LAB REPORT-4

Subtopic: Intermediate Code Generation

A) Write SDT for generating proper Three-Address Codes for your grammar.

B) Apply backpatching to give the destination address of jump statements correctly.

C) Show that for any types of arbitrary blocks of codes containing all types of statements (expressions, control flow, relational operator, array, function call, etc.) written in your programming language, your compiler can generate the appropriate three address codes.

D) Draw the annotated parse trees with the semantic translations and DAGs for the sample programs of C).

A) Write syntax directed translations for generating proper Three-Address Codes for your grammar.

```
void makeQ(char *R, char *A1, char *A2, char *Op)
{

    allQ[qIndex].R = (char*)malloc(strlen(R)+1);
    allQ[qIndex].Op = (char*)malloc(strlen(Op)+1);
    allQ[qIndex].A1 = (char*)malloc(strlen(A1)+1);
    allQ[qIndex].A2 = (char*)malloc(strlen(A2)+1);

    strcpy(allQ[qIndex].R, R);
    strcpy(allQ[qIndex].A1, A1);
    strcpy(allQ[qIndex].A2, A2);
    strcpy(allQ[qIndex].Op, Op);
    allQ[qIndex].I = qIndex;

    qIndex++;

    return;
}
```

The **makeQ** function plays a crucial role in constructing quadruples for intermediate code generation. It dynamically allocates memory for each component (R, A1, A2, Op) and assigns provided values, ensuring coherence and sequentially in quadruple generation. This pivotal functionality facilitates subsequent optimization and translation stages in the compilation process.

```
char *makeStr(int no, int flag)
{
    char A[10];
    Xitoa(no, A);

    if(flag==1)
    {
            strcpy(tString, "T");
            strcat(tString, A);
            insertRecord("ICGTempVar", tString, -1, 0);
            return tString;
    }
    else
    {
            strcpy(lString, "L");
            strcat(lString, A);
            insertRecord("ICGTempLabel", lString, -1, 0);
            return lString;
    }

}
```

The makeStr function is crucial in code generation tasks, particularly in creating unique identifiers for temporary variables and labels. When generating code for complex expressions or control flow structures like loops and conditionals, temporary variables and labels are often needed. These identifiers ensure that each variable or label is distinct within the generated code. The function dynamically constructs a string representation by accepting an integer input (no) and a flag (flag). When the flag is set to 1, indicating the need for a temporary variable, the function prefixes the string with 'T'. Conversely, when the flag is 0, it prefixes the string with 'L' to signify a label. For example, if no is 3 and the flag is 1, the function would return "T3". If the flag is 0 instead, it would return "L3". These generated strings can then be used as identifiers within the generated code, ensuring uniqueness and clarity in the resulting codebase.

```
void codeGenOp(node *opNode)
{
    if(opNode == NULL)
    {
        return;
    }

    if(opNode->NType == NULL)
    {
        if((!strcmp(opNode->id->type, "Identifier")) || (!strcmp(opNode->id->type, "Constant")))
        {
            printf("T%d = %s\n", opNode->nodeNo, opNode->id->name);
            makeQ(makeStr(opNode->nodeNo, 1), opNode->id->name, "-", "=");
        }
        return;
    }
```

```
if((!strcmp(opNode->NType, "If")) || (!strcmp(opNode->NType, "Elif")))
{
    switch(opNode->noOps)
    {
        case 2 :
        {
            int temp = lIndex;
            codeGenOp(opNode->NextLevel[0]);
            printf("If False T%d goto L%d\n", opNode->NextLevel[0]->nodeNo, lIndex);
            makeQ(makeStr(temp, 0), makeStr(opNode->NextLevel[0]->nodeNo, 1), "-", "If False");
            lIndex++;
            codeGenOp(opNode->NextLevel[1]);
            lIndex--;
            printf("L%d: ", temp);
            makeQ(makeStr(temp, 0), "-", "-", "Label");
            break;
        }
        case 3 :
        {
            int temp = lIndex;
            codeGenOp(opNode->NextLevel[0]);
            printf("If False T%d goto L%d\n", opNode->NextLevel[0]->nodeNo, lIndex);
            makeQ(makeStr(temp, 0), makeStr(opNode->NextLevel[0]->nodeNo, 1), "-", "If False");
            codeGenOp(opNode->NextLevel[1]);
            printf("goto L%d\n", temp+1);
            makeQ(makeStr(temp+1, 0), "-", "-", "goto");
            printf("L%d: ", temp);
            makeQ(makeStr(temp, 0), "-", "-", "Label");
            codeGenOp(opNode->NextLevel[2]);
            printf("L%d: ", temp+1);
            makeQ(makeStr(temp+1, 0), "-", "-", "Label");
            lIndex+=2;
            break;
        }
    }
    return;
}
```

The **codeGenOp** function is central to the code generation process, traversing the abstract syntax tree (AST) and producing intermediate code based on node properties. It generates assignments for identifiers or constants, constructs quadruples for assignments, and handles conditional statements by generating code for condition evaluation, branching, and label creation. This facilitates accurate translation of high-level conditionals into executable intermediate code. It handles various control flow structures: for "If" statements, it recursively generates code for the "If-Elif" block.

```
if(!strcmp(opNode->NType, "Else"))
{
    codeGenOp(opNode->NextLevel[0]);
    return;
}

if(!strcmp(opNode->NType, "While"))
{
    int temp = lIndex;
    codeGenOp(opNode->NextLevel[0]);
    printf("L%d: If False T%d goto L%d\n", lIndex, opNode->NextLevel[0]->nodeNo, lIndex+1);
    makeQ(makeStr(temp, 0), "-", "-", "Label");
    makeQ(makeStr(temp+1, 0), makeStr(opNode->NextLevel[0]->nodeNo, 1), "-", "If False");
    lIndex+=2;
    codeGenOp(opNode->NextLevel[1]);
    printf("goto L%d\n", temp);
    makeQ(makeStr(temp, 0), "-", "-", "goto");
    printf("L%d: ", temp+1);
    makeQ(makeStr(temp+1, 0), "-", "-", "Label");
    lIndex = lIndex+2;
    return;
}

if(!strcmp(opNode->NType, "Next"))
{
    codeGenOp(opNode->NextLevel[0]);
    codeGenOp(opNode->NextLevel[1]);
    return;
}

if(!strcmp(opNode->NType, "BeginBlock"))
{
    codeGenOp(opNode->NextLevel[0]);
    codeGenOp(opNode->NextLevel[1]);
    return;
}

if(!strcmp(opNode->NType, "EndBlock"))
{
    switch(opNode->noOps)
    {
        case 0 :
        {
            break;
        }
        case 1 :
        {
            codeGenOp(opNode->NextLevel[0]);
            break;
        }
    }
    return;
}
```

It handles various control flow structures: for "Else" statements, it recursively generates code for the "Else" block; for "While" loops, it generates code for loop initiation, condition evaluation, and looping; for "Next" and "BeginBlock" statements, it generates code for their contents recursively. Finally, for "EndBlock" statements, it handles different cases based on the number of operands, generating code recursively for nested blocks.

```
if(!strcmp(opNode->NType, "Func_Name"))
{
    printf("Begin Function %s\n", opNode->NextLevel[0]->id->name);
    makeQ("-", opNode->NextLevel[0]->id->name, "-", "BeginF");
    codeGenOp(opNode->NextLevel[2]);
    printf("End Function %s\n", opNode->NextLevel[0]->id->name);
    makeQ("-", opNode->NextLevel[0]->id->name, "-", "EndF");
    return;
}

if(!strcmp(opNode->NType, "Func_Call"))
{
    if(!strcmp(opNode->NextLevel[1]->NType, "Void"))
    {
        printf("(T%d)Call Function %s\n", opNode->nodeNo, opNode->NextLevel[0]->id->name);
        makeQ(makeStr(opNode->nodeNo, 1), opNode->NextLevel[0]->id->name, "-", "Call");
    }
    else
    {
        char A[10];
        char* token = strtok(opNode->NextLevel[1]->NType, ",");
    int i = 0;
        while (token != NULL)
        {
                i++;
            printf("Push Param %s\n", token);
            makeQ("-", token, "-", "Param");
            token = strtok(NULL, ",");
        }

        printf("(T%d)Call Function %s, %d\n", opNode->nodeNo, opNode->NextLevel[0]->id->name, i);
        sprintf(A, "%d", i);
        makeQ(makeStr(opNode->nodeNo, 1), opNode->NextLevel[0]->id->name, A, "Call");
        printf("Pop Params for Function %s, %d\n", opNode->NextLevel[0]->id->name, i);

        return;
    }
}
```

The code recursively generates intermediate code for expressions or statements separated by newline nodes. For assignment nodes, it creates quadruples to assign the right-hand side value to the left-hand side variable. When encountering a function name node, it prints "Begin Function" followed by the function name, processes the function's body recursively, and prints "End Function", creating corresponding quadruples. For function call nodes, it handles parameter passing, prints statements indicating parameter count and the function call, and creates appropriate quadruples.

B) Apply the concept of backpatching to give the destination address of jump statements correctly.

```
typedef struct Patch {
    int quadrupleIndex;
    int targetAddress;
} Patch;

Patch* pendingPatches = NULL;
int patchIndex = 0;

typedef struct List {
    int *array;
    int size;
    int capacity;
} List;

List trueList, falseList;

void initializeList(List *list) {
    list->array = malloc(10 * sizeof(int)); // initial capacity
    list->size = 0;
    list->capacity = 10;
}

void addToTrueList(int quadIndex) {
    if (trueList.size == trueList.capacity) {
        trueList.array = realloc(trueList.array, (trueList.capacity * 2) * sizeof(int));
        trueList.capacity *= 2;
    }
    trueList.array[trueList.size++] = quadIndex;
}

void addToFalseList(int quadIndex) {
    if (falseList.size == falseList.capacity) {
        falseList.array = realloc(falseList.array, (falseList.capacity * 2) * sizeof(int));
        falseList.capacity *= 2;
    }
    falseList.array[falseList.size++] = quadIndex;
}
```

Backpatching is a vital technique in compiler design for managing code generation in the presence of conditional expressions and loops. It operates by maintaining **True Lists and False Lists** during parsing or intermediate code generation, which store the addresses where conditions evaluate to true or false, respectively. As code is generated, placeholders are inserted where jumps or branches are to occur, and target addresses are recorded. During backpatching, these placeholders are replaced with the actual target addresses based on the evaluated conditions, ensuring correct control flow.

```
void merge(List *dest, List *src1, List *src2) {
    dest->size = src1->size + src2->size;
    dest->capacity = dest->size;
    dest->array = realloc(dest->array, dest->size * sizeof(int));
    memcpy(dest->array, src1->array, src1->size * sizeof(int));
    memcpy(dest->array + src1->size, src2->array, src2->size * sizeof(int));
}

void addPatch(int quadrupleIndex, int targetAddress) {
    pendingPatches = realloc(pendingPatches, (patchIndex + 1) * sizeof(Patch));
    pendingPatches[patchIndex].quadrupleIndex = quadrupleIndex;
    pendingPatches[patchIndex].targetAddress = targetAddress;
    patchIndex++;
}
```

```
void performBackpatching() {
    for (int i = 0; i < patchIndex; i++) {
        int quadrupleIndex = pendingPatches[i].quadrupleIndex;
        int targetAddress = pendingPatches[i].targetAddress;
        allQ[quadrupleIndex].targetAddress = targetAddress;
    }
    free(pendingPatches);
    pendingPatches = NULL;
    patchIndex = 0;
}

void codeGenOp(node *opNode) {
    if (!strcmp(opNode->Op, "If False")) {
        if (targetAddressAvailable) {
            allQ[quadrupleIndex].targetAddress = targetAddress;
        } else {
            addPatch(quadrupleIndex, targetAddress);
        }
    }
}
```

Merge is the process of combining True Lists or False Lists from different expressions or code blocks. For example, when encountering logical operators like AND or OR, the True Lists or False Lists of the operands need to be merged appropriately to get the correct result. initializeList initializes a list with an initial capacity of 10. **addToTrueList** and **addToFalseList** add a quadruple index to the true and false lists, respectively, dynamically resizing the array if needed. **merge** merges two lists into a single list. These functions are used to manage true and false lists, which are essential for handling conditional expressions during code generation.

C) Show that for any types of arbitrary blocks of codes containing all types of statements (expressions, control flow, relational operator, array, function call, etc.) written in your programming language, your compiler is able to generate the appropriate three address codes.

Input:

```
import math

x=1.543
y=-2.7
z=3
i=(x+10)*2.5-y/z*x

def f1(x,y,i):
        while(x<y):
                x=y
                y=x
                if(i=='+'):
                        x+y
                elif(i=='-'):
                        x-y
                elif(i=='-'):
                        x-y
```

```
        elif(i=='-'):
                x-y
        else:
                x*y
        return

x1 = f1(x,10,'+')
x2 = f1(x1,2.5,'*')
x3 = f1(z,x,'*')
x4 = f1(y,x3,'/')

i = f1(x2, x4, '-')
```

3 address code:

```
import math
T2 = 1.543
x = T2
T5 = 2.7
T6 = - T5
y = T6
T9 = 3
z = T9
T12 = x
T13 = 10
T14 = T12 + T13
T15 = 2.5
T16 = T14 * T15
T17 = y
T18 = z
T19 = T17 / T18
T20 = x
T21 = T19 * T20
T22 = T16 - T21
i = T22
Begin Function f1
T26 = x
T27 = y
T28 = T26 < T27
L0: If False T28 goto L1
T29 = y
x = T29
T32 = x
y = T32
T35 = i
T36 = '+'
T37 = T35 == T36
If False T37 goto L2
T38 = x
T39 = y
```

```
T40 = T38 + T39
goto L3
L2: T43 = i
T44 = '-'
T45 = T43 == T44
If False T45 goto L2
T46 = x
T47 = y
T48 = T46 - T47
goto L3
L2: T51 = i
T52 = '-'
T53 = T51 == T52
If False T53 goto L2
T54 = x
T55 = y
T56 = T54 - T55
goto L3
L2: T59 = i
T60 = '-'
T61 = T59 == T60
If False T61 goto L2
T62 = x
T63 = y
T64 = T62 - T63
goto L3
L2: T67 = x
T68 = y
T69 = T67 * T68
return
L3: L3: L3: L3: goto L0
L1: End Function f1
Push Param x
Push Param  x
Push Param  10
(T91)Call Function f1, 3
Pop Params for Function f1, 3
x1 = T91
Push Param x1
Push Param  x1
Push Param  2.5
(T98)Call Function f1, 3
Pop Params for Function f1, 3
x2 = T98
Push Param z
Push Param  z
Push Param  x
(T105)Call Function f1, 3
Pop Params for Function f1, 3
x3 = T105
```

```
Push Param y
Push Param  y
Push Param  x3
(T112)Call Function f1, 3
Pop Params for Function f1, 3
x4 = T112
Push Param x2
Push Param  x2
Push Param  x4
(T119)Call Function f1, 3
Pop Params for Function f1, 3
i = T119


--------------------------------All Quads--------------------------------
I       OP      A1      A2      R
0       import math     -       -
1       =       1.543   -       T2
2       =       T2      -       x
3       =       2.7     -       T5
4       -       T5      -       T6
5       =       T6      -       y
6       =       3       -       T9
7       =       T9      -       z
8       =       x       -       T12
9       =       10      -       T13
10      +       T12     T13     T14
11      =       2.5     -       T15
12      *       T14     T15     T16
13      =       y       -       T17
14      =       z       -       T18
15      /       T17     T18     T19
16      =       x       -       T20
17      *       T19     T20     T21
18      -       T16     T21     T22
19      =       T22     -       i
20      BeginF  f1      -       -
21      =       x       -       T26
22      =       y       -       T27
23      <       T26     T27     T28
24      Label   -       -       L0
25      If False        T28     -       L1
26      =       y       -       T29
27      =       T29     -       x
28      =       x       -       T32
29      =       T32     -       y
30      =       i       -       T35
31      =       '+'     -       T36
32      ==      T35     T36     T37
33      If False        T37     -       L2
```

34	=	x	-	T38
35	=	y	-	T39
36	+	T38	T39	T40
37	goto	-	-	L3
38	Label	-	-	L2
39	=	i	-	T43
40	=	'-'	-	T44
41	==	T43	T44	T45
42	If False	T45	-	L2
43	=	x	-	T46
44	=	y	-	T47
45	-	T46	T47	T48
46	goto	-	-	L3
47	Label	-	-	L2
48	=	i	-	T51
49	=	'-'	-	T52
50	==	T51	T52	T53
51	If False	T53	-	L2
52	=	x	-	T54
53	=	y	-	T55
54	-	T54	T55	T56
55	goto	-	-	L3
56	Label	-	-	L2
57	=	i	-	T59
58	=	'-'	-	T60
59	==	T59	T60	T61
60	If False	T61	-	L2
61	=	x	-	T62
62	=	y	-	T63
63	-	T62	T63	T64
64	goto	-	-	L3
65	Label	-	-	L2
66	=	x	-	T67
67	=	y	-	T68
68	*	T67	T68	T69
69	return	-	-	-
70	Label	-	-	L3
71	Label	-	-	L3
72	Label	-	-	L3
73	Label	-	-	L3
74	goto	-	-	L0
75	Label	-	-	L1
76	EndF	f1	-	-
77	Param	x	-	-
78	Param	x	-	-
79	Param	10	-	-
80	Call	f1	3	T91
81	=	T91	-	x1
82	Param	x1	-	-
83	Param	x1	-	-

```
84      Param  2.5    -       -
85      Call   f1     3       T98
86      =      T98    -       x2
87      Param  z      -       -
88      Param  z      -       -
89      Param  x      -       -
90      Call   f1     3       T105
91      =      T105   -       x3
92      Param  y      -       -
93      Param  y      -       -
94      Param  x3     -       -
95      Call   f1     3       T112
96      =      T112   -       x4
97      Param  x2     -       -
98      Param  x2     -       -
99      Param  x4     -       -
100     Call   f1     3       T119
101     =      T119   -       i
--------------------------------------------------------------------------
```