2018

Section A

1. a. 10/2 + 25/5 * 0.5
= 5 + 5 * 0.5
= 5 + 2.5
= 7.5 double

(b) int sumofNeg (int a[], int len) {

```
if (len <= 0) return 0;
int S=0;
for (int i=0; i<len; i++) if (a[i]<0) S+=a[i];
return S;
```

}

(c) i. since arr one dim.. pointer.

pointer + offset
↓
points at first element

ii.

a[5] *(a+5).

ii. int sumofNeg (int *a, int len) {

```
if (len≤0) return 0;
int S=0;
for (int i=0; i<len; i++) if (*(a+i)<0) S+=*(a+i)
return S;
```

}

(d)

```
int ** copy (int ** org, int len) {
    int ** new = malloc(sizeof(int *) * len);
    for (int i=0; i<len; i++) new[i]= org[i];
    return new;
}
```

2. (a)
1. 6.4
2. False.
3. 2.
4. True.
5. -3.

(b)

```
take :: Int → [a] → [a]
take 0 xs = [ ]
take n [ ] = error "Nothing left to take"
take n (x:xs) | n<0 = error "wrong n"
              | otherwise = x : take n-1 xs
```

(c) divisors :: Int → [Int]
divisors x = [y | y ← [1..x], x `mod` y == 0]

isPrime :: Int → Bool
isPrime x = divisors x == [1, x]

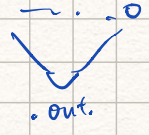
(d) cuber :: [Int] → Int
cuber xs = foldr \x acc → x^3 * acc 1. (filter (>0) xs)

(e) mono :: (Int → Int) → Bool.
mono f = foldr \x acc → (f(x) - f(x-1)) ≥ 0 && acc True. [1..n]

Section B

3. (a). void. empty / in fnctn without return / no param.

linking



literal value is what it exactly means. (not var. / element or representation)

fnctn decl. int f(--) ?

(b). Struct.
int key
node* prev, succ.

(c) naive str. matching

4. (a) input buffering

Save -- user to -- program.

fnctn def.

file scope.

The file area where global vars are accessible

std err.

an standard output stream where err msgs are sent.

(b) false - NULL. / 0.

true anything else then NULL / 0

short int [arr].

(c) stack.

stack frame is discarded when func exits.

Section C.

5. (a) high order func.

operator section. $(+1) = \backslash x \to x+1$.

type class set of $\mathbb{P}$

QC.

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

rev :: [a] -> [a].
rev xs = foldr snoc [] xs.
```

Use an accumulator approach.

(c)
```
iter :: Int -> (a -> a) -> a -> a
iter 0 f x = x.
iter n f x = iter (n-1) f (f x)

iter' :: Int -> (a -> a) -> a -> a.
iter n f x = foldr (\_ x -> f x) x [0..n-1]
```

(d)
```
data NTree = NilTree.
           | Node Int NTree NTree.

ntSum :: NTree -> Int.
ntSum NilTree = 0
ntSum (Node key lt rt) = key + ntSum lt + ntS
```

```
ntList :: NTree → List
ntList NilTree = [ ]
ntList (Node key lt rt) = key : (ntList lt ++ ntList rt)
```

6. (a) lazy evaluation: Haskell only evaluates expr when necessary

tail rec : no ops done after the return of recsive call.

poly — type : a type that can be ≥1 actual DT.

lambda abs : an anonymous func. / used when inserting func.

(b).

```
fac1 :: Int → Int
fac1 0 = 1
fac1 x = if x < 0 then error " cannot --- "
         else x * (fac1 (x - 1))
```

```
fac2 :: Int → Int
fac2 x | x<0 = error " —— "
       | otherwise = foldr (*) 1 [x, x-1 .. 1].
```

```
prop_facs :: Int → Property
prop_facs x = x > 0 ==>.
              fac1 x == fac2 x
prop_fac_larger :: Int → Property
prop_facs_larger x = x > 0 ==>.
                     (fac1 x ≥ x) && (fac 2x ≥ x)
```

```
testInt :: IO(Bool)
testInt = do
    input ← getInt.
    if input == 0 then return True.
                  else return (input `mod` 2 == 0) &&
                       testInt
```