Section A.

1. a) 9 / 3 / 2 * 6 + 2 * 1.5

= 3 / 2 * 6 + 2 * 1.5

= 1 * 6 + 2 * 1.5

= 6 + 3.0

= 9.0 double.

b) i) C char is a number, a ASCII Representative of the orginal char.

length = 6.

ii) int strlength (char * str) {

int len = 0

while (str[len] != '\0') lenth++;

return len;

}

h e l l o
0 1 2 3 4 5

iii) char * copyStr (char* org)

←.

iv) int count (char* str, char a) {

return countRec (str, strlen(str) - 1, a);

}

int countRec (char* str, int i, char a) {

if (i < 0) return 0;

if (str[i] == a) return 1 + countRec (str, i-1, a);

else return countRec (str, i-1, a);

}.

2. a). 1. -0.666667.

2. True.

3. True.

4. Error. (* -) is not a function.

5. Error Char not comparable with [Char]

b)
```
maximum' :: Ord a => [a] -> a
maximum' [ ] = error "max ---"
maximum' [x] = [x]
maximum' (x : xs) | x < maxTail = maxTail
                  | otherwise   = x.
                    where maxTail = maximum' xs.
```

c)

1	0	3	3
2	1	4	6
		5	10

no

$$\frac{(0+(n-1))\,n}{2}$$

```
intersect :: Int -> Int.
intersect n = (n-1) * n `div` 2.
```

d)
```
allReverse :: [[a]] -> [[a]]
allReverse [ ] = [ ]
allReverse (x:xs) = allReverse xs ++ [reverse x].
```

```
reverse :: [a] -> [a]
reverse [ ] = [ ]
reverse (x:xs) = reverse xs ++ [x].
```

e)
```
foldl :: (a->b->a) -> a -> [b] -> a.
foldl f a [ ] = a
foldl f a (x:xs) = foldl f (f a x) xs
```

Section B

3. a) stack frame.

a place where local variables / parameters are stored (local scope)

local scope.

The scope which have access to a local. variable.

Pointer arithmetic.

Perform arithmetic operations on pointers to put an offset on the mem. addr.

Function definition

define a return type, function name and parameters for a function.

b) ✓ double =12 ; ok.
✗ long is incompatible with 2.3
✗ Cannot get the address of a int value

c) i) struct node {
int key
node* prev;
node *succ;
};

ii) struct node {
int key;
char *label;
node* succ;
};

iii). void addToList (char *label, int key, node *head) {
node* prevNode = head.
head = head->succ;
while (head != NULL) {
prevNode = head;
head = head->succ;

```
    }.
    Node* newNode = malloc(sizeof(node));
    newNode -> key = key
    newNode -> label = label
    prevNode -> succ = newNode
  }.
```

4. Virtual memory

Virtual memory is mapped to physical mem or disk to give programs a continious section of memory while preventing outer access to other physical memory.

free

to free up heap memory allocated dynamically by malloc/calloc

compound statements

a set of stmts. group together by { }.

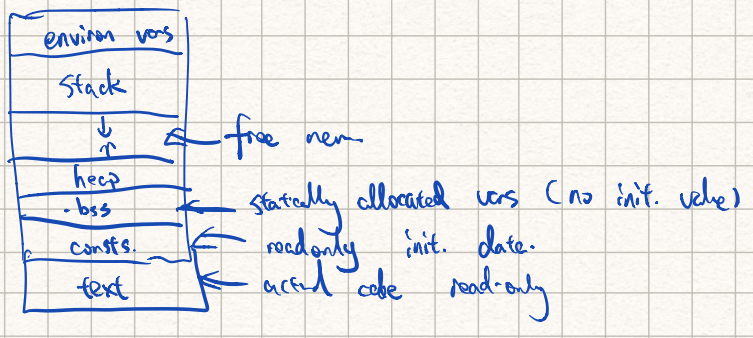
Var declaration.

declare a data type / init. value for a variable.

b)

```
int f(int n) {.
  if (n <= 1) return 1;
  else return n * f(n-1);
}.
```

c).



d)

```
void filledTriangle(int n). {.
  for (int i=0; i<n; i++) {.
    for (int j=0; j<=i; j++) printf("*");
    printf("\n");
  }.
}.
```

Seen in C.

pre condition.

Certain conditions that should be satisfied before the execution of some code.

side effect.

function doesn't alter any other variables outside of its local scope.

operator section.

pass only part of the parameters to an op. to make it a function. (e.g. (+1) -> \x->x+1.

higher order function.

a function that takes in a function as a parameter.

b)
```
raised :: IO()
raised = do
    input <- getLine
    putStrLn $ toUpperList input.

toUpperList :: String -> String
toUpperList xs = [toUpper x | x <- xs]
```

c)
```
zipWith :: (a->b->c) -> [a] -> [b] -> [c]
zipWith f [ ] [ ] = [ ]
zipWith f [ ] bs = [ ]
zipWith f as [ ] = [ ]
zipWith f (a:as) (b:bs) = f a b : zipWith f as bs.
```

d)
```
zipWith :: (a->b->c) -> [a] -> [b] -> [c]
zipWith f as bs = map (uncurry f) (zip as bs)
```

e)
```
data Vector a = Vector [a]

ok :: (Num a) => (Vector a, Vector a) -> Bool
ok (Vector xs, Vector ys). not null xs && not null ys &&
                           length xs == ys.
```

```
dotProduct :: (Num a) => (Vector a, Vector a) -> Vector a
dotProduct (Vector xs, Vector ys) | ok(Vector xs, Vector ys) = Vector (zipWith (*) xs ys)
                                  | otherwise error "not multipliable".
```

6. a)

recursive variant.

that changing values / parameters in a recursive function.

lambda abstraction

An anonymous function. / used to conveniently put fmes as parameters

partial application

partially gives parameters to a function to make it a new function.

type class - a set of datatypes with the same fmsgn due to similar characteristics

b)
```
coalesce :: [(a, Float)] -> [(a, Float)]
coalesce xs = foldl cluster [] xs

cluster :: [(a, Float)] -> (a, Float) -> (a, Float)
cluster acc new | acc `isIn` new = map (inc new) acc
                | otherwise      = acc ++ [new].

isIn :: (a, Float) -> [(a, Float)] -> Bool.
isIn new []   = false
isIn new (old : pairs) = if (fst new == fst old) then True else.
                         isIn new pairs.

inc :: (a, Float) -> (a, Float) -> (a, Float)
inc new old | (fst new == fst old) = (fst old, snd new + snd old)
            | otherwise            = old
```

c)
```
dist :: [(a, Float)] -> Bool.
dist pairs = (inRange pairs) && (sumPairs pairs == 1)
```

```
inRange :: [(a, Float)] -> Bool
inRange pairs = foldr (\acc new -> acc && (snd new <= 1 && snd new >= 0)) True pairs
```

```
sumPairs :: [(a, Float)] -> Float
sumPairs pairs = foldr (\acc new -> acc + snd new) 0 pairs
```

d)
```
normalise :: [(a, Float)] -> [(a, Float)]
normalise pairs | distPairs pairs = pairs
                | sumPairs pairs == 0 = error "set of 0s"
                | otherwise = divPairs pairs (sumPairs pairs)
```

```
divPairs :: [(a, Float)] -> Float -> [(a, Float)]
divPairs pairs k = map (\x -> x `div` k) pairs
```

e).
```
distribution :: [(a, Float)] -> [(a, Float)]
distribution pairs = normalise $ cycle pairs
```