BIRLA INSTITUTE OF TECHNOLOGY, MESRA, RANCHI
(END SEMESTER EXAMINATION)

CLASS: BTECH | **SEMESTER : VI**
BRANCH: CSE | **SESSION : SP/2025**

SUBJECT: CS333 COMPILER DESIGN

TIME: 3 Hours | **FULL MARKS: 50**

INSTRUCTIONS:
1. The question paper contains 5 questions each of 10 marks and total 50 marks.
2. Attempt all questions. Answers without justification will get reduced credit.
3. Unless stated otherwise, all programs in the questions are given to be error free. Missing data, if any, may be assumed suitably.
4. Before attempting the question paper, be sure that you have got the correct question paper.
5. A data sheet at the end gives SDTS, X86 architecture and caller callee conventions.

			CO	BL

Q.1(a) Consider the following C code. [5] CO 1

```
*float x;                      // D1
int num;                       // D2
int main() {
int x  5;                      // D3
short num[2]={3, 9999999};     // D4
double x;                      // D5
char ch[1]={"CD", "OS'};       // D6
}
```

(i) Identify all the distinct lexemes in the code given in Column 1 and group them into tokens. Write the token along with the set of lexemes it represents.

(ii) Identify errors or warnings, if any, in each of the declarations, D1 to D6 in the code, as lexical, syntax, semantic or runtime errors.

Q.1(b) Consider the following 10 terms that are used in the context of compiler design. **FOR ANY 5 TERMS FROM THE GIVEN LIST**, describe the term in brief and state the phase of the compiler in which it is used or applied. [5] CO 2

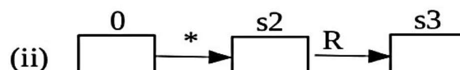
1. Viable prefix 2. Activation Record 3. Shift reduce conflicts 4. Type conversion
5. Symbol table creation 6. FIRST and FOLLOW 7. Translation of Parameter passing
8. Control flow graph 9. Deterministic finite automata 10. Loop invariant code motion

Q. 2(a) Examine the following CFG, G = (T, N, P, S) where T has 3 tokens { **id,** =, *}, N has 3 nonterminals, P has 6 production rules and start symbol is S. A rule with nonterminal X in the LHS is defined to be left or right recursive, if X =>* Xα or X =>* βX respectively; where =>* denotes derivation in zero or more steps. [5] CO 3

1, 2) S → L = R | R   3, 4) L → * R | R   5, 6) R → * L | **id**

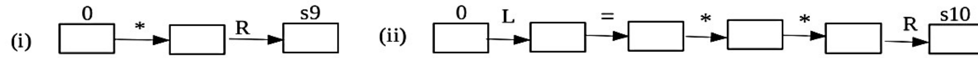
Do not change the grammar for this part.
(i) Determine if G is left or right recursive with evidence.
(ii) Construct and write FIRST() and FOLLOW() for all nonterminals of G.
(iii) Construct **LL(1)** parsing table and report ONLY the rows corresponding to the nonterminals S and L. Conclude with reasons whether G is a LL(1) grammar.

Q.2(b) Consider the grammar of **Q2(a)** above, augmented with the rule, **S' → S**, where S' is the new start symbol. You have to construct the sets of LR(0) items for the augmented grammar for the states s1 and s3 which are reached from state 0 after the state transitions given below. Write the sets of all LR(0) items in the states s1 and s3. Construct the rows of SLR(1) parsing table ONLY FOR s1 and s3. Determine conflicts, if any, with reasons. A state is represented by a rectangle in the figure. [5] CO 3



PTO

Q.3(a) Consider the augmented grammar of Q2(b). You have to construct Canonical LR(1) parser for this [5] 3



grammar as described below. The path in the CLR(1) automaton from the start state 0 to two particular states of interest, named as s9 and s10, are given below. The intermediate states are not named. You have to write all the LR(1) sets of items (**LR(0) item with the lookahead**) of s9 and s10. Using the sets of LR(1) items constructed by you, show the CLR(1) table entries for these **2 states only**. Report conflicts, if any, in these states with reasons.

Q.3(b) Consider the grammar G for arithmetic expressions whose operators follow C semantics. Terminals are {+ * **id**}. [5] 3

1, 2) E → E + T | T   3, 4) T → T * F | F   5) F → **id**

Write a SDTS using synthesized attributes only so that it can convert infix expressions to prefix form. For example, given the expression, " a + b * c + d * e" the SDTS prints the expression, "+ + a * b c * d e".

Your SDTS should not change the properties of the operators. You are given a function, cat(s1,s2,s3), that can concatenate all its argument strings and return a single string. You may augment the given grammar, but do not change the rules given.

Q.4(a) Consider the code fragment given below. It is required to generate intermediate code for this code using partial evaluation for boolean expressions and the SDTS given in the data sheet. [5] 4

```
a = 15; b = 8;
if true
 then  begin a = a + b; b = a – b end ↕1
 else
   while b > 0 do begin a = 2 * a; b = b /2 end ↕2
sum = a + b;
```

(i) Show the intermediate code and the attribute values that are generated when the parser and the SDTS, reaches the two points marked by the symbols, $\updownarrow_1$ and $\updownarrow_2$.

(ii) Generate intermediate code for the entire fragment. Construct and display the control flow graph. Also detect unreachable code, if present, in the control flow graph.

Q.4(b) A new construct **repeat** S **while** B, has to be added to the SDTS for control structures. The semantics of this loop is that its body S is executed at least once. Subsequent execution of S depends on the boolean expression B. In case B evaluates to **False**, the body S is repeated, else if B evaluates to **True**, the loop is exited. [5] 3

S →**repeat** S_1 **while B**

(i) Write a SDTS for repeat-while construct that honours the semantics described above on the same lines as the SDTS for other control constructs are provided. You have to use partial evaluation of boolean expressions. Write all the attributes used by you along with their intended purpose. If you change the SDTS that is already given, state the changes made by you along with the reasons.

(ii) Manually generate intermediate code using your SDTS of part (i) for the code fragment given below.

```
repeat  if  a < 0  then
                    while  c > 0  do begin a = a + d ; c = a + c end
while ( a <= b);   a = a+b
```

Q.5(a) Consider the C program given below. [5] 4

```
#include <stdio.h>
int c = 4; static int val = 5.0;
int f(int x) {int res = x*val + c; return res;}
```

```
int main()
{ int a, b = 2;  a = b + c;
  printf ("a = %d f(val) = %d \n", a, f(val));
  return 0;}
```

PTO

Trimmed assembly code for f() and main() are given below. You have to examine the assembly code and write the assembly code against the two situations given below. In both case, you are required to write only the assembly code statements, along with the task they perform with respect to run time environment. Assembly statements that are not concerned are not to be included. (i) assembly code in f() that relate to each task done by it as a callee function.

(ii) assembly code in main() that are concerned with the implementation of the printf() statement, mention task performed by each assembly code:

printf ("a = %d f(val) = %d \n", a, f(val));

Assembly code for f()

```
f:
    endbr64
    pushq  %rbp   #
    movq   %rsp, %rbp     #,
    movl   %edi, -20(%rbp) # x, x
    movl   val(%rip), %eax # val
    imull  -20(%rbp), %eax # x,
    movl   %eax, %edx     # val.0_1, _2
    movl   c(%rip), %eax  # c, c.1_3
    addl   %edx, %eax     # _2, tmp87
    movl   %eax, -4(%rbp) # tmp87, res
    movl   -4(%rbp), %eax # res, _7
    popq   %rbp   #
    ret

.LC0:
    .string "a = %d f(val) = %d \n"
    .text
    .globl  main
    .type   main, @function
```

Assembly code for main()

```
main:
    endbr64
    pushq  %rbp   #
    movq   %rsp, %rbp     #,
    subq   $16, %rsp      #,
    movl   $2, -8(%rbp)   #, b
    movl   c(%rip), %edx  # c, c.2_1
    movl   -8(%rbp), %eax # b, tmp90
    addl   %edx, %eax     # c.2_1, tmp89
    movl   %eax, -4(%rbp) # tmp89, a
    movl   val(%rip), %eax # val, val.3_2
    movl   %eax, %edi     # val.3_2,
    call   f      #
    movl   %eax, %edx     #, _3
    movl   -4(%rbp), %eax # a, tmp91
    movl   %eax, %esi     # tmp91,
    leaq   .LC0(%rip), %rax      #, tmp92
    movq   %rax, %rdi     # tmp92,
    movl   $0, %eax       #,
    call   printf@PLT     #
    movl   $0, %eax       #, _9
    leave
    ret
```

Q.5(b) Examine the following C program, which is known to execute successfully and produce its desired output. The source code is required to be made efficient by applying one or more of the following optimizations, constant folding, constant propagation, unreachable code removal, loop invariant code movement, removal of dead code, and any other optimization that you may wish to define and apply. The optimizations are to be applied to the source code only. [5] 4

(i) Apply as many optimization as you can and show your result in the format : Name of the optimization; code fragment to which applied; reason for correctness of the optimization; changed source code before and after the optimization for the affected code only.

(ii) Show the final optimized source code. Make sure that your optimized source produces the same output as the original program.

```
#include <stdio.h>
int main()
{ int a = 2, b = 3, c = 40, d, i, j;
 int x[10]={10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
 for ( i = 1; i < 11; i+=2 )
 { if ( i%2)  { d = a * b;  x[i] = x[i-1] + d; c = b*100; }
   else   {d = a * a; x[i+d] = x[i]+x[i+1];}
 };
 printf(" a = %d b = % d c = %d d = %d x[6] = %d \n", a, b, c, d, x[6]); return 0;}
```

DATA SHEET ON COMPILER DESIGN

A few SDTS are given below, you may use them with or without modifications as per your need.

1. SDTS for Full Evaluation of Boolean expressions

$B \to B_1$ **or** B_2 { B.place := newtemp(); emit(B.place ':=' B_1.place 'or' B_2.place) }
$B \to B_1$ **and** B_2 { B.place := newtemp(); emit(B.place ':=' B_1.place 'and' B_2.place) }
$B \to$ **not** B_1 { B.place := newtemp(); emit(B.place ':=' 'not' B_1.place) }
$B \to (B_1)$ { B.place := B_1.place) }
$B \to$ **true** { B.place := newtemp(); emit (B.place ':=' '1') }
$B \to$ **false** { B.place := newtemp(); emit (B.place ':=' '0') }
$B \to E$ { B.place := E.place}
$B \to E_1$ **relop** E_2 { B.place := newtemp(); emit ('if' E_1.place '**relop.op**' E_2.place 'goto' nextstat+3);
emit (B.place ':=' '0'); emit ('goto' nextstat + 2); emit (B.place ':=' '1')}

2. Partial evaluation of boolean expressions :

$B \to B_1$ **or** M B_2 { backpatch (B_1.falselist, M.stat); B.truelist := merge (B_1.truelist, B_2.truelist);
B.falselist := B_2.falselist}
$B \to B_1$ **and** M B_2 {backpatch (B_1.truelist, M.stat); B.falselist := merge (B_1.falselist, B_2.falselist);
B.truelist := B_2.truelist}
$B \to$ **not** B_1 {B.truelist := B_1.falselist ; B.falselist := B_1.truelist }
$B \to (B_1)$ {B.truelist := B_1.truelist ; B.falselist := B_1.falselist }
$B \to$ **true** {B.truelist := makelist (nextstat); emit ('goto ___');
$B \to$ **false** {B.falselist := makelist (nextstat); emit ('goto ___');
$B \to E_1$ **relop** E_2 {B.truelist := makelist (nextstat); B.falselist := makelist (nextstat + 1);
emit ('if' E_1.place 'relop.op' E_2.place 'goto ___'); emit ('goto ___') }
$B \to E$ {B.place := E.place; B.truelist := makelist (nextstat); B.falselist := makelist (nextstat + 1);
emit ('if' B.place 'goto ___'); emit ('goto ___') }
$M \to \varepsilon$ {M.stat := nextstat}

3. SDTS for control flow constructs

$S \to$ if B then M S_1 {backpatch (B.truelist, M.stat); S.nextlist := merge(B.falselist, S_1.nextlist) }
$S \to$ if B then M_1 S_1 N else M_2 S {backpatch (B.truelist, M_1.stat); backpatch (B.falselist, M_2.stat);
S.nextlist := merge(S_1.nextlist, merge(N.nextlist, S_2.nextlist))}
$S \to$ while M_1 B do M_2 S_1 {backpatch (B.truelist, M_2.stat); backpatch (S_1.nextlist, M_1.stat);
S.nextlist := B.falselist ; emit ('goto' M_1.stat)}
$S \to$ begin Slist end {S.nextlist := Slist.nextlist}
Slist $\to$ $Slist_1$; M S {backpatch ($Slist_1$.nextlist, M.stat); Slist.nextlist := S.nextlist}
Slist $\to$ S {Slist.nextlist := S.nextlist}
$S \to$ id = E { p := table-lookup(id.name);
if $p \neq$ nil then emit (p ':=' E.place) else error; S.nextlist := nil }
$E \to E_1 + E_2$ {E.place := newtemp(); emit (E.place ':=' E_1.place '+' E_2.place) }
Similar rules and actions for other operators
$M \to \varepsilon$ {M.stat := nextstat}
$N \to \varepsilon$ {N.nextlist := makelist (nextstat); emit ('goto ___ ')}

4. Generic Caller Callee Conventions : Details may vary across architectures

Procedure Call		Return from a procedure	
Tasks done by Caller	**Tasks done by Callee**	**Tasks done by Caller**	**Tasks done by Callee**
1. Make space for a return value. 2. Pass actual parameters through registers and / or the stack. 3. Jump to the called procedure. Save own return address on the stack.	1. Sets the dynamic link, that is save base address of caller 2. Sets base of the new activation record for itself 3. Make space for local variables on the stack. 4. Save actual parameters to their assigned space for formal arguments in its activation record.	1. Access the return value from its callee, either from a register or from the stack.	1. Save return value is in a designated register or stack. 2. Set the base pointer to the activation record of the caller. 3. Return control to the code of the caller.

X 86 Registers and their use

%rsp Stack Pointer	%rax Return value	%rdi 1st argument	%rsi 2nd argument
%rdx 3rd argument	%rcx 4th argument	%r8 5th argument	%r9 6th argument
%rip instruction pointer	%rbp base pointer		

Pseudo code and their corresponding assembly instructions

Pseudo code	Assembly code	Pseudo code	Assembly code	Pseudo code	Assembly code
call g	pushq %rip; jmp &g	**leave**	movl %rbp, %rsp popq %rbp	**ret**	popq %rip

End of Data Sheet

:::::29/04/2025 M:::::