Programming Languages
Homework 2

Question 1: Activation Records and Lifetimes

1) Issue – This question talks about a run time memory allocation issue in C.Consider the function prinf() with **variable parameters**, the compiler would not know the exact location of the function in the runtime stack. Apart from reversing the arguments, there are a few other things that can be done in order to overcome this issue and know exactly **where the Offset lies**. Calling conventions EX -C (cdecl).
    a. One of the ways is to use the method of **heap indirection**. We can store the variables on the heap and give its reference to the stack. If we go ahead and do this, we will have the runtime space information available for that specific variable.
    b. Add a particular boundary case or boundary location suggesting the end-of-input or eof-esque pointer, that will be inserted into the stack frame before any of the other variables or values to be printed by the format string. Now in the case where we push the data in reverse order and start reading it from the top of the stack, we look at the point where the printf () function tries to read more data than was assigned. It would encounter the end-of-input data point in the stack. In that case, the program would automatically throw a run-time error stating that the printf function was trying to access more stack spaces than was allocated.
    c. If we can force the printf to obtain the address from the **format string** (also on the stack), we can control the address.

```
int main(int argc, char *argv[]) {
char user_input[100];
... ... /* other variable definitions and statements */
scanf("%s", user_input); /* getting a string from user */
printf(user_input); /* Vulnerable place */
return 0;
}
```

2) Lets look at these languages separately first and then draw our final conclusion as to why these languages use different allocation techniques.
    - **Fortran** – This uses static based allocation is used for local variables as the storage requirements are known at compile time. This is true for compiled, linked languages, where the compiler can issue specific memory addresses for the code that it generates, including the variables that have been used.
    - **Algol, C and ADA** – The storage requirement of the local variables are not known at the compile time but at the run time. In this case a stack is used just because it uses a similar architecture of last in first out. You can always make the recursive calls, but data will only be allocated once the function is called.

- **Lisp** - This uses the most flexible type of heap allocation, where data/storage can be allocated and deallocated dynamically during program execution. Lisp uses this as it allows us to use an object or variable easily across the program without copying and the fact that heap memory is handled by the program/computer rather than the user itself having to explicitly maintain it

WHY? – Whenever a variable that are allocated on a stack, every entity/function call will have its own copy of a variable. Moreover, it will not reflect the change in the value through the functions will not reflect the same variable. This is very different from the variables that are stored statically. The main reasons behind this is the code flexibility and as responsible coders we need to exactly choose the language that is viable to the production.

3) Whenever there is a C++ object is destroyed the deconstructor for the derived class is called first, followed by the base class in a reverse order. The slide just before the activation record also talk about epilogue where once the value is returned the space needs to be reclaimed (**Offset has found and returned the requested values and now it is time to clear the memory for new calls in the run time allocation stack**).

   As per the question demand, lets analyze both of these cases.

   a. Heap allocated objects, using keywords as new, are only deleted if the code can understand that the object itself is no longer being used. In C++, this must be done by the programmer or user itself, using the delete keyword. A destructor is called when such an object is explicitly deallocated using the keyword delete. Global variables in C++ are also moved to a heap-based storage. In such cases, the global and/or static objects are destroyed when the program itself ends.
   b. In the case of stack-allocated objects, such as local variables, one of the times a destructor is called when the block it is referred or called in goes out of scope, in the sense that the callee block in the stack is removed or popped out.

4) To trace the optimization and space requirement, lets see the code block by block before we make our final conclusion.

   first block
   Number of variables declared = 3 (a, b, c)
   Bytes required = (a, b, c) = 4*3 = 12

   second (inner #1) block,
   Number of variables declared = 4 (a, b, c, d)
   Bytes required = (a, b, c, d) = 4*4 = 16

   third (inner-of-inner #1) block,
   Number of variables declared = 5 (a, b, c, d, e)

Bytes required = (a, b, c, d, e) = 5*4 = 20

fourth (inner-of-inner #2) block,
Number of variables declared = 5 (a, b, c, d, f)
Bytes required = (a, b, c, d, f) = 5*4 = 20

fifth (inner #2) block,
Number of variables declared = 5 (a, b, c, g, h)
Bytes required = (a, b, c, g, h) = 5*4 = 20

sixth (inner-of-inner #3) block,
Number of variables declared = 6 (a, b, c, g, h, i)
Bytes required = (a, b, c, g, h, i) = **6*4 = 24**

Since, ('d','e','f') and ('g','h','i'), then the entities that are stated prior can use the space of the once declared later.
Therefore, a total space of 24 bytes is required in total.

5) Names that are visible from the specified locations are marked below.
A,C in Procedure R[A is hidden from Procedure P]
B (from Procedure P)
X and Z

Question 2: Nested Subprograms

Answers are as follows:

a. The program calling, or subprogram calling hierarchy is:
MAIN -> BIGSUB -> SUB2 -> SUB3 -> SUB1

The variables set in each individual subprogram are as follows:

Subprogram Name	Values Set
MAIN	Declared: • X: 3
BIGSUB	Declared: • A: 2 • B: 3 • C: 4
SUB2	Declared: • B: 1 • E: 8 Used: • X: 7 • A: 8
SUB3	Declared: • C: 9 Used: • B: 1 (from SUB2) • E: 10
SUB1	Declared: • A: 5 • D: 1 Used: • B: 3 (from BIGSUB) • C: 4 (from BIGSUB) • A: 7

SUB1: print (A, B, C): (7, 3, 4)
SUB3: print (E): (10)

Answer

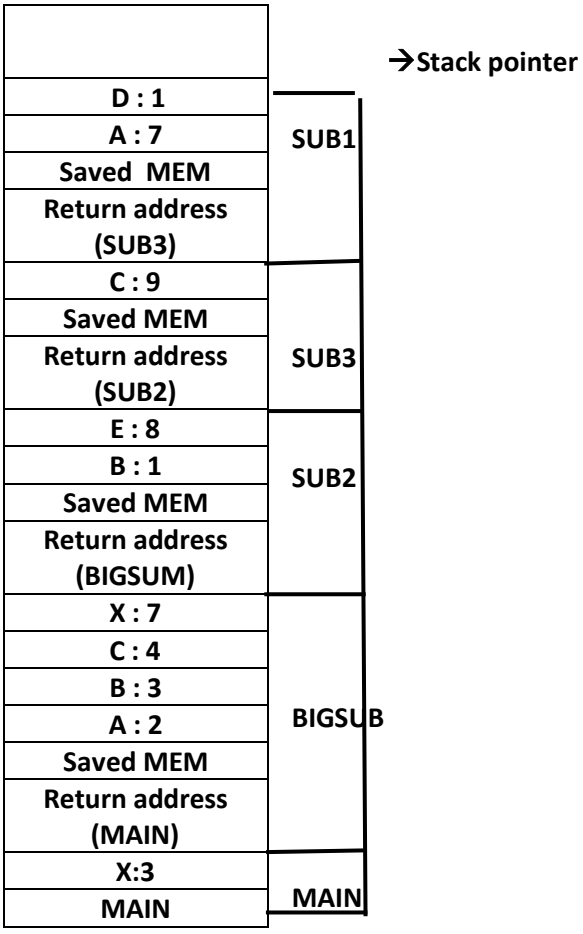
MAIN→BIGSUB→SUB2(7)→SUB3→SUB1→print (7,3,4)→SUB3→print (10)

b. In the scopes and sub-scopes, 2 variables do not change values once they are declared. They are:
   a. Main → (X)
      BIGSUM→ (B,C)
      SUB1 → (D)
      SUB2 → (B,X)
      SUB3 → (C)

      We can see that → Main.X, BIGSUB.B, BIGSUM.C, SUB1.D, SUB2.B, SUB2.X, SUB3.C.

c. No. In this specific example, the behavior of the program will not change even if we go ahead and use Dynamic Scoping. First the parent procedures are called and only after that the child (nested) procedures are called. Therefore, we cannot call a nested procedure first before calling the child procedures. **In conclusion, in order to reach to a child procedure we need to call the parent first.**

d. STACK.



FP – FRAME POINTERS AT EACH LEVEL

SL – STATIC LINKS BETWEEN THE FRAGMENTS

CODE BLOCKS – FOR EACH SUB PROCEDURES ARE MENTIONED TO THE RIGHT

e. No. In the case of static scoping the main method cannot invoke the SUB 3 as it is nested deeper in the code.
Yes, in the other case, SUB3 will be able to invoke the main as it is nested outside of the code and the scope of main is global with respect to SUB 3.

Question 3: Parameter Passing.

3.1
Iteration 1 : a1=2, a2=3, a3=8, a4=22, a5=22

Iteration 2 : a1=25, a2=26, a3=75, a4=252, a5=252

Iteration 3 : a1=278, a2=279, a3=1112, a4=2782, a5=2782

3.2
Iteration 1 : a1=2, a2=2, a3=8, a4=20, a5=20

Iteration 2 : a1=22, a2=2, a3=8, a4=20, a5=20

Iteration 3 : a1=22, a2=2, a3=8, a4=20, a5=20

3.3 Call by Value
Iteration 1: a1=2, a2=2, a3=4, a4=1, a5=12

Iteration 2: a1=3, a2=2, a3=4, a4=2, a5=12

Iteration 3: a1=4, a2=2, a3=4, a4=3, a5=12

Question 4: Lambda calculus

Part 1: Order of abstraction

a) (λx.x) y z
= (((λx.x) y) z)

b) λx.λy.λz.zyx
= (λx. (λy. (λz. (((zy) x))))

c) λx.xyλz.wλw.wxz
= (λx.((xy) (λz. (w (λw.(((wx) z)))))))

d) xλz. xλw.wzy
= (x (λz (x (λw. (((wz) y))))))

e) λz. ((λs. s q) (λq. q z)) λz. z z
= (λz. ((λs . (s q)) (λq. (q z)))) (λz . (z z))

Part 2: Free Variables

a) Free variable: x as x^0
Final expression: (λz. ((z x^0) (λy. (y z))))

b) Free variable: z as x^0
Final expression: ((λx.x) (λx.x (λy.y)) x^0)

c) Free variables: f as x^0, y as x^1, x as x^2
Final expression: (λp. (((λz. (x^0 (λx. (p x^1))) x^2))

d) Free variable: y as y^0 and y as y^1
Final expression: (λx. ((x y^0) (λx. (y^1 x))))

e) Free variable: y as y^0
Final expression: (λx. (x (x y^0)))

Part 3:

a) The lambda calculus equation is: (λxy. zx)(λx. xy) or ((λx. (λy. (zx)))) (λx. (xy))

Alpha-conversion and solution:
((λx^1. (λy^0. ($z x^1$)))) (λx^0. ($x^0 y$))
= (λy^0. (z (λx^0. ($x^0 y$))))

Without Alpha-conversion and solution:
((λx. (λy. (zx)))) (λx. (xy))
= (λy. (z (λx. (xy))))

In this case, the function meaning of the next expression changes from original (y^0 and y are not the same)

b) The lambda calculus equation is: (λx. (λy. (λz. (xyz))) (λz. (zx))

Alpha-conversion and solution:
(λx^0. (λy. (λz^0. ($x^0 y z^0$))) (λz^1 . ($z^1 x$))
= (λy. (λz^0. ((λz1. ($z^1 x$)) y) z^0))
= (λy . (λz^0. ($y x z^0$))

Without Alpha-conversion and solution:
(λx. (λy. (λz. (xyz))) (λz. (zx))
= (λy. (λz. (λz. (zx)) y z))
= (λy. (λz. yxz))

In this case, the expression meaning remains the same.

c) The lambda calculus equation is: (λx. (xz)) (λx. (λz. (x y)))

Alpha-conversion and solution:
(λx^0. ($x^0 z$)) (λx^1. (λz^0. (x^1 y)))
= ((λx^1. (λz^0. (x^1 y))) z)
= (λz^0. zy)
= zy

Without Alpha-conversion and solution:

(λx. (xz)) (λx. (λz. (x y)))

= ((λx. (λz. (x y))) z)
= (λz . zy)

In this case, the expression has changed, with one z being dependent on the other

The lambda calculus equation is: (λx. (x y)) (λx. y)

With Alpha-conversion and solution:
(λx^0. (x^0 y^0)) (λx. y)
= ((λx.y) y^0)
= y

Without Alpha-conversion and solution:
(λx. (x y)) (λx. y)
= ((λx. y) y)
= y
Expression stays the same.

Part 4:

a. ((λy. z y) x) (λx . x y)
= ((λy^0. z y^0) x) (λx^1. x^1 y)
= (z x) (λx^1. x^1 y)

b. (λx . x x x) (λx . x x x)
= (λx^1. x^1 x^1 x^1) (λx. x x x)
= (λx ((x x) x)) (λx ((x x) x)) (λx ((x x) x)))
= ….
This function grows exponentially

c. (λx . x) (λy . x y) (λz. x y z)
= ((λx^1 . x^1) (λy^1 . x y^1)) (λz . x' y z)
= ((λy^1. x y^1) (λz . x' y z))
= (x (λz . (x' y z)))

d. Solution shared in PEN/PENCIL FORMAT.
e. Solution shared in PEN/PENCIL FORMAT.

d) $(\lambda m n f x . m f (n f x))\ (\lambda f x . f(x))\ (\lambda f(x) . f(f(f(x))))$

$\Rightarrow (\lambda n f x . (\lambda f x . f x) f (n f x))\ (\lambda f(x) . f(f(f(x))))$

$\Rightarrow (\lambda n f x . (\lambda x . f x)(n f x))\ (\lambda f(x) . f(f(f(x))))$

$\Rightarrow (\lambda n f x . (f n f x))\ (\lambda f x . f(f(f(x))))$

$\Rightarrow \lambda f(x) (f . (\lambda x . f(f(f(x)))) x)$

$\Rightarrow \lambda f(x) (f (f(f(f(x)))))$

4

e) SUCC 2.

$(\lambda n f x . f (n f x))\ (\lambda f x . f(f(x)))$

$\Rightarrow \lambda f x . f ((\lambda f x . f(f(x))) f x)$

$\Rightarrow \lambda f x . f ((\lambda x f (f(x))) x)$

$\Rightarrow \lambda f x . f(f(f(x)))$

3

Question 5: Lambda calculus (also shared in the .rkt file)

1.

```
(define (arg-max f L)
  (let loop ((L (cdr L))
             (m (f (car L)))
             (k (car L)))
    (cond
      ((null? L) k)
      ((> ( f (car L)) m)
       (loop (cdr L) ( f (car L)) (car L)))
      (else
       (loop (cdr L) m k)))))
```

2.

```
(define (zip . L)
  L)
```

3.

```
(define (unzip L m)
  (cond
    ((null? L) '())
    ((= 0 m)
     (car L))
    (else
     (unzip (cdr L) (- m 1)))))
```

4.

```
(define (check-list L m)
  (cond
    ((null? L) #f)
    ((= m (car L))
     #t)
    (else
     (check-list (cdr L) m))))
(define (intersectlist L1 L2)
  (let fnc ((FinList '())
            (L1 L1)
            (L2 L2))
    (cond
      ((null? L1) FinList)
      ((and (check-list L2 (car L1)) (not (check-list FinList (car L1))))
       (fnc (append FinList (list (car L1))) (cdr L1) L2))
      (else
       (fnc FinList (cdr L1) L2)))))
```

5.

```
(define (sortedmerge L1 L2)
  (let fnc ((FinList '())
            (L1 L1)
            (L2 L2))
    (cond
      ((null? L1) (append FinList L2))
      ((null? L2) (append FinList L1))
      ((< (car L1) (car L2)) (fnc (append FinList (list (car L1))) (cdr L1) L2))
      (else (fnc (append FinList (list (car L2))) L1 (cdr L2))))))
```

6.

```
(define (interleave L1 L2)
  (let fnc ((FinList '())
            (L1 L1)
            (L2 L2)
            (k 0))
    (cond
      ((null? L1) (append FinList L2))
      ((null? L2) (append FinList L1))
      ((= 0 k) (fnc (append FinList (list (car L1))) (cdr L1) L2 1))
      (else (fnc (append FinList (list (car L2))) L1 (cdr L2) 0)))))
```

7.

```
(define (inc x) (+ x 1))


(define (map2 L1 L2 pred f)
  (let fnc ((FinList '())
            (L1 L1)
            (L2 L2)
            (pred pred)
            (f f))
    (cond
      ((and (null? L1) (null? L2)) FinList)
      ((or (null? L1) (null? L2)) (string #\e #\r #\r #\o #\r))
      ((pred (car L1)) (fnc (append FinList (list (f (car L2)))) (cdr L1) (cdr L2) pred f))
      (else (fnc (append FinList (list (car L2))) (cdr L1) (cdr L2) pred f)))))
```

8. Please note that in DrRacket, the list is of the form:
   (edge-to-adjacency-list '((A B) (B C) (A D)))

   Edge To Adjacency List:

```
(define (edge-list-to-adjacency-list edgeList)
 (let loop ((visited-elements '())
          (final-list '())
          (L1 edgeList))
  (cond
   ((null? L1) final-list)
   (else
    (cond
     ((and (is-present-list visited-elements (caar L1)) (is-present-list visited-elements (cadar L1))) (loop visited-elements final-list (cdr L1)))
     ((is-present-list visited-elements (caar L1)) (loop (append visited-elements (list (cadar L1))) (append final-list (list (list (cadar L1) (get-list-of-adjacent-elements edgeList (cadar L1))))) (cdr edgeList)))
     ((is-present-list visited-elements (cadar L1)) (loop (append visited-elements (list (caar L1))) (append final-list (list (list (caar L1) (get-list-of-adjacent-elements edgeList (caar L1))))) (cdr edgeList)))
     (else (loop (append visited-elements (list (caar L1) (cadar L1))) (append final-list (list (list (caar L1) (get-list-of-adjacent-elements edgeList (caar L1)))) (list (list (cadar L1) (get-list-of-adjacent-elements edgeList (cadar L1))))) (cdr edgeList)))))
  )))
;helper function to create edge list from the initial adjacency list for all elements
(define (create-edge-list initialList)
 (let loop ((element (caar initialList))
          (list-lvl (cadr initialList))
          (final-list '()))
  (cond
   ((null? list-lvl) final-list)
   (else (loop element (cdr list-lvl) (append final-list (list (list element (car list-lvl)))))))))
```

(adjacency-to-edge-list '((A (B C)) (B (C)) (C (D))))

Adjacency to Edge List:

```
(define (adjacency-list-to-edge-list adjacencyList)
 (let loop ((final-list '())
          (L1 adjacencyList))
  (cond
   ((null? L1) final-list)
   (else
    (cond
     ((null? (cdar L1)) (loop final-list (cdr L1)))
     (else (loop (append final-list (create-edge-list (car L1))) (cdr L1))))))))
```