

--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

Internal Assessment Test 3 Answer Key– Dec. 2020

Sub:	System Software					Sub Code:	18MCA34	Branch:	MCA
Date:	15/12/2020	Duration:	90 min's	Max Marks:	50	Sem	III A		

Q1) What do you mean by macro? Explain macro definition and expansion, with suitable example [10]

Macro Definition and Expansion: The figure shows the MACRO expansion. The left block shows the MACRO definition and the right block shows the expanded macro replacing the MACRO call with its block of executable instruction.

M1 is a macro with two parameters D1 and D2. The MACRO stores the contents of register A in D1 and the contents of register B in D2. Later M1 is invoked with the parameters DATA1 and DATA2, Second time with DATA4 and DATA3. Every call of MACRO is expended with the executable statements.

Source	Expanded source
M1 MACRO &D1, &D2	.
STA &D1	.
STB &D2	.
MEND	{ STA DATA1
.	STB DATA2
M1 DATA1, DATA2	.
.	{ STA DATA4
M1 DATA4, DATA3	STB DATA3
.	.

The statement M1 DATA1, DATA2 is a macro invocation statements that gives the name of the macro instruction being invoked and the arguments (M1 and M2) to be used in expanding. A macro invocation is referred as a Macro Call or Invocation.

The program with macros is supplied to the macro processor. Each macro invocation statement will be expanded into the statements that form the body of the macro, with the arguments from the macro invocation substituted for the parameters in the macro prototype. During the expansion, the macro definition statements are deleted since they are no longer needed. The arguments and the parameters are associated with one another according to their positions. The first argument in the macro matches with the first parameter in the macro prototype and so on.

Q2) What are different Macro Processor Design options? Explain briefly [10]

1) Recursive Macro Expansion

We have seen an example of the definition of one macro instruction by another. But we have not dealt with the invocation of one macro by another.

The following example shows the invocation of one macro by another macro.

Problem of Recursive Expansion

Previous macro processor design cannot handle such kind of recursive macro invocation and expansion

The procedure EXPAND would be called recursively, thus the invocation arguments in the ARGTAB will be overwritten.

The Boolean variable EXPANDING would be set to FALSE when the "inner" macro expansion is finished, i.e., the macro process would forget that it had been in the middle of expanding an "outer" macro.

2) General-Purpose Macro Processors

Macro processors that do not dependent on any particular programming language, but can be used with a variety of different languages

Pros

- ☒ Programmers do not need to learn many macro languages.
- ☒ Although its development costs are somewhat greater than those for a language specific macro processor, this expense does not need to be repeated for each language, thus save substantial overall cost.

Cons

- ☒ Large number of details must be dealt with in a real programming language Situations in which normal macro parameter substitution should not occur, e.g., comments.
- ☒ Facilities for grouping together terms, expressions, or statements ☒ Tokens, e.g., identifiers, constants, operators, keywords
- ☒ Syntax had better be consistent with the source programming Language

3) Macro Processing within Language Translators

The macro processors we discussed are called "Preprocessors".

- ☒ Process macro definitions
- ☒ Expand macro invocations
- ☒ Produce an expanded version of the source program, which is then

used as input to an assembler or compiler

You may also combine the macro processing functions with the language translator:

- ☒ Line-by-line macro processor
- ☒ Integrated macro processor

Line-by-Line Macro Processor

Used as a sort of input routine for the assembler or compiler

- ☒ Read source program
- ☒ Process macro definitions and expand macro invocations
- ☒ Pass output lines to the assembler or compiler

Benefits

- ☒ Avoid making an extra pass over the source program.
- ☒ Data structures required by the macro processor and the language translator

can be combined (e.g., OPTAB and NAMTAB)

- ☒ Utility subroutines can be used by both macro processor and the language translator.

Scanning input lines

Searching tables

Data format conversion

- ☒ It is easier to give diagnostic messages related to the source statements

Integrated Macro Processor

An integrated macro processor can potentially make use of any information about the source program that is extracted by the language translator.

Ex (blanks are not significant in FORTRAN)

DO 100 I = 1,20

a DO statement

DO 100 I = 1

An assignment statement

DO100I: variable (blanks are not significant in FORTRAN)

An integrated macro processor can support macro instructions that depend upon the context in which they occur.

Q3) Discuss all the machine Independent macro features.[10]

The design of macro processor doesn't depend on the architecture of the machine. We will be studying some extended feature for this macro processor.

These features are:

- Concatenation of Macro Parameters
- Generation of unique labels
- Conditional Macro Expansion
- Keyword Macro Parameters

Most macro processor allows parameters to be concatenated with other character strings. Suppose that a program contains a series of variables named by the symbols XA1, XA2, XA3, ..., another series of variables named XB1, XB2, XB3, ..., etc. If similar processing is to be performed on each series of labels, the programmer might put this as a macro instruction. The parameter to such a macro instruction could specify the series of variables to be operated on (A, B, etc.). The macro processor would use this parameter to construct the symbols required in the macro expansion (XA1, Xb1, etc.).

Suppose that the parameter to such a macro instruction is named &ID. The body of the macro definition might contain a statement like

```
LDA X&ID1
```

Generation of Unique Labels

It is not possible to use labels for the instructions in the macro definition, since every expansion of macro would include the label repeatedly which is not allowed by the assembler.

This in turn forces us to use relative addressing in the jump instructions. Instead we can use the technique of generating unique labels for every macro invocation and expansion.

During macro expansion each \$ will be replaced with \$XX, where xx is a two-character alphanumeric counter of the number of macro instructions expansion. For example, XX = AA, AB, AC...

Conditional Macro Expansion

- Macro-time conditional statements - IF-ELSE-ENDIF
- Macro-time variables
- Macro-time looping statement
- Macro processor function

Macro processor function

```
%NITEMS: THE NUMBER OF MEMBERS IN AN ARGUMENT LIST
```

Q4) Explain machine Dependent compiler Features [10]

1) Intermediate form of the Program

There are many possible ways of representing a program in an intermediate form for code analysis and optimization. One of the methods is representing the executable instructions of the program with a sequence of quadruples.

Each quadruple is of the form

Operation, op1, op2, result

Operation – function to be performed by the object code

Op1 and op2 operands for this operation

Result – where the resulting value is to be placed

Example :

SUM := SUM + VALUE

Could be represented with the quadruples

+, SUM, VALUE, i1

:=, i1, , SUM

These quadruples would be created by intermediate code generation routines.

Many types of analysis and manipulation can be performed on the quadruples for code-optimization purpose.

After optimization, the modified quadruples are translated into machine code.

Quadruples appear in the order which the corresponding object code instructions are to be executed.

This greatly simplifies the task of analyzing the code for the purpose of optimization. It also means that the translation into machine instructions will be relatively easy.

2) Machine Dependent Code optimization

There are several different possibilities for performing machine dependent code optimization.

1) Assignment and use of registers

General purpose registers are used for various purposes like storing values or intermediate results or for addressing (base register, index register).

Registers are also used as instruction operands. Machine instructions that use registers as operands are usually faster than the corresponding instructions that refer to locations in memory. Therefore it is preferable to store values or intermediate results in registers.

There are rarely as many registers available as we would like to use. The problem then becomes one of selecting which register value to replace when it is necessary to assign a register for some other purpose.

One approach is to scan the program and the value that is not needed for the longest time will be replaced. If the register that is being reassigned contains the value of some variable already stored in memory, the value can be simply discarded. Otherwise this value must be saved using a temporary variable.

Second approach is to divide the program into basic blocks. A basic block is a sequence of quadruples with one entry point, which is at the beginning of the block, one exit point, which is at the end of the block and no jumps within the block. When control passes from one block to another all the values are stored in temporary variables.

2) Rearranging quadruples before machine code is generated. Note that the value of the intermediate result *i1* is calculated first and stored in temporary variable *T1*. Then the value of *i2* is calculated. The third quadruple in this series calls for subtracting the value of *i2* from *i1*. Since *i2* had just been computed, its value is available in registers *A*; however, this does no good, since the first operand for a $-$ operation must be in a register. It is necessary to store the value of *i1* from *T1* into register *A* before performing the subtraction.

With a little analysis, an optimizing compiler could recognize this situation and rearrange the quadruples so the second operand of the subtraction is computed first. The resulting machine code requires two fewer instructions and uses only one temporary variable instead of two.

3) Taking advantage of specific characteristics and instructions of the target machine

For example there may be special loop-control instructions or addressing modes that can be used to create more efficient object code.

On some computers there are high level machine instructions that can perform complicated functions such as calling procedures and manipulating data structures in single operations.

Use of such features can greatly improve the efficiency of the object program.

CPU is made of several functional units. On such a system machine instruction order can affect speed of execution. Consecutive instructions that require different functional units can be executed at the same time.

Q5) Explain Machine independent code optimization and structured variables [10]

1) Structured variables

Arrays, records, strings, and sets are examples of structured variables. We are primarily concerned with the allocation of storage for such variables and with generation of code to reference them.

General array declaration:

```
ARRAY [l...u] OF INTEGER
```

Then we must allocate $u-l+1$ words of storage for the array

Eg : A : ARRAY[1...10] OF INTEGER

Multidimensional array declaration

ARRAY[l1..u1,l2...u2] OF INTEGER

The number of words allocated is given by

$$(u1-l1+1)*(u2-l2+1)$$

All the array elements that have the same value of the first subscript are stored in contiguous locations; this is called row-major order.

All elements that have the same value of the second subscript are stored together; this is called column-major order.

Compilers for most of high level languages store arrays using row-major order.

If an array reference involves only constant subscripts, the relative address calculation can be performed during compilation.

Relative address of the referenced array element A[s] is given by

$$W*(s-l)$$

The relative address of B[s1,s2] is given by

$$W * [(s1-l1) * (u2-l2+1) + (s2-l2)]$$

Dynamic arrays could be declared as

INTEGER, ALLOCABLE, ARRAY(: , :) :: MATRIX

The allocation can be accomplished by a statement like

ALLOCATE(MATRIX (ROWS, COLUMNS))

In dynamic arrays values of row and columns are not known at compilation time, the compiler cannot directly generate code. Instead, compiler creates descriptor (often called as dope vector) for the array. This descriptor includes space for storing the lower and upper bounds for each array subscript. When the storage is allocated value of these bounds are computed and stored in the descriptor.

Machine Independent code optimization

1) Elimination of common sub expressions.

One important source code optimization is the elimination of common sub expressions. These sub expressions that appear at more than one point in the program and that compute the same value.

Common sub expressions are usually detected through the analysis of an intermediate form of the program

Example:

1) := #1 I

2)

3)

4)

5) * #2 J i3

6)

7)

8)

9)

10)

11)

12) * #2 J i10 .

We see that quadruples 5 and 12 are same except for the name of the intermediate result produced. Operand J does not change value between 5 and 12. It is not possible to reach quadruples 12 without passing through 5. This means we can delete quadruple 12 and replace any reference to its result (i10) with reference to i3, the result of quadruple 5. This modification eliminates the duplicate calculation of $2*J$, which we identified previously as a common sub expression in the source statement

2) Removal of loop variants.

These are the sub expressions within a loop whose values do not change from one iteration of the loop to the next. Thus their values can be computed once before loop is entered, rather than being recalculated for each iteration. Because most programs spend most of their running time in the execution of loops, the time saving from this sort of optimization can be highly significant.

Example

Loop-invariant computation is the term $2*j$. the result of this computation depends only on the operand J, which does not change in value during the execution of the loop. Thus it can be moved to the point immediately before the loop is entered.

3) Substitution of more efficient operation for less efficient one.

Consider following example:

DO 10 I = 1,20

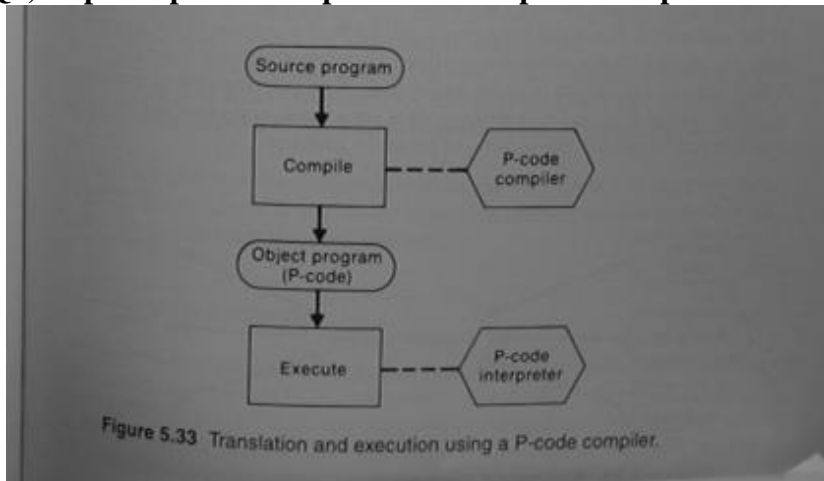
TABLE(I) = 2**I

This DO loop creates a table that contains the first 20 powers of 2. On closer examination, we can see that there is a more efficient way to perform the computation. For each iteration of the loop, the value of I increased by 1. Therefore, the value of $2**I$ for the current iteration can be found by multiplying the value for the previous iteration by 2. Clearly this method of computing $2**I$ is much more efficient than performing a series of multiplications or using a logarithmic technique. Such a transformation is called reduction in strength of an operation.

There are number of other possibilities for machine-impendent code optimization. For example, computations whose operand values are known at compilation time can be performed by the compiler. This optimization is known as folding.

Other optimization include converting a loop into straight line code(loop unrolling) and margining of the bodies of loop (loop jamming))

Q6) Explain p code compiler and Compiler-Compiler with neat diagram [10]



P-code compilers (also called byte code compilers) are very similar in concept to interpreters.

In both cases, the source program is analyzed and converted into an intermediate form, which is then executed interpretively.

With a P-code compiler, however, this intermediate form is the machine language for a hypothetical computer, often called pseudo-machine or P-machine.

The source program is compiled, with the resulting object program being in P-code.

This P-code program is then read and executed under the control of a P-code interpreter

The main advantage of this approach is portability of software. It is not necessary for the compiler to generate different code for different computers, because the p-code object programs can be executed on any machine that has a p-code interpreter.

Even the compiler itself can be transported if it is written in the language that it compiles. To accomplish this, the source version of the compiler is compiled into p-code; this p-code can then be interpreted on another computer.

In this way a p-code compiler can be used without modification on a wide variety of system if a p-code interpreter is written for each different machine.

The design of a P-machine and the associated P-code is often related to the requirements of the language being compiled.

The interpretive execution of a p-code program may be much slower than the execution of the equivalent machine code.

P-code object program is often much smaller than a corresponding machine-code program would be. This particularly useful in machines with severely limited memory size

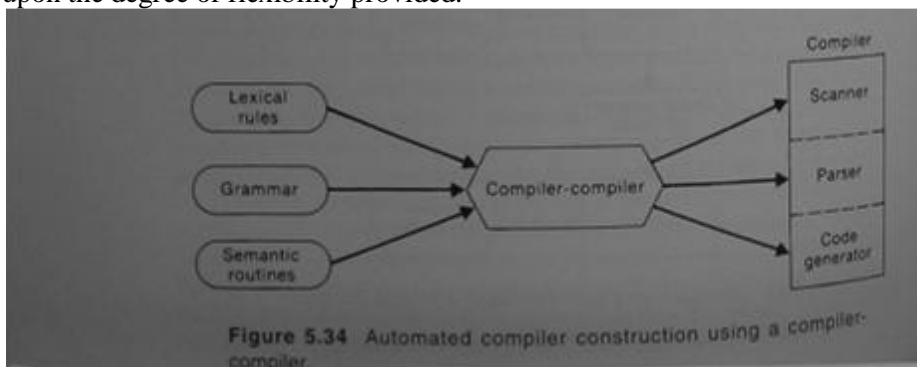
Many p-code compilers designed for a single user running on dedicated microcomputer system.

If execution speed is important some P-code compilers support the use of machine language subroutines.

By rewriting a small number of commonly used routines in machine language, rather than P-code, it is often possible to achieve substantial improvements in performance. But this approach sacrifices some of the portability associated with the use of P-code compiler

Compiler-compiler

- A compiler-compiler is a software tool that can be used to help in the task of compiler construction.
- Such tools are often called compiler generators or translator-writing systems.
- The process of using a typical compiler-compiler is illustrated below.
- The user provides a description of the language to be translated. This description may consist of a set of lexical rules for defining tokens and a grammar for the source language.
- Some compiler-compilers use this information to generate a scanner and a parser directly.
- Others create tables for use by standard table driven scanning and parsing routines that are supplied by the compiler-compiler.
- In addition to the description of the source language, the user provides a set of semantic or code-generation routines.
- Compiler-compilers frequently provide special languages, notations, data structures and other similar facilities that can be used in the writing of semantic routines.
- The main advantage of using a compiler-compiler is of course ease of compiler construction and testing.
- The amount of work required from the user varies considerably from one compiler-compiler to another depending upon the degree of flexibility provided.



Q7a) Assume the array B is declared B: ARRAY{1...4,1...5} OF INTEGER with each element occupying 3 bytes. Generate quadruples for the statement B[I,J] :=6

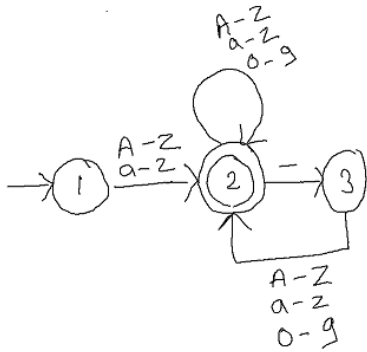
```

-   I   #1   i1
*   i1  #5   i2
-   J   #1   i3
+   i2  i3   i4
    
```

* 3 i4 i5
 := #6 B[i5]

Q7b) Indicate whether the finite automation given in below fig recognizes the following string

- 1)4Beta 2) hello_5 3)hello10 4) alpha2beta 5) alpha_ 3delta 6) charlie5_
 7)Charlie_detlat2 8)abcccccab 9)r0123abc 10)34abc



- | | |
|-------------------|------------------|
| 1)4Beta | - Not recognized |
| 2) hello_5 | - Recognized |
| 3)hello10 | - Recognized |
| 4) alpha2beta | - Recognized |
| 5) alpha_ 3delta | - Not recognized |
| 6) charlie5_ | - Not recognized |
| 7)Charlie_detlat2 | - Recognized |
| 8)abcccccab | - Recognized |
| 9)r0123abc | - Recognized |
| 10)34abc | - Not recognized |

Q*) What is grammar? Using the BNF grammar below, represent the syntax of the pascal statement

VAR := SUMSQ DIV 100 - MEAN * MEAN in the form of a parse tree

