CS3300 - A2 CS21B040

1) At the entry of foo:

Actual Parameters	{x, len, 0}
Returned Values	
Control link	caller
Local variables	x[ ] = {2,3,4}, len = 3

At the entry of sum from foo:

Actual Parameters	{x+1, len-1, sum+x[0]}
Returned Values	
Control Link	(foo)
Local variables	i, v

At the entry of sum from sum in the last recursive call)

Actual Parameters	{ }
Returned Values	sum = 9
Control Link	(sum)
Local Variables	i, v

At the exit of foo's

Actual Parameters	{x, len, 0}
Returned Values	sum(x, len, 0) = 9
Control Link	caller
Local Variables	x[ ] = {2,3,4}, len = 3

2)

$P \rightarrow S$	S.next = new Label(); P.code = S.code || label(S.next)
$S \rightarrow$ whileStmt	whileStmt.next = S.next S.code = whileStmt.code
$S \rightarrow$ Assignment	S.code = Assignment.code
$S \rightarrow S_1 ; S_2$	S_1.next = new Label(); S_2.next = S.next; S.code = S_1.code || ";" || label(S_1.next) || S_2.code
$S \rightarrow \varepsilon$	S.code = ""
Assignment $\rightarrow x = E;$	Assignment.code = "x" || "=" || E.temp || ";"
$E \rightarrow$ RelEx	E.temp = RelEx.temp
$E \rightarrow$ AddEx	E.temp = AddEx.temp
$E \rightarrow$ Id	E.temp = top.get(Id.lexeme)
RelEx $\rightarrow E_1 < E_2$	RelEx.temp = new Temp(); gen(RelEx.temp "=" E_1.temp "<" E_2.temp)
AddEx $\rightarrow E_1 + E_2$	AddEx.temp = new Temp(); gen(AddEx.temp "=" E_1.temp "+" E_2.temp)

Whilestmt → while (Id) {Stmt2}

begin = new Label();
if_true = new Label();
end = whilestmt.next;
Stmt2.next1 = begin;
Stmt2.next2 = end;
t = new Temp();

whilestmt.code = begin || gen(t "=" top.get(Id.lexeme)) || "if" || t || "goto" || if_true || "goto" || end || label(if_true) || Stmt2.code || "goto" || begin

Stmt2 → S

S.next = Stmt2.next1
Stmt2.code = S.code

Stmt2 → Break

Break.next = Stmt2.next2

Break → break

gen("goto" || Break.next)

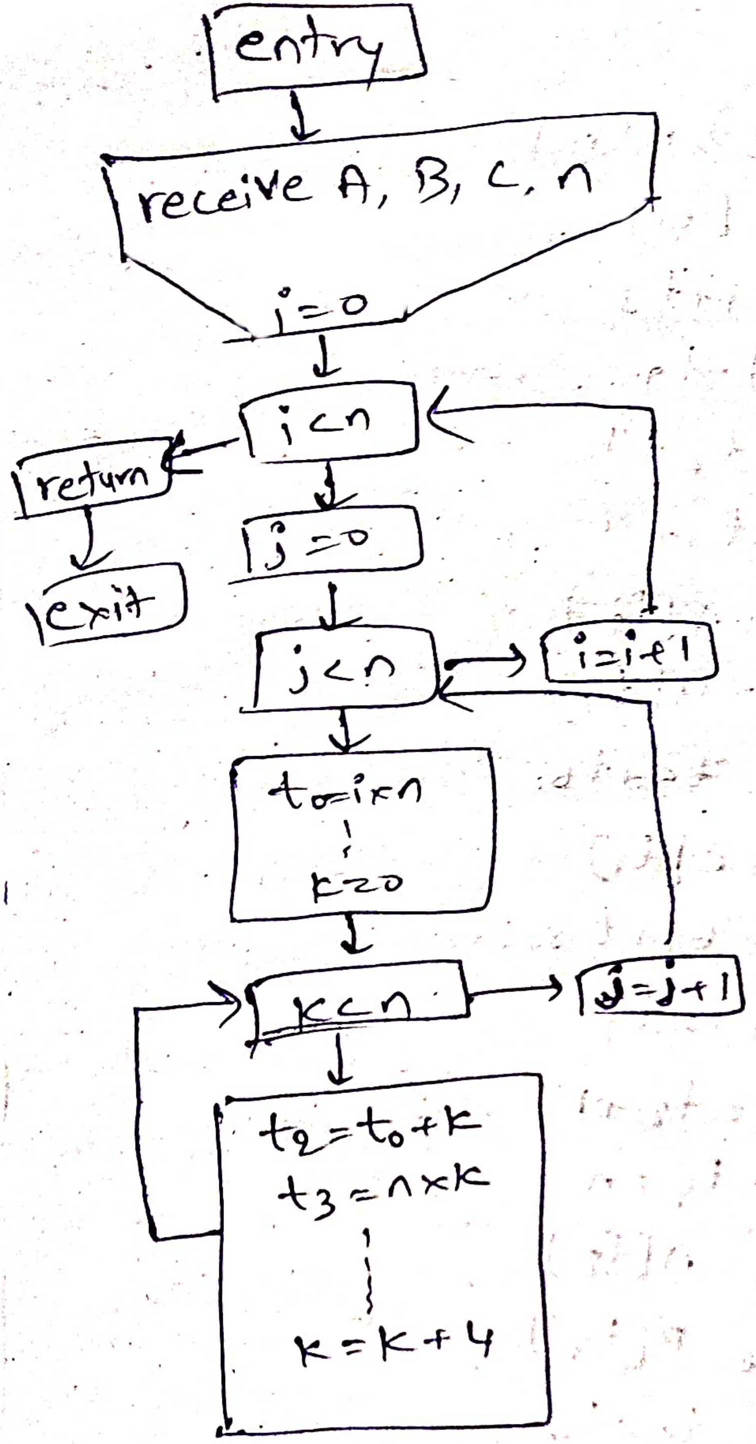
Attributes	Usage
next (Inherited)	Refers to the next line after the piece of code
code (synthesized)	Contains the code (3-address code)
temp (synthesized)	Refers to a temporary assigned to the expression.
lexeme	Value of Identifier
next1, next2 (only for Stmt2)	next1 → refers to begin of while next2 → refers to end of while

gen → generate a piece of code

Label, Temp → To create a new label, temp respectively.

3)

```
receive A (arr), B (arr), C (arr), n (val)
    i = 0
L0: if i < n goto L1
    goto L2
L1: j = 0
L3: if j < n goto L4
    go to L5
L4: t0 = i x n
    t1 = t0 + j
    c[t1] = 0
    k = 0
L6: if k < n goto L7
    go to L8
L7: t2 = t0 + k
    t3 = n x k
    t4 = t3 + j
    t5 = A[t2]
    t6 = B[t4]
    t7 = t5 + t6
    t8 = c[t1]
    t9 = t7 + t8
    c[t1] = t9
    t2 = t2 + 1
    t4 = t4 + n
    t5 = A[t2]
    t6 = B[t4]
    t7 = t5 + t6
    t8 = c[t1]
    t9 = t7 + t8
    c[t1] = t9
    t2 = t2 + 1
    t4 = t4 + n
    t5 = A[t2]
    t6 = B[t4]
    t7 = t5 + t6
    t8 = c[t1]
    t9 = t7 + t8
    c[t1] = t9
    t2 = t2 + 1
    t4 = t4 + n
    t5 = A[t2]
    t6 = B[t4]
    t7 = t5 + t6
    t8 = c[t1]
    t9 = t7 + t8
    C[t1] = t9
    k = k + 4
    goto L6
L8: j = j + 1
    go to L3
L5: i = i + 1
    goto L0
L2: return
```


entry
receive A, B, c, n
i=0
i<n
return
exit
j=0
j<n
i=i+1
t0=i×n
k=0
k<n
j=j+1
t2=t0+k
t3=n×k
k=k+4

4) Given: A variable v is live at a program point L that is used at later point of L.

Aim: In(L) contains v

According to live variable analysis,

$In(n) = use(n) \cup (out(n) - def(n))$ [Let n=L]

From the given information, v is used at a later point of L ⇒ use(n) doesn't contain v.

⇒ $v \notin use(n)$

Let's say v is used at a later point of L at n_1 [First use]

⇒ $In(n_1)$ contains v ⇒ Predecessor of n_1 (n_2) contains v in it's out ⇒ out[n_2] contains v

As def[n_2] doesn't contain v, In(n_2) contains v.

Similarly, by this logic we can say that out[n] contains v.

⇒ $v \in out(n)$.

From the given information, v is live at n ⇒ v is defined in one of it's predecessors.

⇒ def(n) doesn't contain v.

So, use(n), def(n) doesn't contain v but out(n) contains.

From the above mentioned computation of In(n),

∴ In(n) contains v.

∴ If a variable v is live at a program point L that is used at later point of L then $v \in In(L)$

⇒ Live Variable Analysis computes the liveness information for each variable conservatively.

5) Suppose you are designing a compiler for a advanced programming language which includes support for array operations, function calls and loops. Apply peephole optimization techniques to generate improved machine code for the following pieces of code.

i)
```
for(int i=0; i< array.length; i++){
    sum += array[i];
}
```

ii)
```
int sum = add(a, add(b, c))
```

iii)
```
int res = *ptr + 5
```

iv)
```
int compute1(){ // Some code; return r1}
int compute2(){ // Some code; return r2}
void final(int a, int b){ // Some code;}
int a = compute1(); int b = compute2();
final(a, b);
```

Can (iv) be changed to final(compute1(), compute2())? If not why? If yes Justify.

Answers:

i)
```
int len = array.length
for(int i=0; i< len; i++){
    sum += array[i];
}
```

As array's length doesn't change inside the loop.

ii)
```
int t = b + c
int sum = a+t
```

Eliminates overhead of a function call.

iii) int t = *ptr   Eliminates redundant dereferences
int res = t + 5

iv) No, it can't be changed.

Reason: If compute1() and compute2() involve common subcomputations [like incrementing a global value] the improved code doesn't work.