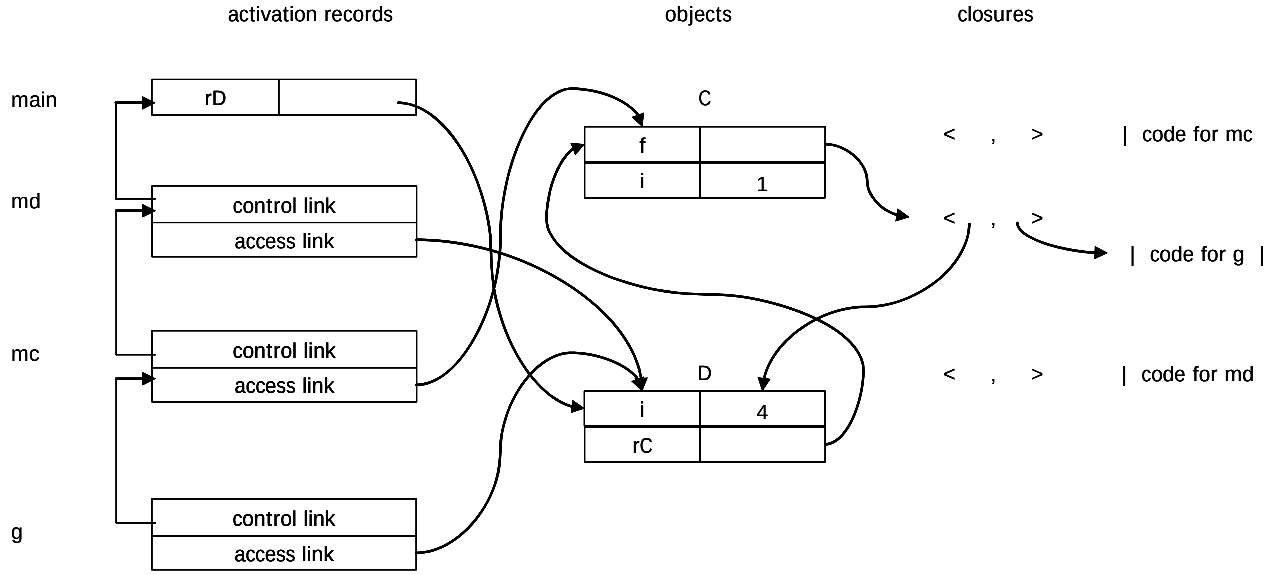
Question 1 Runtime-systems, scoping, types (weight 40%)

1a (25%)



1 b (15%)

a)

The call `f(rP)` via the call `rc.mc()`will imply that `g` is called with `rp` that denotes a `Point` object, and `g` attempts to access the `c` attribute.

b)

No. The problem arises from the assignment of `rP` to the formal parameter `cp` as a result of the call `f(rp)` via the call `rc.mc()`. An explicit casting should have been `f((ColorPoint)rp)`, but that would rule out calls `f(rp)`where `rp` denotes a `Point` object, and that should be allowed in cases where f is given an actual parameter with a same parameter type, i.e. `Point`.

c)

Yes. `rP = new ColorPoint()`

Question 2 ML (weight 40%)

2a Type inference (15%)

a)

The function f1 takes two lists as input and returns a list of pairs with corresponding elements. (Extra: The function will fail if the lists are of different lengths.) (The function is also known as the zip function)

b)

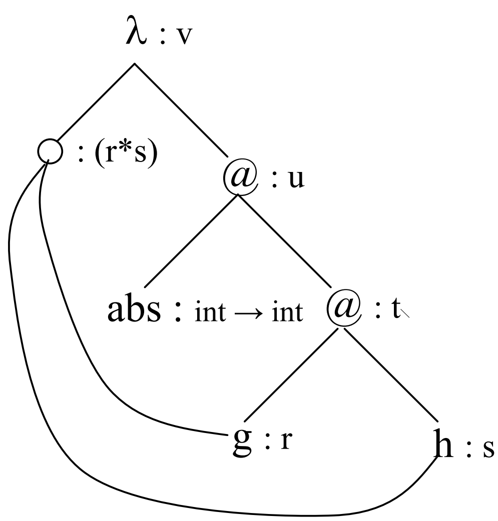
The type of f1 is `'a list * 'b list -> ('a * 'b) list`

The input `x::xs` and `y::ys` are lists, which we see by the `::` operator (and the `[]` in the second clause). We call the types of the two arguments `'a list` and `'b list` . The output is also a list, which we see by the `::` operator(or the `[]`) in the right hand side. By the first clause we see that it must be a list of pairs (2-tuples). The type of the first member of the pair must match the type of the elements in the first input list (`'a`) and the second must match the type of the elements in the second list (`'b`). Hence the output of the function is a value of type `('a*'b) list` .

c)

The type is `('a → int) * 'a → int`

1. Assign types to the subexpressions in the tree, using variables (r,s,t, etc.) where the type is unknown.



2. generate a set of constraints on the types (using the rules for abstraction and application):

$r = s \rightarrow t$

$int \rightarrow int = t \rightarrow u$

$v = r * s \rightarrow u$

3. Solve the constraints by unification/substitution

1. $int \rightarrow int = t \rightarrow u$	=>	$t=int$, $u=int$
2. $r = s \rightarrow t$	=>	$r = s \rightarrow int$ (by 1.)
3. $v = r * s \rightarrow u$	=>	$v = (s \rightarrow int) * s \rightarrow int$ (by 1 and 2)

Use `'a` for s and the resulting type is: `('a → int) * 'a → int`

2b Programming with lists (15%)

a)

```
fun getEquals((x,y)::ps)= if x=y then (x,y)::getEquals(ps) else getEquals(ps)
  | getEquals(nil) = nil ;

fun sumPairs((x,y)::ps) = (x+y)::sumPairs(ps)
  | sumPairs(nil) = nil ;
```

Lots of other variants are also possible. F.ex.

```
fun getEquals(nil) = nil
  | getEquals(p::ps) =
    if #1(p) = #2(p) then (#1p,#2p) :: getEquals(ps) else getEquals(ps) ;
```

b)

```
fun getEquals(ps) = filter (op=) ps ;
fun sumPairs(ps) = map (op+) ps ;
```

c)

```
fun snoc(x,xs) =
      case xs of nil => [x]
            | y::ys => y::(snoc(x,ys)) ;
```

or alternatively

```
fun snoc(x,(y::ys)) = y::(snoc(x,ys))
      |snoc(x,nil) = [x] ;
```

2c Records

```
fun listToRec(rs:(state list), {il=is,jl=js}) =
    case rs of (r::rs') =>
      listToRec(rs', { il=snoc((#i r),is) , jl=snoc((#j r),js) })
              | nil => {il=is,jl=js} ;
```

Other solutions are possible, but the lists in the resulting record should come out in the same order as the input lists and not reversed.

Question 3 Prolog (weight 20%)

3a

royal(X,male,_,_).

(1% without the _)

3b

- (male(X) :- royal(X,male,_,_).
- female(X) :- royal(X,female,_,_).
- child(X,Y) :- royal(X,_,Y,_).
- descendant(X,X).
  descendant(X,Y) :- child(X,Z), descendant(Z,Y).
- older(X,Y) :- royal(X,_,_,YearX), royal(Y,_,_,YearY), YearX < YearY.

3c

```
candidate(X) :- regent(K), descendant(X,K),
        ( male(X);
          female(X), \+ born_before(X,1971) ).
```

3d

```
yca(X,X,X).
yca(X,Y,A) :- older(X,Y), child(Y,P), yca(X,P,A).
yca(X,Y,A) :- \+ older(X,Y), child(X,P), yca(P,Y,A).
```