BRAC UNIVERSITY

Department of Computer Science and Engineering

Examination: Final Exam | Semester: Summer 2025

Duration: 1 Hour 30 Minutes | **SET - B** | Full Marks: 30

CSE 420: Compiler Design

Figures in the right margin indicate marks. Answer all the questions.

Name:	Student ID:	Section:

1.*(C04)* Below is the complete attribute grammar in SDT format for a programming language along with necessary global utility methods. Answer the following questions using the production and SDT rules of this grammar.

P -> *{counter = 0; S.next = newlabel("s_");*
top = new SymbolTable(); off = 0;
offStack = stack(); tabStack = stack();}
S *{P.code = S.code || Label(S.next);}*

S -> assign *{S.code = assign.code; }*

assign -> id = E; *{assign.code = E.code*
|| gen(id.lexeme '=' E.addr);}

| **L = E;** *{assign.code = E.code || L.code*
|| gen(L.addr.base '[' L. addr '] '=' E. addr);}

E -> E_1 **arith** E_2 *{E.addr=newtemp("e_");*
E.code = E_1*.code ||* E_2*.code*
|| gen(E. addr '=' E_1.addr arith.op E_2.addr);
}

| **id** *{E.addr = id.lexeme; E.code = "";}*

| **num** *{E.addr = num.value; E.code = "";}*

| **L** *{E.addr = newtemp("a_");*
E.code = L.code
|| gen(E.addr '=' L.array.base '[' L.addr ']');}

L -> id [E] *{L.array = top.get(id.lexeme);*
L.type = L.array.type. elem;
L. addr = newtemp("i_");
L.code = E.code
|| gen(L.addr '=' E.addr '' L.type.width); }*

| L_1 **[E]** *{ L.array =* L_1 *.array;*
L.type = L_1*.type.elem;*
t = newtemp("t_") ; L. addr = newtemp("i_");
L.code = E.code || L_1*.code*
|| gen(t '=' E.addr '' L.type.width); }*
|| gen(L. addr '=' L_1*. addr '+' t);}*

S -> D

S -> *{*S_1*.next = newlabel("s_");* S_2*.next = S.next;}* S_1S_2
{S.code = S_1*.code || label(*S_1*.next) ||* S_2*.code; }*

S -> if *{B.true = newlabel("bt_"); B.false=*S_1*.next=S.next;}*
(B) {S_1**}** *{S.code = B.code || Label(B.true) ||* S_1*.code;}*

S -> *{B.true = newlabel("bt_"); B.false=S.next;*
S_1*.next = newlabel("lo_");}*
while (B) {S_1**}** *{S.code = label(*S_1*.next)*
|| B.code || label(B.true)
|| S_1*.code || gen('goto' S1.next);}*

S -> *{B.true = newlabel("bt_");*
B.false= newlabel("bf_");
S_1*.next =* S_2*.next = S.next;}*
if (B) {S_1**} else {**S_2**}**
{S.code = B.code || label(B.true) || S_1*.code*
|| gen('goto' S.next) || label(B.false) || S_2*.code;}*

B -> E_1 **rel** E_2 *{B.code =* E_1*.code ||* E_2*.code*
|| gen('if' E_1*.addr rel.op* E_2*.addr 'goto' B.true)*
|| gen('goto' B.false);}

B -> *{*B_1*.true = B.false;* B_1*.false=B.true;}*
!B_1 *{B.code =*B_1*.code;}*

B -> *{*B_1*.true = B.true;* B_1*.false = newlabel("bf_");*
B_2*.true = B.true;* B_2*.false = B.false;}*
B_1 **||** B_2 *{ B.code =* B_1*.code*
|| label(B_1.false) *||* B_2*.code;}*

D -> T id *{symbol = new Symbol('name': id.lexeme);*
symbol.type = T.type;
symbol.width = T.width; symbol.offset = off;
top.insert(id.lexeme, symbol);
off = off + T.width; } **; D**

D -> ε

T -> B *{C.bT = B.type; C.bW=B.width;}*
C *{T.type = C.type; T.width = C.width}*

B -> int *{B.type = int; B.width = 4;}*

B -> float *{B.type = float; B.width = 8;}*

C -> [**num**] *{C_1.bT = C.bT; C_1.bW=C.bW;}*
C_1 *{C.type = array(num.value, C_1.type);*
*C.width = C_1.width * num.value;}*

C -> ε *{C.type = C.bT; C.width = C.bW; }*

T- > record *{tabStack.push(top); offStack.push(off);*
top = new SymbolTable(); off = 0;}
{D} *{T.type = top; T.width = off;*
off = offStack.pop(); top = tabStack.pop();}

B -> *{B_1.true = newlabel("bt_"); B_1.false = B.false;*
B_2.true = B.true; B_2.false = B.false; }
B_1 && B_2 *{B.code = B_1.code*
|| label(B_1.false) || B_2.code;}

B -> *{B_1.true = B.true; B_1.false = B.false; }*
(B_1) *{B.code = B_1.code;}*

Note that || means string concatenation in SDT rules

```
func newlabel(prefix) {
    label = prefix || counter;
    counter++;
    return label;
}
func newtemp(prefix) {
    temp = prefix || counter;
    counter++;
    return temp;
}
func label(label) { return label || ":"; }
func gen(params[]) {
   str = "";
   for (param in params) str = str || param
  return str;
}
```

1.1 Using the grammar given above generate the parse tree for the following declaration sequence *(3 points)*, annotate the parse tree nodes with proper values of off variable in different D nodes *(2 points)*, proper values of type and width for different B, C, and T nodes (for brevity, you can use t for Type and w for indicating width and write t=sTable for record type variables to avoid cluttering your diagram) *(3 points)*, and show the content of various symbol tables *(2 points)*. 10

```
int x;
float[5] k;
record {
   int a;
   record {
      float c;
      int b;
    } w;
} y;
int i;
```

1.2 Using the variable declarations of Question I, generate the three address code for the following C code snippet by first drawing the parse tree, annotating each B and S nodes of tree with proper values of label attributes, annotating the E nodes with proper values of address attribute *(2 points)* (note that the labels and addresses are generated in the same tree traversal), then finally write the three address codes for B, S, and P nodes using those labels and addresses *(5 points)*. Be careful, your generated code and attributes must agree with the SDT rules given.
 	
`while (i > x + 10 && x != 0) {`
`  x = k[i * 2];`
`  i = i + 1;`
`}`
`k[i - x] = i * x;`
	12

2.*(CO5)* Draw the content of the activation record of a function in the runtime stack *(2.5 points)* and explain how nested function calls are handled using the stack*(2.5 points)*.	5
3.*(CO5)* Generate the three address code *(1.5 points)* and quadruple sequence *(1.5 points)* for the below code. `x = bar(r, k * l + 3) + b;`	3