25 April 2017 # CS335: END SEMESTER EXAM **Total Marks: 190**

NOTE:

- Presenting your answers properly is your responsibility. You lose credit if you can not present your ideas clearly, and in proper form. Please DO NOT come back for re-evaluation saying, ”What I actually meant was ....”.
- Be precise and write clearly. Remember that somebody has to read it to evaluate!
- If required, make suitable assumptions and write them along with your answer.

1. .... **(Marks: 4+12+3+3+3 = 25)**

Consider a programming language in which arrays are stored in row-major format, and elements of `float` type take 8 bytes of storage. For the program:

```
float A[1..10, 1..20];
for i from 1 to 10 do
    for j from 1 to 20 do
        A[i,j] = 0.5
    end for
end for
for i from 1 to 10 do
    A[i,i] = 1.0
end for
```

Assume usual semantics for the `for` loop, and that the base of array `A` is stored at address `addr_A`.

(a) Show the translation (address computation) of array element `A[i,j]`.

(b) Give the 3-address code for the program. If you use any 3-address instruction not covered in the class, describe its semantics.

(c) Construct the control-flow graph for the 3-address code. Show each of the following steps clearly:

i) Identify the **leaders**; state the reason as to why is a certain statement identified a leader.

ii) Construct the **basic-blocks**.

iii) Draw the **control-flow edges**.

base address of A correspond to A[1,1], float size is 8.

a) A[i,j] -> addr_A + ((i-1)*20 + j-1)*8 = addr_A + 8 * (20*i + j) - 168
= addr_A + 160*i + 8*j -168

A[i,i] -> addr_A + ((i-1)*20 + i-1)*8 = addr_A + 8 * (20*i + i) - 168
= addr_A + 160*i + 8*j -168 = addr_A + 168*(i - 1)
= addr_A + 168*i - 168

b) Three Address Code (Variations possible, but the basic blocks will remain the same)

```
1. i=1          // Leader - 1st stmt of the program.                  B1
--------------------------------------------------------------------------
2. j = 1        // Leader - target of goto                            B2
--------------------------------------------------------------------------
3. t1 = 20 * i  // Leader targt of goto                               B3
4. t2 = t1 + j
5. t3 = t2 * 8
6. t4 = t3 - 168
7. a[t4] = 0.5
8. j = j + 1
9. if j <= 20 goto (3)  //... j < 20 is also OK
--------------------------------------------------------------------------
10. i = i + 1  // Leader: stmt after goto                             B4
11. if i <= 10 goto (2)  //... i < 10 is also OK

--------------------------------------------------------------------------
12. i = 1// Leader: stmt after goto                                   B5

--------------------------------------------------------------------------
13. t5 = i - 1 // Leader: target of goto                              B6
14. t6 = 168 * t5
15. a[t6] = 1.0
16. i = i + 1
17. if i < 10 goto (13) // ... i < 10 is also OK
```

CFG

$$\begin{tikzcd}
{} \arrow[d] \\
B1 \arrow[d] \\
B2 \arrow[d] \\
B3 \arrow[d] \arrow[loop right] \\
B4 \arrow[d] \arrow[uu, bend right] \\
B5 \arrow[d] \\
B6 \arrow[d] \arrow[loop right] \\
{}
\end{tikzcd}$$

2. .... **(Marks: 20+2+3 = 25)**

Consider the following grammar, that represents a simple program having variable declarations followed by a sequence of assignment statements:

$$\begin{array}{rcl}
P & \rightarrow & L\ S \\
L & \rightarrow & L\ D \mid \epsilon \\
D & \rightarrow & T\ V \\
V & \rightarrow & V\ \text{id} \mid \text{id} \\
S & \rightarrow & S\ A \mid A \\
A & \rightarrow & \text{id} = E \\
E & \rightarrow & E + E \mid E * E \mid (E) \\
 & \mid & \text{id} \mid \text{float-const} \mid \text{int-const} \\
T & \rightarrow & \text{int} \mid \text{float}
\end{array}$$

We want to write a translator to estimate the total power consumption of any program accepted by the above grammar. Assume the following characteristics for the target machine:

- Power consumption for execution for primitive operations on **int** arguments:

Operation	**Description**	**Cost**
store (=)	store value in memory	s
loadV	load a variable's value from memory	v
loadC	load a constant	i
+	addition	a
*	multiplication	m

- For **float** arguments, the power consumption for any operation is 4 times the corresponding consumption for **int** operators.
- If an operation has mixed arguments (int-s and float-s), it will be considered a float operation (e.g. 3 * 4.0 will be float multiplication).
- For assignments, it is an error to assign float value to an int variable on LHS.
- The target machine has an infinite supply of registers, so that a variable need to be loaded only once. Assume write through policy (if a variable is already loaded in register, write will update both the memory location as well as the register).
- The power requirement to read from and write to a register is 0 (negligible).

(a) Create a Syntax Directed Definition (SDD) to compute power consumption for input programs in the language.

(b) What is the cost of the following program with your estimator:

```
float x z
int y
x = (y + 3) * 5.0
z = y + x
x = y * y
```

(c) Mention the type (synthesized/inherited) of every attribute that you use. Is your SDD S-attributed or L-attributed or none? Justify your answer.

(a)

```
P -> L S  { P.pow =  S.pow }
L  -> L D
D -> T V { V.type = T.type}
V -> V1 id  {V1.type = V.type; id.type = V.type}
V -> id {id.type = V.type}
S -> S1 A   {S1.preload = S.preload; A.preload = S1.postload;
              S.pow = S1.pow + A.pow; S.postload = A.postload;}
S -> A {A.preload = S.preload; S.pow = A.pow; S.postload = A.postload; }
T -> int {T.type = int }
T -> float {T.type = float }

A -> id = E {
               E.preload = A.preload;
               if id.type == int && E.type == float then ERROR;
               else if (id.type == int) {
                       A.pow = s + E.pow;
               } else { // id.type = float
                       A.pow = 4*s + E.pow;
               }
               A.postload = E.postload;
              }

E -> E1 + E2 { E1.preload = E.preload;
                E2.preload = E1.postload;
                E.postload = E2.postload;
                E.cost = (E1.type == float || E2.type == float)
                          ? 4*a + E1.cost + E2.cost, float)
                          : a +  E1.cost + E2.cost;
                E.type = (E1.type == float || E2.type == float)
                          ? float : int;
               }
E -> E1 * E2 {  E1.preload = E.preload;
                E2.preload = E1.postload;
                E.postload = E2.postload;
                E.cost = (E1.type == float || E2.type == float)
                          ? 4*m + E1.cost + E2.cost, float)
                          : m +  E1.cost + E2.cost;
                E.type = (E1.type == float || E2.type == float)
                          ? float : int;
               }
E-> (E1) {  E1.preload = E.preload;
            E.postload = E1.postload;
            E.cost = E1.cost;
            E.type = E1.type;
          }
E -> id    {E.type = id.type; E.postload = merge(id, E.preload);
            var f = lookup(v, E.preload) ? 0 : v; // reuse!!
            E.cost = id.type==float? 4*f : f;}
E -> float-const {E.type = float;
                   E.cost = 4*i;}
E -> int-const {E.type = int;  E.cost = i;}

// Assuming type attribute of id is set by lexer using some mechanism like symbol
// table. Direct use of symbol table is also OK.
// Reusing constants in SDD is also OK
```

```
(b) S3:   v +  a + 5i + 4m + 4s
    S4: 4v + 4a +            4s
    S5:                  m + 4s
Total:   5v + 5a + 5i + 5m + 12s
```

(c) preload attribute is inherited.
type attribute is inherited
All others synthesized.
L-attributed defn.

3. .... (Marks: 12+8+10 = 30)

Answer the following questions with respect to an intermediate representation consisting of three-address assignments, conditional and unconditional goto statements:

(a) Write the translation scheme for translating the following **Repeat-Until loop** to three-address. You are allowed to introduce non-terminals (like markers) within the productions.

$$S \rightarrow \text{repeat } S_1 \text{ until } E$$

The semantics of repeat-until is as follows:

Step 1.	Execute S_1.
Step 2.	Evaluate E.
Step 3.	If the value of E is **false**, goto Step 1. Otherwise, go to the following statement.

The semantic rules for translating some of the other statements are provided for your aid; your translation scheme must work correctly with these definitions provided below (assume that any assignment that is already in three-address form can be reduced to the non-terminal A):

E $\rightarrow$ id1 relop id2	`E.truelist = makelist(nextquad);` `E.falselist = makelist(nextquad+ 1);` `emit(if id1 relop id2 goto --- );` `emit(goto ---);`
S $\rightarrow$ begin L end	`S.nextlist = L.nextlist;`
S$\rightarrow$A	`S.nextlist = makelist();`
L $\rightarrow$ L$_1$; M S	`backpatch(L`$_1$`.nextlist, M.quad);` `L.nextlist = S.nextlist;`
L$\rightarrow$S	`L.nextlist = S.nextlist;`
M$\rightarrow\epsilon$	`M.quad = nextquad;`

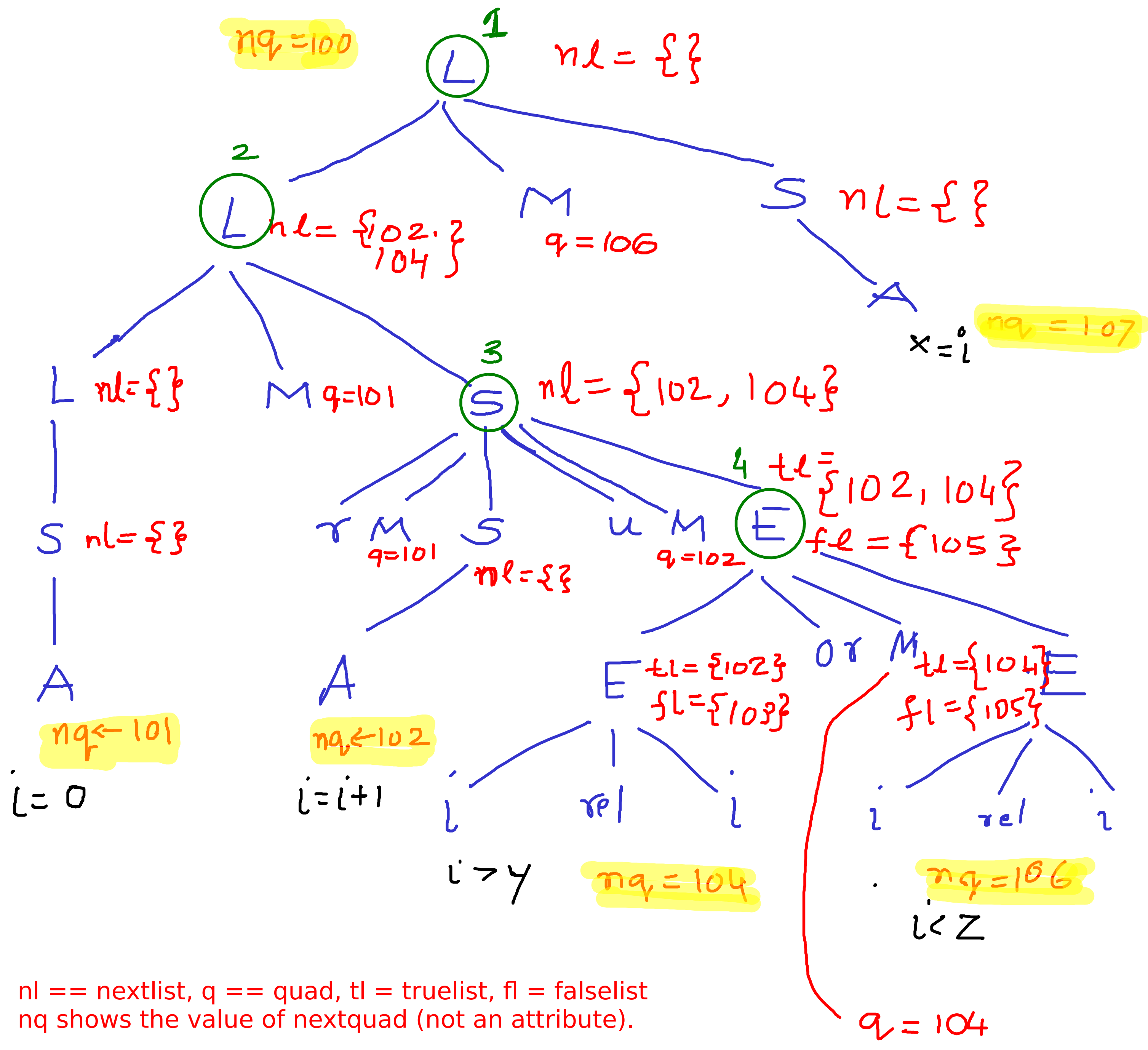
(b) For the following snippet of code, construct the parse tree and annotate it with the value of each attribute. Clearly mark the nodes where backpatching is applied, and clearly write the number of the instruction to be backpatched and the goto-address that is backpatched. **Untidy diagrams will fetch no marks.**

```
i = 0;
repeat
  i = i + 1;
until (i > y or i < z);
x = i;
```

(c) For the above snippet of code, show the current state of the three-address code generated just before each backpatching is applied (for each instance when back-patching happens, you must show the complete code generated **till that point only**).

(a)
```
S -> repeat M1 S1 until M2 E
      { backpatch (S1.nextlist, M2.quad);
        backpatch (E.falselist, M1.quad);
        S.nextlist = E.truelist;
      }
```

(b) assume starting at quad #100.
each A emits exactly one quad, so that nextquad is incremented by 1.



nl == nextlist, q == quad, tl = truelist, fl = falselist
nq shows the value of nextquad (not an attribute).

backpatching applied at green circled nodes, 1,2,3,4.
at node 2, L1.nextlist is empty, so it is OK if you did not mention it.
similarly, one backpatching at node 3 is of emptylist.

node 4 => backpatch({103}, 104)

node 3 => backpatch({}, 102), backpatch({105}, 101)

node 2 => backpatch({}, 101)

node 1 => backpatch({102, 104}, 106)

(c) Empty Backpatches also counted

```
100: i = 0
101: i = i + 1
102: if i > y goto ---
103: goto ---
104: if i < z goto ---
105: goto ---
// before first backpatch applied
```

```
100: i = 0
101: i = i + 1
102: if i > y goto ---
103: goto 104
104: if i < z goto ---
105: goto ---
// before second/third
// backpatch applied
```

```
100: i = 0
101: i = i + 1
102: if i > y goto ---
103: goto 104
104: if i < z goto ---
105: goto 101
// before fourth/fifth
// backpatch applied
```

```
--- OPTIONAL ---
100: i = 0
101: i = i + 1
102: if i > y goto 106
103: goto 104
104: if i < z goto 106
105: goto 101
// after all backpatches applied
// -- rest of the code
106: x = i
```

4. .... **(Marks: 5+10+5+10 = 30)**
Design a variation of the basic Sethi-Ullman algorithm in which the only changes in assumptions are:

- The source language includes the C conditional expression

$$e_1 \; ? \; e_2 \; : \; e_3$$

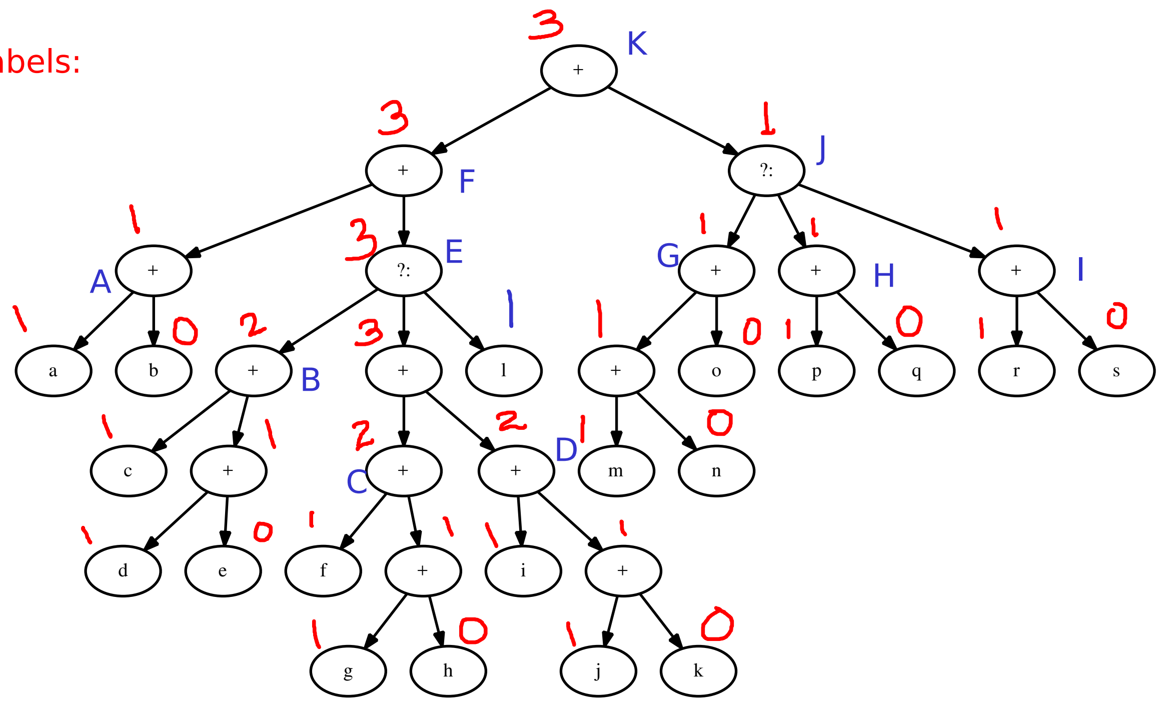
- The machine includes additional instructions:

```
       conditional jumps:   if (R==0) goto L
                            if (R!=0) goto L
     unconditional jump:    goto L
```

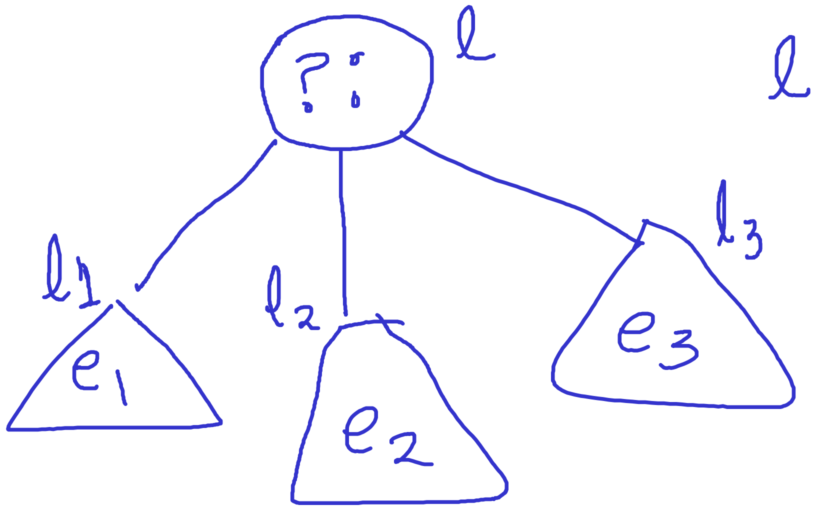
Here `R` is a register name and `L` is the label of some instruction.

(a) Describe the changes in the labeling algorithm.
(b) Describe the changes in the function **gencode**.
(c) For expression tree shown below, show the labels at different nodes. The leaves in the tree are single character names (alphabets a, b, c, ...).
(d) Show the code generated for the expression tree assuming that there are only 2 registers R_0 and R_1. Add comments indicating which code fragments correspond to which sub-expressions, to help correction.

(c) Labels:

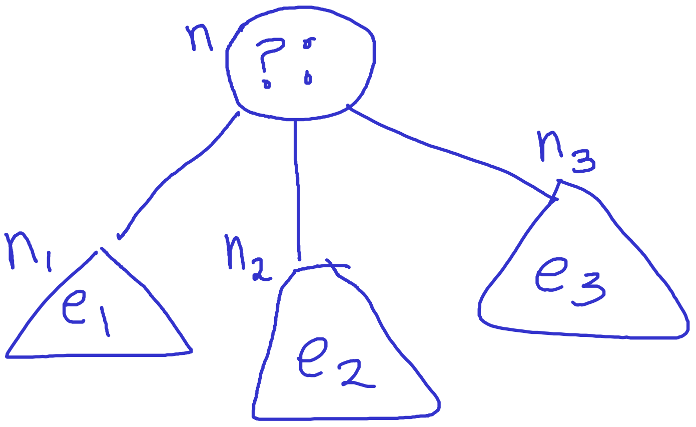


(a) For labelling, we assume the result of ?: node is stored in a register.
To do so, we first have to evaluate COND (e1), store its result in a register (to generate if (R==...) instruction) . Since the branch can be decided and taken before we evaluate e2 OR e3, the register can be reused. Hence,



$$l = \max(l_1, l_2, l_3)$$

(b) gencode(n) where n holds ?: subtree



```
gencode(n1)
L1 = newlabel()
L2 = newlabel()
L3 = newlabel()
gen(if top(rstack) != 0 goto L1)
gen(if(top(rstack) == 0 goto L2)
           // gen(goto L2) is also OK
gen(L1:)
gencode(n2)
gen(goto L3)
gen(L2:)
gencode(n3)
gen(L3:)
```

(d)

```
// gencode(K) -> gencode(F)
//   -- will swap rstack since we evaluate 2nd operand of node F first.
// node B
r1 := c
r0 := d
r0 := r0 + e
r1 := r1 + r0
// node E, cond part
if (r1 != 0) goto L1
if (r1 == 0) goto L2

// node E, true branch
L1:
// node D : note that parent is a major node, so we eval Right branch first
// and leave it in temporary.
r1 := i
r0 := j
r0 := r0 + k
r1 := r1 + r0
t0 := r1
// node C
r1 := f
r0 := g
r0 := r0 + h
r1 := r1 + r0
// + : parent of C and D
r1 := r1 + t0
goto L3 // reqd for ?:, skip false branch

// node E, false branch
L2:
r` := l

L3:
// at this point, result of node E is in r1, and r1 is removed from the stack
// node A,
r0 := a
r0 := r0 + b

// node F
r0 := r0 + r1
```

```
// gencode(K) -> gencode(J), r0 is popped; only available register is r1
// ->gencode(G)
r1 := m
r1 := r1 + n
r1 := r1 + o  // this is 'o' not zero
// node J, cond check
if (r1 != 0) goto L4
if (r1 == 0) goto L5

// true branch, gencode(H)
L4:
r1 := p
r1 := r1 + q
goto L6 // reqd for ?:, skip false branch

// false branch, gencode(I)
L5:
r1 := r // name 'r', not register
r1 := r1 + s

L6:
// result of ?: in r1
// compute + for node K
r0 := r0 + r1
```

IMPORTANT POINTS:

1. code can be slightly different if a different labelling algorithm is used.
2. However, some "stores" will be required as the register requirement will be more than 2; no matter what labelling algo is used.
3. For node F, right child is evaluated first, so stack swap is necessary. Hence, r0 and r1 will change their roles.
4. For subtree rooted at node J, r0 can not be used.

5. ....**(Marks: 5+15+10 = 30)**

Aho-Johnson Algorithm: Consider a machine with two general purpose registers R_0 and R_1, and having the following instructions. The cost of each instruction is also indicated.

1.	$r \leftarrow c$ (load constant)	cost 1
2.	$r \leftarrow m$ (load from memory)	cost 1
3.	$m \leftarrow r$ (store in memory)	cost 1
4.	$r_i \leftarrow r_i \; op \; r_j$	cost 2
5.	$r_i \leftarrow r_i \; op \; m$	cost 3
6.	$r_i \leftarrow ind(r_i)$	cost 1
7.	$r_i \leftarrow ind(r_i + m)$	cost 4

Assume that a program has the following declaration:

```
struct {
    float f;
    int a[5];
    int *p;
} _s;
struct _s s;
int j, k;
```

(a) For the expression

$$\texttt{s.a[j]} + *(\texttt{s.p}) + \texttt{k}$$

Draw the expression tree. Make the usual assumptions about widths of the different types. Addition (+) is left associative.

(b) Annotate each node in the tree with the array of costs.

(c) Generate optimal code for the algorithm.

6. ....**(Marks: (0+2+2+6)+5+5 = 20)**

(a) Consider the grammar:

$$\begin{array}{rcl} s & \rightarrow & a\ S\ a \\ & | & a\ a \end{array}$$

The grammar accepts all even length strings of a-s. Since there are only **two** productions in the grammar, we can devise a **brute-force backtracking**-based parsing technique using the following observation: if we give precedence to the production $S \rightarrow a\ a$, then we can only recognize the string aa. Therefore, we must give precedence to the production $S \rightarrow a\ S\ a$, and try the other production only if this fails. We backtrack to the previous applications of the rules, if both the productions fail.

Show that this parser can parse strings aa and $aaaa$, but can not parse $aaaaaa$.

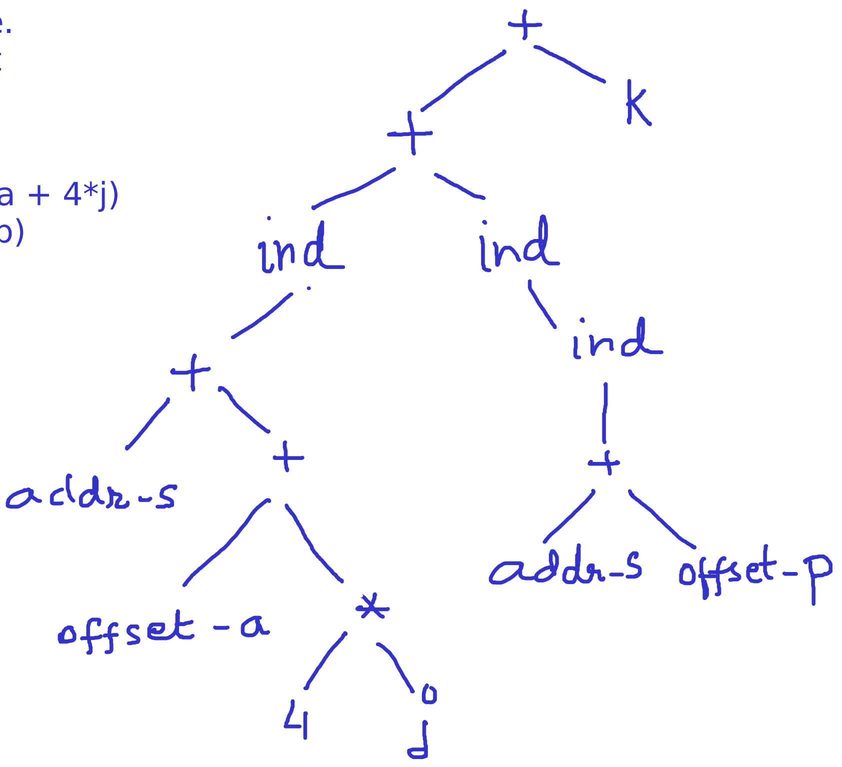
(b) Is the code generated by Sethi-Ullman algorithm *strongly contiguous*? Justify your answer.

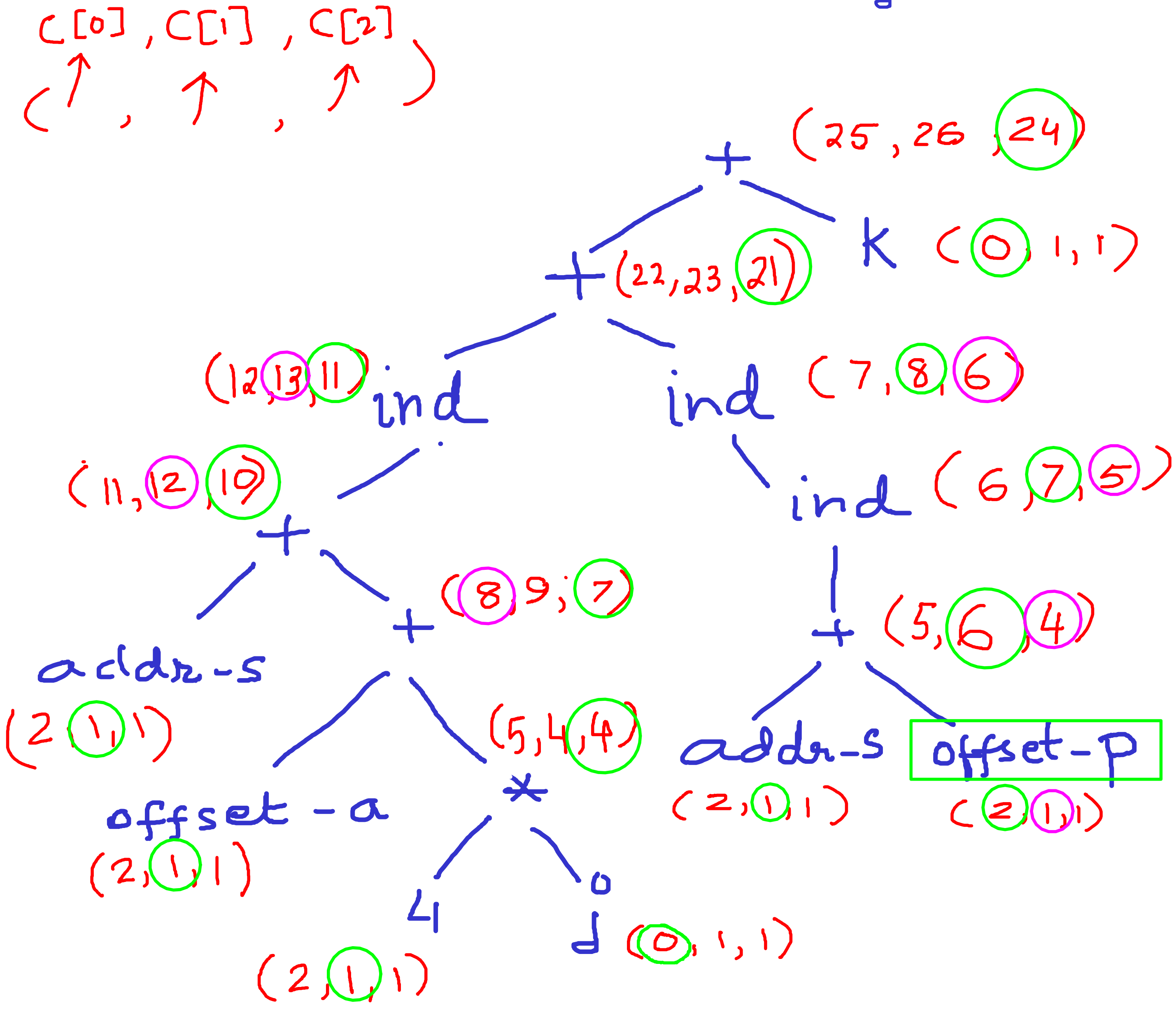
(c) Show the smallest expression (the expression which, represented as a tree, will contain the minimum number of nodes) that will require at least 3 stores to evaluate using basic Sethi-Ullman algorithm, on a machine with two registers.

(5)

Multiple solutions possible.
offset_a = 8 // 8 byte float
offset_p = 28 // 4 byte int

s.a[j] = *(addr_s + offset_a + 4*j)
s.p   = *(addr_s + offset_p)
*(s.p) = *(...)





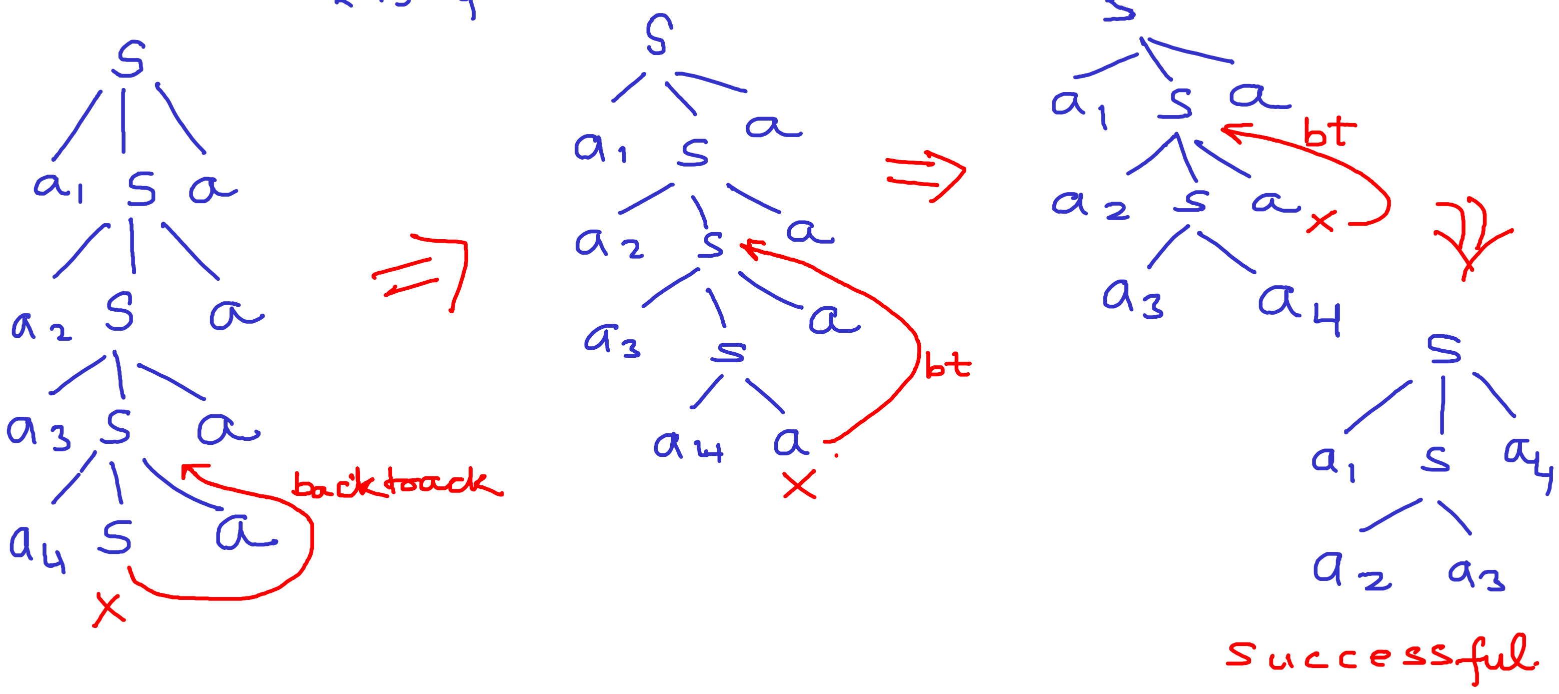
```
// code for purple choices
r0 := j
r1 := 4
r1 := r1 * r0
r0 := 8 // offset_a
r0 := r0 + r1
m0 := r0
r1 := 28 // offset_p
r0 := addr_s
r0 := r0 + r1
r0 := ind(r0)
r0 := ind(r0)
r1 := addr_s
r1 := r1 + m0
r1 := ind(r1)
r1 := r1 + r0
r0 := k
r1 := r1 + r0
```

```
// code for green choices
r0 := 28 // offset_p
m0 := r0
r0 := j
r1 := 4
r1 := r1 * r0
r0 := 8 // offset_a
r0 := r0 + r1
r1 := addr_s
r1 := r1 + r0
r1 := ind(r1)
r0 := addr_s
r0 := r0 + m0
r0 := ind(r0)
r0 := ind(r0)
r1 := r1 + r0
r0 := k
r0 := r1 + r0
```

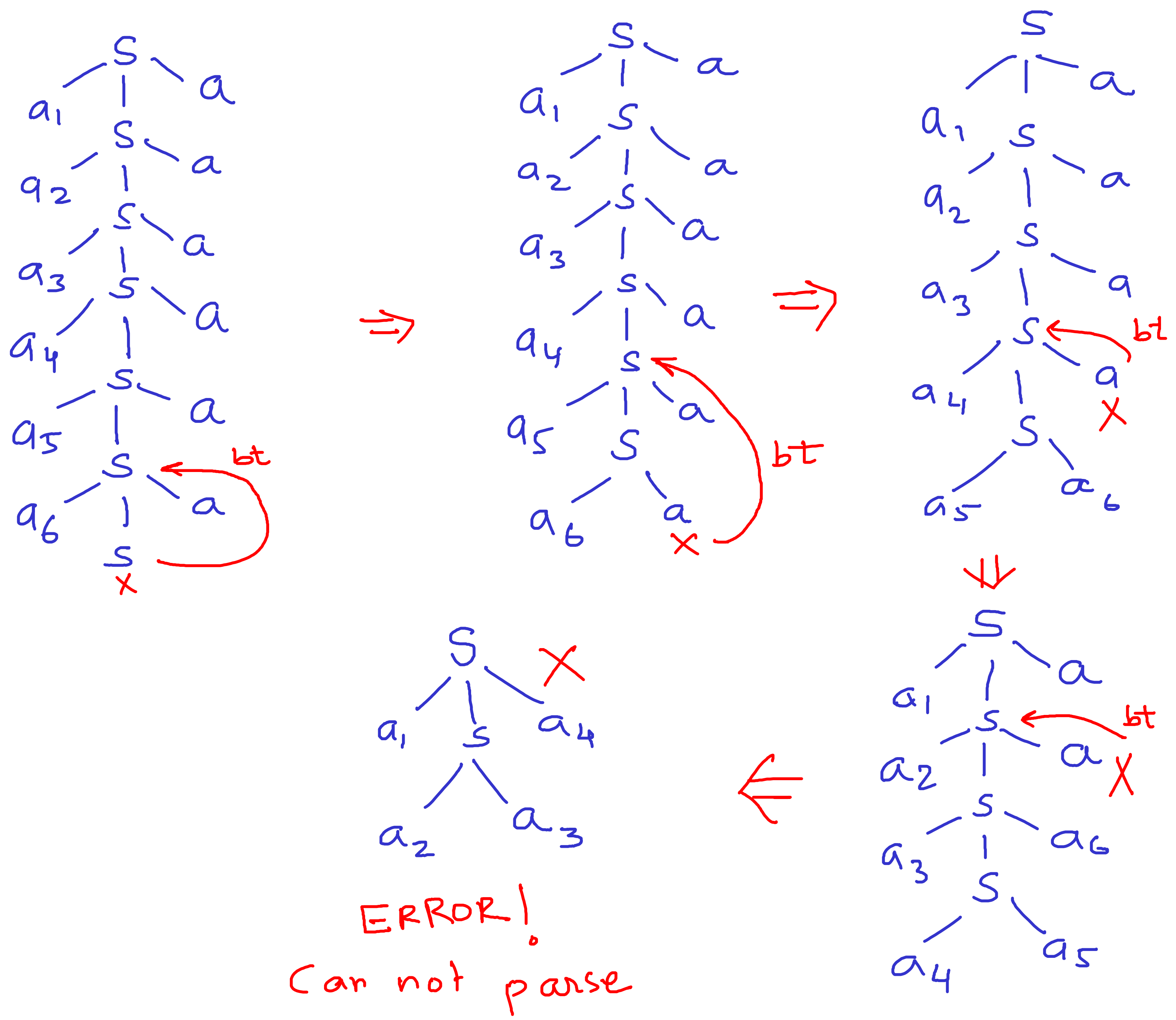
Many combinations will give optimal cost. Green and purple circles show 2 such combinations (wherever there is a choice to break ties). However, there will be exactly one store in phase 3.

6(a)
parsing aa is trivial (no marks for that)

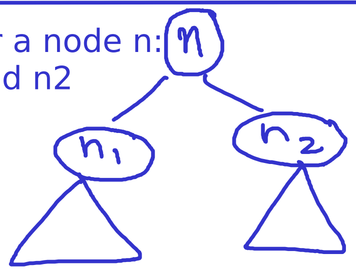
aaaa: $a_1 a_2 a_3 a_4$



$a_1 a_2 a_3 a_4 a_5 a_6$

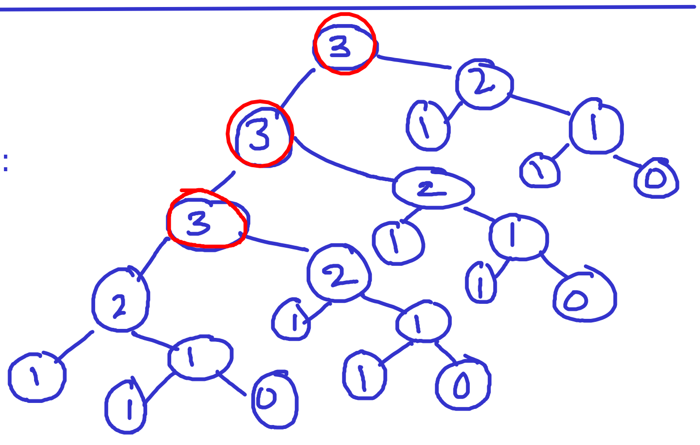


6(b) Code produced by Sethi-Ullman is strongly contiguous. Consider a node n: gencode(n) sequentially invokes gencode on two children n1 and n2 (in any order). Each gencode completely evaluates the subtree rooted at that child node before moving on to the next child. This holds recursively at each level.



6(c) For 3 stores, you need 3 major nodes in the tree. Can you create a tree having 3 major nodes, and having fewer nodes than:

both children having label ≥ 2



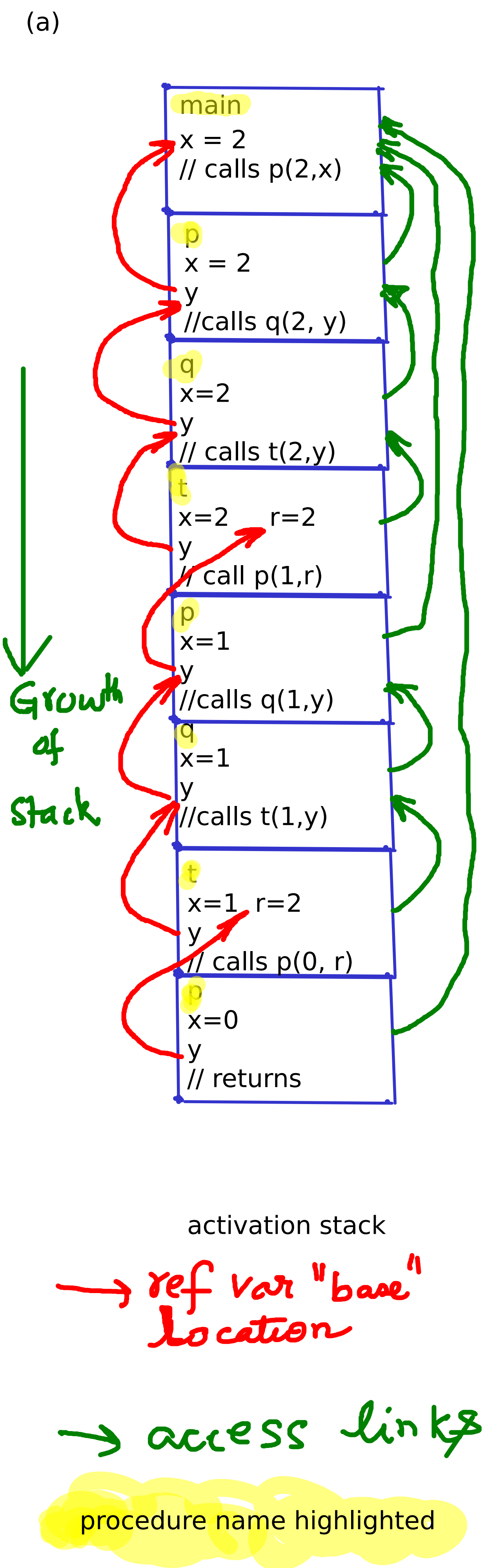
7. ....**(Marks: 8*3+4+2 = 30)**
Consider a programming language having support for nested procedures and static scoping. The language allows parameter passing by value (keyword: `val`) as well as by reference (keyword: `ref`). For the program:

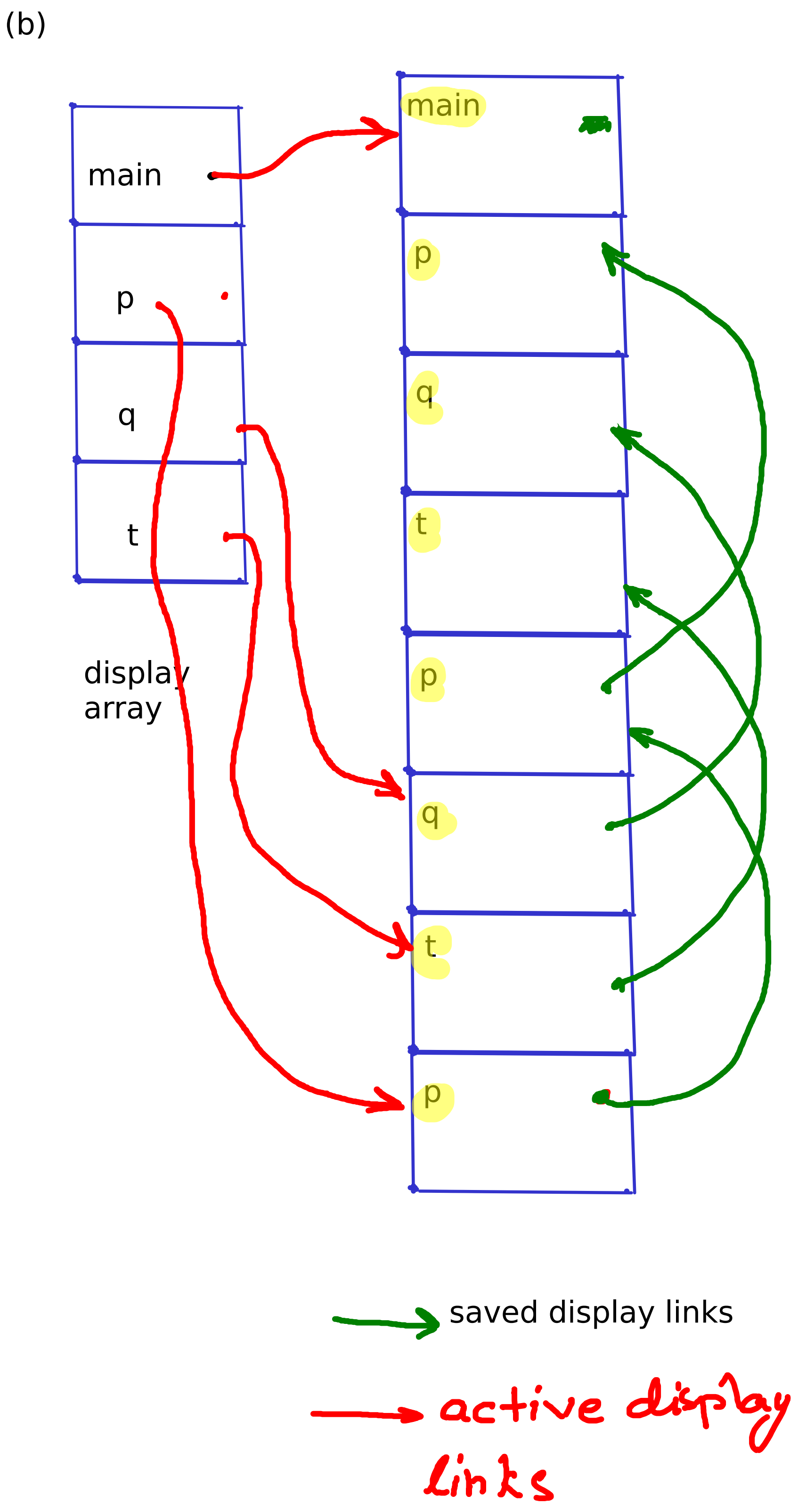
```
procedure main() {
    int x;
    procedure p(val int x, ref int y) {
        procedure q(val int x, ref int y) {
            procedure t(val int x, ref int y) {
                int r;
                r = y;
                p(x-1,r);
            }
            t(x,y);
        }
        if (x==0) then x=1 else q(x, y);
    }
    x=2;
    p(2,x);
}
```

(a) Assuming an implementation using access links, show the runtime stack when it has grown to the maximum possible height. In each activation frame,show (i) local variables, (ii) parameters, and (iii) access link. **Do not** show any other information in the frame.

(b) Repeat (7a) using displays. This time, you only show the saved displays in the activation frames. Do not show local variables or parameters or any other information.

The End **Total Marks: 190**


(a)
main
x = 2
// calls p(2,x)
p
x = 2
y
//calls q(2, y)
q
x=2
y
// calls t(2,y)
t
x=2 r=2
y
// call p(1,r)
p
x=1
y
//calls q(1,y)
q
x=1
y
//calls t(1,y)
t
x=1 r=2
y
// calls p(0, r)
p
x=0
y
// returns
Growth of Stack
activation stack
ref var "base" location
access links
procedure name highlighted


(b)
main
p
q
t
display array
main
p
q
t
p
q
t
p
saved display links
active display links