Ans 1

(define (listfromTo a b)

(if (= a b) (list a) (cons a (listfromto (+ a 1) b))))

(define (removeMult a l)

(cond ((null? l) '() )

(else (if (and (= 0 (modulo (car l) a)) (not (= (car l) a)) ) (removeMult a (cdr l)) (cons (car l) (removeMult a (cdr l))) ))

))

(define (sieveaux n a l)

(cond ((= a n) l)

(else (sieveaux n (+ 1 a) (removeMult a l))))

)

(define (sieve n)

(sieveaux n 2 (listfromTo 2 n))

)

Ans 2(a)

Functions added or modified in interpreter. Complete interpreter and library attached with the assignment.

(define (my-eval exp env)

(cond

((symbol? exp) (lookup exp env))

((not (pair? exp)) exp)

((eq? (car exp) 'quote) (cadr exp))

((eq? (car exp) 'cond) (handle-cond (cdr exp) env))

((eq? (car exp) 'let) (handle-let (cadr exp) (cddr exp) env ))

((eq? (car exp) 'let\*) (handle-let\* (cadr exp) (cddr exp) env ))

((eq? (car exp) 'letrec) (handle-letrec (cadr exp) (cddr exp) env (cdr (list '())) (cadr exp)))

((eq? (car exp) 'if)

(handle-if (cadr exp) (caddr exp) (cadddr exp) env))

((eq? (car exp) 'longest)

(handle-longest (cdr exp) env))

((eq? (car exp) 'lambda)

(list 'closure exp env))

((eq? (car exp) 'letrec)

(handle-letrec (cadr exp) (cddr exp) env)) ;; see explanation below

(else

(handle-call (map (lambda (sub-exp) (my-eval sub-exp env)) exp)))

))

(define (len l)

(if (null? l) 0 (+ 1 (len (cdr l)))))

(define (evalexp l index count)

(if (= count index) (top-eval (cadar l)) (evalexp (cdr l) index (+ 1 count)))

)

(define (findmax l count m maxl)

(cond ((null? l) m)

(else (if (> (len (top-eval (caar l))) maxl) (findmax (cdr l) (+ 1 count) count (len (top-eval (caar l)))) (findmax (cdr l) (+ 1 count) m maxl)))))

(define (handle-longest l env)

(evalexp l (findmax l 1 1 0) 1)

)

Ans 2(b)

Since let, cond are the syntax of scheme language, we can use it in the interpreter. Inbuilt grammar is defined for let, cond, if e.t.c which is automatically parsed whenever these keywords are encountered. Therefore we can use those in the definitions of handle-if, handle-cond itself.

Ans 3 (a)

fun compose f g = fn x => g (f x);

Ans3 (b)

Since ML is strongly typed language, it knows at the compile time what operations can be performed on a list beforehand. If the list is not homogenous, then it can’t know the operations that can be performed on the list. This feature helps in removing run-time errors that can occur in other languages which check type during run-time(It may result in run-time errors like if you add a string to an integer). In ML all type checking is performed at the compile time.

Ans3 (c)

fun foo f (op >) (x,y) z =

let fun bar a = if x > y then z else a

in bar [1,2,3]

end

'a -> ('b \* 'c -> bool) -> 'b \* 'c -> int list -> int list

Ans3 (d)

Initially ML will assign different variables to all different arguments.

1. f can be anything so we take it as ‘a

2. (op >) takes a tuple of two arguments which can be of any type 3. (‘b\*’c). Note that it is not necessary that two elements will be same because (op >) may have been defined such as l1 > length l2.

4. For now x and y can be a tuple of any thing so lets call them (‘d \* ‘e)

5. z can also be anything (‘f)

Now, in the function body, we are doing > operation on x and y, therefore x and y must of same type as the input of (op >). Therefore x will be ‘b and y will be ‘c

We also know that z and a must be of same type since in SML return type is same. In let body we are calling function bar with int list therefore a must be int list. Since a and z are of same type therefore z will also be of int list.

Also the function returns int list therefore it will be the return type of the function.

Hence final types we have is

'a -> ('b \* 'c -> bool) -> 'b \* 'c -> int list -> int list

Ans4(a). This is not an ADT because here Queue is not a type and there is only one queue. For it to be an ADT, we should also pass the queue on which we will perform insert and extract.

Ans4(b). package queue is

type queue is private

function extract (q: queue) return integer;

function insert (q: queue, x: integer) ;

end queue

private

type queue is record

elements ;

front : Natural

back :Natural

end queue package;

package body queue is

function extract(q : queue) return integer is

begin

return q.elements(q.front);

end extract;

Procedure insert(q: in out queue, x : integer) is

Begin

q.elements(q.back) = x;

q.back = q.back + 1;

end insert;

end queue;

Ans5(a). (i) Encapsulation of data and code into objects.

(ii) Inheritance

(iii) Subtyping with dynamic overloading(ability to treat

one class as if it was other class)

Ans5(b).

(i). A type B is a subtype of A, if all the values of B are also in A or(B has a subset of values which A has). This is subset interpretation of subtyping.

(ii). Suppose a class B (Car) extends class A (Vehicle). Now suppose set S corresponds to A and set S’ corresponds to set B. Now we know that B extends A i.e. B is a subtype of A, therefore every B will be an A i.e. every car will be a vehicle. Wherever an A is required a B will work. Therefore wherever an A is required, it can take more values than where B is required. Hence S forms the superset of S’. Therefore It fits into the subset interpretation of subtyping.

Example:

class Vehicle {

int speed;

int position;

}

class Car extends Vehicle {

int color;

}

Here we can do

Car c=new Car();

Vehicle v = c;

Now v is a vehicle but it can point to a car(since every car is a vehicle and color is just a limiting factor for car). Therefore Java satisfies subset interpretation of subtyping.

(iii). Subtyping on functions in Scala is contra-variant on the input types. For example if

B <: A then A-> Int <: B -> Int

Here whenever an A is expected, we can send a B as all B’s are A’s. Hence B –> Int defines a larger set than A -> Int, hence B->Int :> A->Int

Subtyping on functions in Scala is co-variant w.r.t to Output types. For example if

B <: A then Int->B <: Int->A

Here whenever a function is expected to return an A, it can also return a B, since all B’s are A too. Therefore Wherever an A is expected a B can be returned therefore

Int->B <: Int->A

Ans 5(c)

abstract class Tree[T <: Ordered[T]]

case class Node[T <: Ordered[T]](v:T, l:Tree[T], r:Tree[T]) extends Tree[T]

case class Leaf[T <: Ordered[T]](v:T) extends Tree[T]

def max\_tree[T <: Ordered[T]](obj : Tree[T]):T =

obj match {

case Leaf(v) => return v;

case Node(v, l, r) => if (v >= max\_tree(l) && v >= max\_tree(r)) return v

else if (v <= max\_tree(l) && max\_tree(l) >= max\_tree(r)) return max\_tree(l)

else return max\_tree(r)

}

Ans5(d).

(i)

Suppose B is a subtype of A, checking if C<B> is a subtype of C<A>

void f(C<A> x) {

A y;

x.insert(y);

return x;

}

main(){

C<B> z;

f(z);

}

Here we are passing C<B> where a C<A> is required. In the insert function now we will trying to insert A into a collection of B. Now suppose there was another function, which iterates through the collection of B, when it encounters A and doesn’t have a field which is in B, it’ll be error.

(ii)

void f(C<T extends A> x) {

A y;

x.insert(y);

return x;

}

Here T can be any of the class which is a subtype of A.

Ans5(e).

(i).

Class C[+T]{

}

(ii).

Class List[+T]{

Static int top=0;

(def insert(List[A] : L , element : A) => L[top++]=element;

}

Here in insert function, suppose B is a subtype of A i.e. B <: A, then in a list of A, we can insert any element A or B, since all B’s are A’s. Therefore, List[A] can have more elements than List[B], hence List[A] :> List[B]

Ans5 (f).

(i)

Class C[-T] {

}

(ii).

Class List[-T] {

def extract(L: List[A]) => return L[--top];

}

Here the function is expected to return an Object of type A, but if we pass a List[B] instead, a B object will be returned. Since wherever an A is expected, A b will work fine since all B’s are A’s. Therefore, any subtype of List[A] can be passed here. Hence List[T] is contravariant w.r.t A.

Ans6 (a).

(i). Big cycles/chains may cause problems because when we have to update reference count of one the root of these chains, we will have to traverse the whole chain which will take time. In Mark and Sweep collectors, we just have to traverse the cycle whenever heap fills up. Also if there are loops, RC might fail to detect loops, as one will refer to other in the loop and their reference count will never reach 0.

(ii). Since counts can be much larger, there will be a lot of memory wastage in storing these counts. Since we don’t know the exact range of these counts, there will be cases when there will be an overflow.

Ans6 (b).

The cost of garbage collection in Copying mark and sweep is proportional to the size of live objects O(L) instead of size of heap O(M) in Mark and Sweep Garbage Collector.

Ans6 (c).

Generational copying garbage collection uses the basic idea that temporary variables are much more than the long-lived objects. Therefore it allots more space for temporary objects and lesser space for long-lived objects. It can have different levels, which represent life of different objects. Suppose if an object is still live after the first round of copying gc, then it is promoted to second level. Now we know that, if the object is still live after first round, there is a good probability that this object’s lifetime is long and therefore Garbage collection on second level is called lesser number of times as compared to first level. Here number of levels will be equal to number of heaps. These heaps are called generations. Younger Generations call Garbage collectors more times as there are more temporary objects. Live objects are traversed to older generations if they outlive the garbage collection phase in younger generation.

Ans6(d).

void delete(node \*x)

{

long ref\_count=x.refcount;

node\* xadj=x->next;

if(ref\_count==0)

{

while(xadj!=NULL)

{

delete(xadj);

xadj=xadj->next;

}

addToFreeList(x);

}

else

x.refcount --;

}