5. (b) DAG representation.

Ans. A **directed acyclic graph**, also called a **DAG**, is a **directed graph** with no **directed cycles**; that is, for any **vertex** v, there is no nonempty **directed path** that starts and ends on v.[1][2][3]

The reachability relation in a DAG forms a partial order, and any finite partial order may be represented by a DAG using reachibility. DAGs may also be used to model proceses in which information flows in a consistent direction through a network of processors; and to provide space-efficient data structures for representing sets of sequences.

DAG Construction

- Assume there are initially no nodes and MODE ( ) is undefined for all arguments.
- The 3 address statements has one of three cases :

(i) A = B OP C

(ii) A = OP B

www.btechsamplepapers.in

(iii) A = B

Follow the following steps :

(i) If MODE (B) is undefined, create a leaf labelled B and let MODE (B) be this node.

(ii) In case (i), determine if there is a node labelled OP whose left chiled is MODE (B) and right child MODE (C) if not create such a node. In case (ii), deterine whether there is a node labelled OP whose lone child to MODE (B). It not create such a node. Let n be the node or created in both cases. In case (ii) let n be node (B).

(iii) Append A to the list of attached identifiers for the node n in (ii). Delete A from the list attached identifiers for MODE (A). Finally set MODE (A) to n.

Application of DAG

(i) Delete common sub expression.

(ii) Deterine which statements compute values which could be used to outside the block.

Q. 5. (c) Loop unrolling and Loop jamming.

Ans. The loop unroll and jam transformation is primarily intended to increase register exploitation and decrease memory loads and stores per operation within an iteration of a nested loop. Improved register usage decreases the need for main accesses and allows better exploitation of certain machine instructions.

Unroll and jam involves partially unrolling one or more loops higher in the nest than the innermost loop, and fusing ("jamming") the resulting loops back together. For unroll and jam to be effective, a loop must be nested and must contain data references that are temporally reused with respect to some loop other than the innermost (temporal reuse is described in **"Data reuse"**). The unroll and jam optimization is automatically applied only to those loops that consist strictly of a basic block.

Loop unroll and jam takes place at + 03 and above and is not enabled by default in the HP compilers. To enable loop unroll and jam on the command line, use the + 0loop_unroll_jam option. This allows both automatic and directive-specified unroll and jam. Specifying +0 noloop_transform disables loop unroll and jam, loop distribution, loop interchange, loop blocking, loop fusion, and loop unroll.

The unroll_and_jam directive and pragma also enables this transformation. The no_unroll_and_jam directive and pragma is used to disable loop unroll and jam for an individual loop.

Consider the following matrix multiply loop :

```
DO I = 1, N
  DO J = 1, N
   DO K = 1, N
        A (I, J) = A (I, J) + B (I, K) * C (K, J)
          ENDDO
        ENDDO
      ENDDO
```

Here, the compiler can exploit a maximum of 3 registers : one for A (I, J), one for B (I, K), and one for C (K, J).

Register exploitation is vastly increased on this loop by unrolling and jamming the I and J loops. First, the compiler unrolls the I loop. To simplify the illustration, an unrolling factor of 2 for I is used. This is the number of times the contents of the loop are replicated.

The following Fortran example shows this replication :

```
DO I = 1, N, 2
```

www.btechsamplepapers.in

```
      DO J = 1, N
        DO K = 1, N
            A (I, J) = A (I, J) + B (I, K) * C (K, J)
        ENDDO
      ENDDO
      DO J = 1, N
        DO K = 1, N
              A (I + 1, J) = A (I + 1, J) + B (I +
1, K) * C (K, J)
        ENDDO
      ENDDO
ENDDO
```

The "jam" part of unroll and jam occurs when the loops are fused back together, to create the following :

```
DO I = 1, N, 2
      DO J = 1, N
        DO K = 1, N
          A (I, J) = A (I, J) + B (I, K) * C (K, J)
        A (I + 1, J) = A (I + 1, J) + B (I + 1, K) * C
          (K, J)
        ENDDO
      ENDDO
ENDDO
```

www.btechsamplepapers.in