2015 Section A.

1. a)
```
  3 | 2 * 4 + 10.6 / 2
= 1 * 4 + 10.6 / 2
= 4 + 10.6 / 2
= 4 + 5.3
= 9.3        double.
```

b) i)
```
int getColonPos (char* label) {
    int len = strlen(label);
    for (int i=0; i<len; i++) if (label[i] == ':') return i;
    return -1;
}
```

ii)
```
char* getLabelCopy (char* label) {
    int len = strlen(label);
    int colonPos = getColonPos(label);
    char *labelCopy = malloc(sizeof(char) * (len - colonPos));
    for (int i = colonPos+1; i<len; i++) labelCopy[i-colonPos-1] = label[i];
    labelCopy[len-colonPos-1] = '\0';
    return labelCopy;
}
```

iii)
```
int getLabelVal (char* label) {
    char* labelStr = getLabelCopy(label);
    int len = strlen(labelStr);
    int s = 0;
    for (int i=1; i<len; i++)
        s = s * 10 + (labelStr[i] - 48);
    return s;
}
```

2. a)
1. -533
2. (+) :: Num a ⇒ a → a → a.
3. error '+' '-' causes confusion
4. (\x → x) :: p → p
5. False

b)
```
ins :: Integer → [Integer] → [Integer]
ins x [] = [x]
ins x (y:ys) | x ≤ y      = x : (y : ys)
             | otherwise  = y : ins x ys
```

```
iSort :: [Integer] -> [Integer]
iSort [] = []
iSort x:xs = ins x (iSort xs)
```

c)
```
xor :: Bool -> Bool -> Bool.
xor x y = not ( x == y).
```

d)
```
combi :: (Real a) => [(a, a)] -> [a].
combi xs = sum [fst x | x <- xs, fst x > snd x] : [sum [snd x | x <- xs, fst x > snd x]]
```

e)
```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f [] [] = []
zipWith f [] ys = []
zipWith f xs [] = []
zipWith f (x:xs) (y:ys) = f x y : (zipWith f xs ys).
```

Section B.

a) variable lifetime.

the time period when a variable is accessible.

linking

Method of combining several object file to a single executable.

pointer dereference

Retrieving the value stored in memory address pointed by the pointer.

compound statements

statements grouped together by { }.

b) :) true / false are represented by 0, 1 int vals

↓

null.

ii) int isPrime (int n) {

```
int i;
int flag=1;
for (i=2; i<= trunc(sqrt(n)); i++)
        if (n % i ==0) flag= 0;
return flag;
}.
```

c) i)
```
typedef struct S.
    int a;
    char * b;
    double c;
} myStruct
```

ii)
```
myStruct * generator (int a, char* b, double c) {.
    myStruct item;
    item.a = a.
    item.b = b
    item.c = c
    return &item
}.
```

4. a) Scope - the code area where a variable / function is accessible.

heap memory - a part of memory that's using a heap structure to organise.

dynamically allocatable using malloc (calloc and free

abstraction - technique to abstract similar activities into functions

cast expression - to change data type while maximally preserving its original value

b) z = 1 + 3 * 2 = 7.

c).
```
int sum (int *a, int length) {
    int s=0;
    for (int i=0; i< length; i++)  s+= a[i];
    return s;
}
```

d)
```
void block (int n, char a, char b) {
    for (int i= 0; i< n; i++) {
        for (int j=0; j<n; j++) if (i % 2 ==0) printf("%c", a);
                                else printf("%c", b);
        for (int j=0; j<n; j++) if (i % 2 ==0) print - - - b.
                                else —  - a.);
        printf("\n");
    }
}
```

Section C.

5. a) type class : a set of func/behaviours which can be implemented to a set of datatypes

tail recursion : no operations is done after the return of a recursive func.

higher order function : function that takes in a function as a parameter.

Basic Types : types in Prelude package

b)
```
qsort :: [Int] -> [Int]
qsort []     = []
qsort [x]    = [x]
qsort (x: xs) = [l | l <- xs, l <= x] ++ [x] ++ [r | r <- xs, r > x]
```
(qsort) (qsort)

c) curry functions

allow lambda functions && partial results applied with higher order fnes

d)
```
curry :: ((a, b) -> c) -> a -> b -> c.
curry f a b = f (a,b).

uncurry :: (a -> b -> c) -> (a,b) -> c.
curry f (a,b) = f a b
```

e)
```
filterFirst :: [a] -> [a].
filterFirst (x:xs) | not (p x)   = xs
                   | otherwise   = x : filterFirst xs

filterLast :: [a] -> [a]
filterLast xs | not (last xs)   = init xs
              | otherwise       = (filterLast (init xs)) ++ [last xs].
```

6. a) lazy evaluation.

Haskell only evaluates expression when needed

Polymorphic types : a type that can be multiple data types

Pattern matching : checks whether given input follows certain pattern.

ii) wildcard pattern

list pattern

b)
```
backwards :: IO ()
backwards = do
              input <- getLine
              putStrLn (reverse input).
```

c)
```
mapFuns :: [a -> b] -> a -> [b].
mapFuns [] a = []
mapFuns (f:fs) a = (f a) : mapFuns a fs.
```

d)
```
mapFuns :: [a -> b] -> a -> [b]
mapFuns fs a = map (\f -> f a) fs
```

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

f) split :: [a] -> ([a],[a])

split xs -> ([o | i <- [1..length xs], odd i, let o = xs !! (i-1)],
[e | i <- [1..length xs], even i, let e = xs !! (i-1)]).