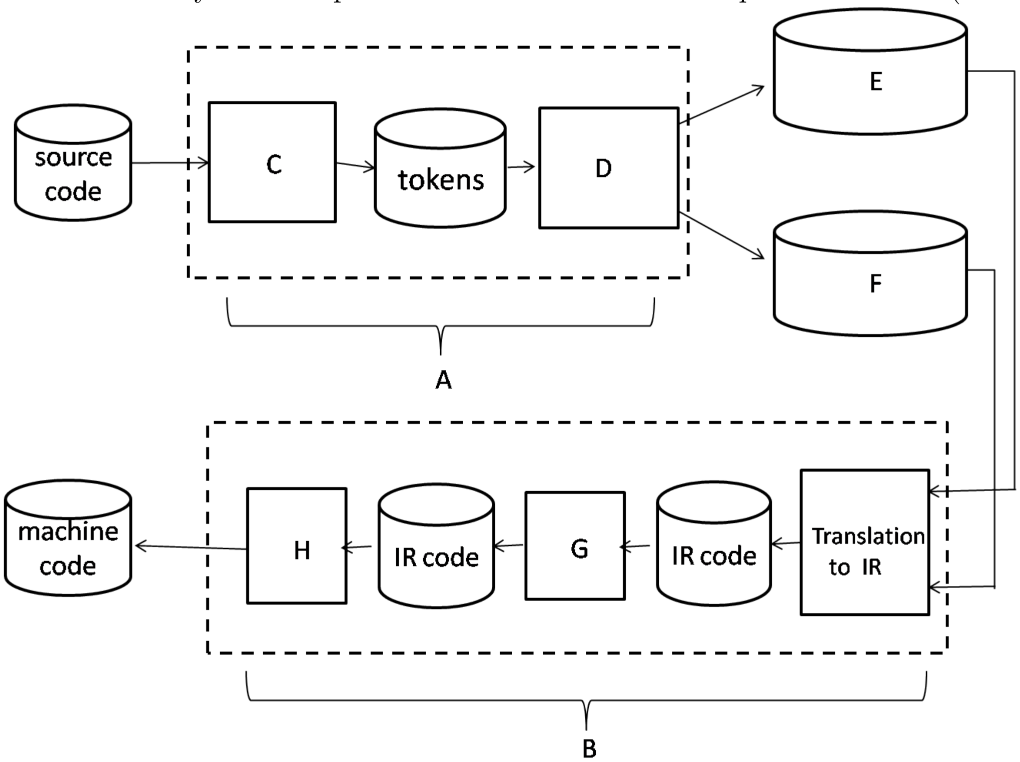
1. (8 pts) Fill in the blanks below, giving the names of the various parts of a compiler. (Recall that the cylinders represent data and the boxes represent actions (i.e. functions).)



A front-end

B back-end

C lexer

D parser

E AST

F symbol table

G optimization

H code generation

2. (12 pts) For each of the statements below, indicate which memory management approach(es) it describes: reference counting (RC), non-copying garbage collection (NG), or copying garbage collection (CG). If a statement applies to more than one approach, you should write all of the approaches it describes.

   Cannot handle cyclical references

   RC

   Uses a "free area" model to represent free memory

   CG

   Is best for spreading out the cost of garbage collection throughout the program

   RC

   At any time, only half of memory is in use

   CG

   Unreachable memory may not be freed immediately

   NG, CG

   Iterates over the entire heap at once (not just reachable memory)

   NG

   Does not move reachable data

   RC, NG

3. (14 pts) In class, we gave the following translation schemes for translating source programs into an intermediate representation (IR). All but the first take an AST (expression or statement) to a sequence of IR instructions.

   [e] : translate expression e to IR; returns pair (IR instruction list, location of value)

   [S] : translate statement S to IR

   $[e]_x$: translate expression e to code that stores value of e in variable x

   $[S]_L$: translate statement S in context of a loop or switch statement, where L is the target of a break statement

   $[e]_{Lt,Lf}$: translate expression e to code that branches to Lt if e is true, or Lf otherwise (the short-circuit evaluation scheme)

   The instructions in our intermediate representation were: x = n; x = y; x = y + z (for any operation +); JUMP L; CJUMP x, L1, L2; and x = LOADIND y.

   (a) Give the following translations. (You may use functions getloc() and getlabel() to get fresh memory locations and fresh instruction labels, respectively.)

      i. $[e_1 + e_2]$

         let t1, t2, t3 = getloc() in
         $[e_1]_{t1}$
         $[e_2]_{t2}$
         t3 = t1 + t2

      ii. $[e_1 \; ? \; e_2 \; : \; e_3]_x$ (for full credit, use the short-circuit scheme for e_1)

         $[e_1]_{L1,L2}$
         L1: $[e_2]_x$
         JUMP L3
         L2: $[e_3]_x$
         L3:

      iii. $[e_1 \; \&\& \; !e_2]_{Lt,Lf}$ (e_2 should not be evaluated if e_1 is false)

         $[e_1]_{L1,Lf}$
         L1: $[e_2]_{Lf,Lt}$

(b) (5 pts extra credit) Give IR code for a for loop. A for loop has the form "for(S_1; e; S_2) S_3", where S_1 is executed before the loop begins, the loop ends when e evaluates to false, S_2 is executed at the end of each iteration of the loop, and S_3 is the loop body. For full credit, use the short-circuit scheme for e.

$[S_1]$
JUMP L2
L1: $[S_3]_{L3}$
$[S_2]$
L2: $[e]_{L1,L3}$
L3:

4. (22 pts)

   (a) Give the type of the following function: fun f -> fun g -> fun x -> g (f x) x

   $(\alpha \rightarrow \beta) \rightarrow (\beta \rightarrow \alpha \rightarrow \gamma) \rightarrow \alpha \rightarrow \gamma$

   (b) Write an OCaml function *update* such that update f a b is a function that returns b when given a as input but otherwise behaves the same as f.

   let update f a b = fun x -> if x = a then b else f x

   (c) Write an OCaml function *double* that duplicates each element of a list, using fold_right instead of explicit recursion. For example, double [1; 2; 3] = [1; 1; 2; 2; 3; 3]. Remember that fold_right has type (α -> β -> β) -> α list -> β -> β.

   let double lis = fold_right (fun x y -> x :: x :: y) lis []

   (d) Write an OCaml function *sum_pairs* that takes a list of pairs and returns a list containing the sum of the elements of each pair, using map instead of explicit recursion. For example, sum_pairs [(1, 2); (3, 4); (5, 6)] = [3; 7; 11].

   let sum_pairs = map (fun (x, y) -> x + y)

(e) (5 pts extra credit) Write an OCaml function *maxf* that takes a function f and a list lst and returns a pair (max, index), where *max* is the largest value produced by applying f to an element of lst, and *index* is the index in lst of the element x such that f x = max, where the first element of the list has index 0. If there are multiple such elements, you may return the index of any one of them. For example, maxf (fun x -> x + 2) [1; 2; 3] = (5, 2). You may assume that lst is never empty. You may also assume that f takes elements of lst and returns only positive integers. Your function should use fold_right instead of explicit recursion.

```
let maxf f lst = fold_right (fun x (m, i) -> if f x > m then (f x, 0) else (m, i+1)) lst (0,0)
```

5. (15 pts) In homework 9, you defined multisets to be functions of type α -> int; in particular, you used the definition `type 'a multiset = 'a -> int`. In that homework, you defined functions add, member, union, disjointUnion, intersection, remove, filter, and fromList. Define the following additional functions on multisets:

   (a) fromSet: 'a set -> 'a multiset, such that fromSet s returns a multiset containing 1 copy of each element in s. Recall that the set type is defined by `type 'a set = 'a -> bool`.

   let fromSet s = fun x -> if s x then 1 else 0

   (b) count: 'a multiset -> 'a list -> int, such that count m lst returns the total number of occurrences of elements from lst in m. You may assume that lst contains no duplicate elements.

   let count m lst = fold_right (+) (map m lst) 0

   (c) subtract: 'a multiset -> 'a multiset -> 'a multiset, such that subtract a b has n copies of the value x if a has p copies and b has q copies and n = p - q. If b has more copies of x than a, then subtract a b should have 0 copies of x.

   let subtract a b = fun x -> max (a x - b x) 0

6. (15 pts) A multiset, or a bag, is a set that can contain multiple copies of an element. Just like sets, multisets are not ordered. In this assignment we represent multisets with functions. A multiset function returns the number of occurrences of the given element.

```
type 'a multiset = 'a -> int

let emptymultiset : 'a multiset = fun x -> 0
```

Some examples for possible implementations of multisets:

```
{1,1,1,2,2} = fun n -> match n with
                         1 -> 3
                       | 2 -> 2
                       | _ -> 0
{4,2,4,2,3} = fun n -> if n = 4 || n = 2 then 2
                       else if n = 3 then 1
                       else 0
```

Implement the following multiset operations.

(a) `add n s : int -> 'a multiset -> 'a multiset`.

```
let add n s = fun x -> if x=n then s x + 1 else s x
```

(b) `member n s : 'a -> 'a multiset -> bool`.

```
let member n s = s n > 0;;
```

(c) `union s1 s2 : 'a multiset -> 'a multiset -> 'a multiset`.
E.g. $\{1,1,1,2,2,3\} \cup \{1,1,2,3,3,4\} = \{1,1,1,2,2,3,3,4\}$

```
let union s1 s2 = fun x -> max (s1 x) (s2 x)
```

(d) `disjointUnion s1 s2 : 'a multiset -> 'a multiset -> 'a multiset`.
E.g. $\{1,1,1,2,2,3\} \uplus \{1,1,2,3,3,4\} = \{1,1,1,1,1,2,2,2,3,3,3,4\}$

```
let disjointUnion s1 s2 = fun x -> s1 x + s2 x
```

(e) `intersection s1 s2 : 'a multiset -> 'a multiset -> 'a multiset`.
E.g. $\{1,1,1,2,2,3\} \cap \{1,1,2,3,3,4\} = \{1,1,2,3\}$

```
let intersection s1 s2 = fun x -> min (s1 x) (s2 x)
```

(f) `remove n s : 'a -> 'a multiset -> 'a multiset`.
Remove an occurrence of `n` from `s`.

```
let remove n s = fun x -> if x=n then if s n > 0 then s n - 1 else 0
                          else s x
```

7. Use the simplification rules given in the notes to evaluate the following expression. As on the homework, mark each rewrite with the name of the rule you used. Note that functions fst and snd, which take the first and second element of a pair, respectively, are defined in δ rules. The evaluation has 18 steps. We've given you hints for the first few steps, by naming the simplification rule used. (*Note:* If we ask a question like this on the exam, we will give you the simplification rules; also, the evaluation won't be this long.)

```
    let f = fun y -> fun z -> (z,y)
    in let rec g = fun p -> if fst p = 0 then snd p else g(snd p, fst p - 1)
       in g (f 0 2)

==> (let)
    let rec g = fun p -> if fst p = 0 then snd p else g(snd p, fst p - 1)
    in g ((fun y -> fun z -> (z,y)) 0 2)

==> (beta)
    let rec g = fun p -> if fst p = 0 then snd p else g(snd p, fst p - 1)
    in g ((fun z -> (z,0)) 2)

==> (beta)
    let rec g = fun p -> if fst p = 0 then snd p else g(snd p, fst p - 1)
    in g (2,0)

==> (letrec1)
    let rec g = fun p -> if fst p = 0 then snd p else g(snd p, fst p - 1)
    in (fun p -> if fst p = 0 then snd p else g(snd p, fst p - 1)) (2,0)

==> (beta)
    let rec g = fun p -> if fst p = 0 then snd p else g(snd p, fst p - 1)
    in if fst (2,0) = 0 then snd (2,0) else g(snd (2,0), fst (2,0) - 1))

==> (delta)
    let rec g = fun p -> if fst p = 0 then snd p else g(snd p, fst p - 1)
    in if 2 = 0 then snd (2,0) else g(snd (2,0), fst (2,0) - 1))

==> (delta)
    let rec g = fun p -> if fst p = 0 then snd p else g(snd p, fst p - 1)
    in if false then snd (2,0) else g(snd (2,0), fst (2,0) - 1))

==> (if)
    let rec g = fun p -> if fst p = 0 then snd p else g(snd p, fst p - 1)
    in g(snd (2,0), fst (2,0) - 1))

==> (delta)
    let rec g = fun p -> if fst p = 0 then snd p else g(snd p, fst p - 1)
```

```
    in g(0, fst (2,0) - 1))

==> (delta)
    let rec g = fun p -> if fst p = 0 then snd p else g(snd p, fst p - 1)
    in g(0, 2 - 1)

==> (delta)
    let rec g = fun p -> if fst p = 0 then snd p else g(snd p, fst p - 1)
    in g(0, 1)

==> (letrec1)
    let rec g = fun p -> if fst p = 0 then snd p else g(snd p, fst p - 1)
    in (fun p -> if fst p = 0 then snd p else g(snd p, fst p - 1)) (0,1)

==> (beta)
    let rec g = fun p -> if fst p = 0 then snd p else g(snd p, fst p - 1)
    in if fst (0,1) = 0 then snd (0,1) else g(snd (0,1), fst (0,1) - 1)

==> (delta)
    let rec g = fun p -> if fst p = 0 then snd p else g(snd p, fst p - 1)
    in if 0 = 0 then snd (0,1) else g(snd (0,1), fst (0,1) - 1)

==> (delta)
    let rec g = fun p -> if fst p = 0 then snd p else g(snd p, fst p - 1)
    in if true then snd (0,1) else g(snd (0,1), fst (0,1) - 1)

==> (if)
    let rec g = fun p -> if fst p = 0 then snd p else g(snd p, fst p - 1)
    in snd (0,1)

==> (letrec2)
    snd (0,1)

==> (delta)
    1
```

8. Consider the following Java class:

```
class A
{
   public void f(Object o) { }
}
```

A *v-table* is a table of pointers to all non-static methods. Here is `A`'s *v-table*.

⟶	A's `f(Object o)`

(Note hwo we've identified the specific method to which the v-table points.)

(a) Draw the *v-table* of the following class (using the same notation to identify specific methods):

```
class B extends A
{
   public void f(String s) { } // overloading
}
```

⟶	A's `f(Object o)`
⟶	B's `f(String s)`

(b) Draw the *v-table* of the following class:

```
class C extends B
{
   public void f(Object o) { } // overriding
}
```

⟶	C's `f(Object o)`
⟶	B's `f(String s)`

(c) For each call site, show which method is invoked at runtime.

```
String strval = "Hello";
Object objval = "World";

A b1 = new B();
```

```
b1.f(strval);  //  A's f(Object)

B b2 = new B();
b2.f(strval);  //  B's f(String)
b2.f(objval);  //  A's f(Object)

A c1 = new C();
c1.f(strval);  //  C's f(Object)
c1.f(objval);  //  C's f(Object)

C c2 = new C();
c2.f(strval);  //  B's f(String)
c2.f(objval);  //  C's f(Object)
```

9. (14 pts) Write a function object in Java for the OCaml function *apply_pos*, defined as follows:

   apply_pos f lst = map (fun x -> if x > 0 then f x else x) lst

   For simplicity, we assume that lst is a list of integers. As in the OCaml code, your Java solution should call Map.map, which is given here:

```
interface IntFun {
   int apply (int n);
}

class Map {
   static int[] map (IntFun f, int lis[]) {
      int lis2[] = new int[lis.length];
      for(int i = 0; i < lis.length; i++)
         lis2[i] = f.apply(lis[i]);
      return lis2;
   }
}

class Apply_Pos {
   static int[] apply_pos (final IntFun f, int lis[]) {
      // complete this method
      IntFun g = new IntFun(){
        int apply(int n){
          return n > 0 ? f.apply(n) : n;
        }
      };
      return Map.map(g, lis);
   }
}
```