Problem 1  (40 points)

Consider the following program written in C-like syntax. Assume static scoping.

```c
char a[5] = {'a','b','c','d','e'};
int i=1;

void f(int x) {
    i++;
    x = 'a';
    a[2] = 'd';
    i--;
}

main() {
    for (i=0;i<4;i++)
        f(a[i]);
    printf("'%c', '%c', '%c', '%c', '%c'\n",a[0],a[1],a[2],a[3],a[4]);
}
```

a. (10 points) What is the output if function parameters are passed by value?

'\text{a}', \text{'b'}, \text{'d'}, \text{'d'}, \text{'e'}

b. (10 points) What is the output if function parameters are passed by reference?

'\text{a}', \text{'a'}, \text{'d'}, \text{'a'}, \text{'e'}

c. (10 points) What is the output if function parameters are passed by value-result?

'\text{a}', \text{'a'}, \text{'d'}, \text{'a'}, \text{'e'}

d. (10 points) What is the output if function parameters are passed by name?

'\text{a}', \text{'a'}, \text{'d'}, \text{'a'}, \text{'a'}
Problem 2  (30 points)

a. (10 points) What referencing environment is used for a subprogram passed as parameter when a programming language uses shallow binding? (circle the correct answer)

1. The environment of the call statement that enacts the passed subprogram
2. The environment of the definition of the passed subprogram
3. The environment of the call statement that passed the subprogram as a parameter

b. Consider the following program written in C-like syntax. Assume static scoping.

```c
int x;

void print_int() {
    printf("x = %d\n",x);
}

/* sub2: Takes a subprogram 'subx' as parameter */
void sub2(void(*subx)()) {
    int x = 4;
    subx();
}

void sub1() {
    int x=3;
    sub2(print_int);
}

main() {
    x = 1;
    sub1();
}
```

(i) (10 points) What is the output of the program if shallow binding is used for subprograms passed as parameters?

x = 4

(ii) (10 points) What is the output of the program if deep binding is used for subprograms passed as parameters?

x = 1
Problem 3  (30 points)

a.  (10 points)  Give two disadvantages of dynamic scoping (as opposed to static scoping)

Slower at runtime
Precludes static type checking
Poor readability
Poor reliability

b.  Consider the following program written in C-like syntax. Assume pass-by-value (like in C).

```c
int y = 0;
int x = 0;

void B() {
    x = x + y;
    y = y + 1;
}

void A() {
    int y = 2;
    B();
}

int main() {
    int x = 4;
    A();
    printf("%d,%d\n",x,y);
    B();
    printf("%d,%d\n",x,y);
}
```

(i)  (10 points)  What is the output of the program if static scoping is used?

4 1
4 2

(ii)  (10 points)  What is the output of the program if dynamic scoping is used?

6 0
6 1
Problem 4  (20 points)

What is the result of evaluating the expression

\[ 2 \times 5 - 4 - 3 \times 2 \]

if we assume:

(i) (10 points) that multiplication (\(\times\)) has lower precedence than subtraction (\(-\)), and that all operators are left-associative? (put parentheses in the original expression and give the numerical result)

\[ 2 \times ((5 - 4) - 3) \times 2 = 2 \times (-2) \times 2 = -8 \]

(ii) (10 points) that multiplication (\(\times\)) and subtraction (\(-\)) have the same precedence, and that all operators are right-associative? (put parentheses in the original expression and give the numerical result)

\[ (2 \times (5 - (4 - (3 \times 2)))) = 14 \]

Problem 5  (50 points)

a. (10 points) Give one advantage and one disadvantage of implicit branches at the end of code segments in selection statements.

disadvantage: poor reliability
advantage: allows for one “case” to be a subset of another one

b. (10 points) What are the advantages and disadvantages of allowing the modification of loop variables in the loop body?

disadvantage: poor reliability
advantage: good flexibility (e.g. early exit)

c. (10 points) How would a language system implement loop variables that are not modifiable in the loop body.

with an internal variable

d. (20 points) Consider the following fragment of C code:
Some authors have argued that in this case a goto statement enhances readability! Rewrite this code without using goto (and of course preserving the exact semantics of the original code).

```
int a[100][100];
int i,j;
...
for (i=0;i<100;i++) {
    for (j=0;j<100;j++) {
        if (a[i][j] == 0) goto out;
    }
}
out: printf("done\n");
```

```
int a[100][100];
int i,j;
...
for (i=0;i<100;i++) {
    int exiting = 0;
    for (j=0;j<100;j++) {
        if (a[i][j] == 0) {
            exiting = 1;
            break;
        }
    }
    if (exiting)
        break;
}
out: printf("done\n");
```

Problem 6  (20 points)  (extra credit)

Joe “I partied too hard in TJ” Bob wrote the following Common Lisp function to count recursively the number of elements in a list:

```
(defun recursive-length (l)
  (cond ((atom l) 1)
        (t (+ (recursive-length (car l))
             (recursive-length (cdr l))))))
```
but it doesn’t seem to be working properly. The behavior of recursive-length should be:

\[
\begin{align*}
* \text{ (recursive-length '())} &= 0 \\
* \text{ (recursive-length '(a b))} &= 2 \\
* \text{ (recursive-length '((a b) c))} &= 3 \\
\end{align*}
\]

a. \textit{(10 points)} What is the output of Joe’s function on the list (a b c)?

4

b. \textit{(10 points)} What is wrong with Joe’s code and how would you fix it?

\textbf{Reminder:} Behavior of the Common Lisp predicate functions:

\[
\begin{align*}
\text{(atom () )} &\rightarrow \text{T} \\
\text{(atom 'a)} &\rightarrow \text{T} \\
\text{(atom '(a)) } &\rightarrow \text{NIL} \\
\text{(consp () )} &\rightarrow \text{NIL} \\
\text{(consp 'a)} &\rightarrow \text{NIL} \\
\text{(consp '(a)) } &\rightarrow \text{T} \\
\text{(listp () )} &\rightarrow \text{T} \\
\text{(listp 'a)} &\rightarrow \text{NIL} \\
\text{(listp '(a)) } &\rightarrow \text{T} \\
\text{(null () )} &\rightarrow \text{T} \\
\text{(null 'a)} &\rightarrow \text{NIL} \\
\text{(null '(a)) } &\rightarrow \text{NIL} \\
\end{align*}
\]

\[
\text{(defun recursive-length (l)}
\begin{align*}
&\text{(cond ((null l) 0) } \\
&\text{ ((atom l) 1) } \\
&\text{ (t (+ (recursive-length (car l))} \\
&\text{ (recursive-length (cdr l)))))) \\
\end{align*}
\]