1a

```
public class MyStack<T> {
    int maxSize;
    Object[] stackArray;
    int top = -1;

    public MyStack(int maxSize) {
        this.maxSize = maxSize;
        stackArray = new Object[maxSize];
    }

    public void push(T element) {
        stackArray[++top] = element;
        // HERE, see text below
    }
    public T pop() {
        return (T) stackArray[top--];
    }
    public boolean isEmpty() {
        return top < 0
    }
}
```

```
class Person {
    public String name;
    public Person(String name) { this.name = name; }
    /* some more content here */
}

class Student extends Person {
    public Student(String name) { super(name); }
    /* some more content here */
}
```

Suppose now that the stack is used like this:

```
class Program {
    public static void main(String[] args) {
        MyStack<Student> s = new MyStack<Student>(10);
        s.push(new Student("Volker"));
        s.push(new Student("Eyvind"));
    }
}
```

Draw the runtime stack with activation blocks and objects (including static and dynamic links, using `this` for static links to objects, and local variables), at the point when the call to `s.push(new Student("Eyvind"))` has just been made, and the execution is at the point labeled "`// HERE`" in the code. You may assume that arrays are implemented as objects with an appropriate number of slots for their elements.

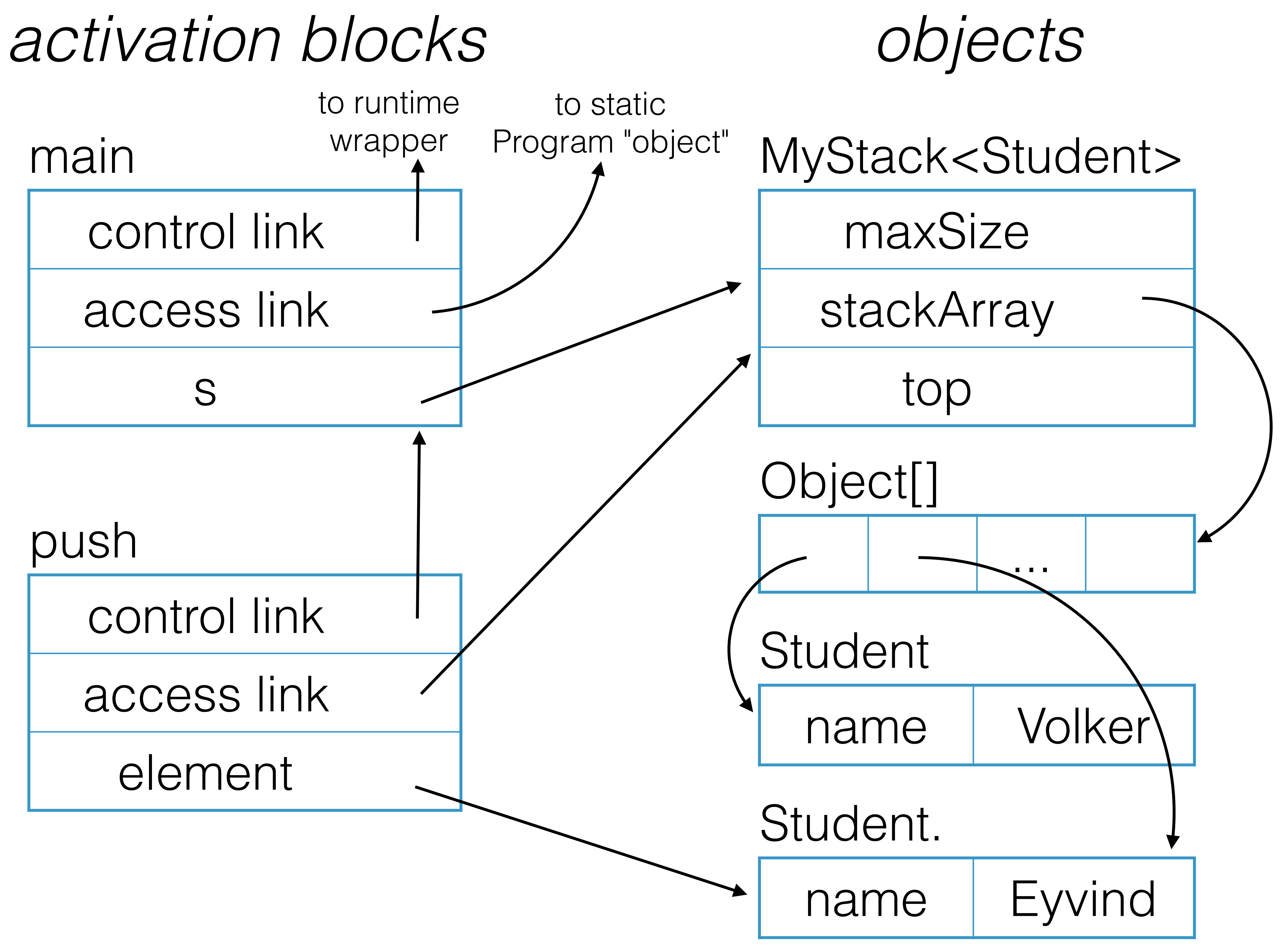
1a

```java
public class MyStack<T> {
    int maxSize;
    Object[] stackArray;
    int top = -1;

    public MyStack(int maxSize) {
        this.maxSize = maxSize;
        stackArray = new Object[maxSize];
    }

    public void push(T element) {
        stackArray[++top] = element;
        // HERE, see text below
    }
    public T pop() {
        return (T) stackArray[top--];
    }
    public boolean isEmpty() {
        return top < 0
    }
}
```

```java
class Person {
    public String name;
    public Person(String name) { this.name = name; }
    /* some more content here */
}

class Student extends Person {
    public Student(String name) { super(name); }
    /* some more content here */
}
```

Suppose now that the stack is used like this:

```java
class Program {
    public static void main(String[] args) {
        MyStack<Student> s = new MyStack<Student>(10);
        s.push(new Student("Volker"));
        s.push(new Student("Eyvind"));
    }
}
```



(viser ikke arv)

1b

Consider now the following program fragment, which uses the stack implementation from **1a**:

```
MyStack<Object> myStack = new MyStack<Object>(10);
myStack.push("Hello INF3110!");
myStack.push(new Object());
myStack.push(123);
```

Does this program fragment work (i.e., does it compile and run without any errors, provided it is wrapped in a suitable method and class declaration)? If yes, explain briefly how and why it works. If not, explain briefly what is wrong with it.

Ja, det virker. myStack.push tar imot objekter som er Object (eller arver fra Object).

- "..." er String, som arver fra Object
- new Object() er Object
- 123 er int, som autoboxes til Integer, som arver fra Object

1c

Suppose now that we replace the first line of the program fragment from **1b** with the following code:

```
MyStack<Object> myStack = new MyStack<String>(10);
```

Explain how the fragment now differs from the one in **1b**. Will the compiler react differently to it? If it compiles correctly, will the runtime behavior of the fragment be different?

MyStack<Object> og MyStack<String> er ikke kompatible typer, selv om Object er en superklasse av String. Koden vil ikke lenger kompilere.

1d Returning now to persons and students, assume that we want to write a method that can process stacks of persons and stacks of students, e.g. like this:

```
public static < … > void processPersons(... persons) {
   while(!persons.isEmpty()) {
       System.out.println(persons.pop().name);
   }
}
```

Replace the ellipses (…) two places in the method signature above with the appropriate generic declarations to make the following calls to `processPersons` work:

```
MyStack<Person> persons =   // some initialization code here,
MyStack<Student> students = // you do not need to provide this code

processPersons(persons);  // this call and the next should work
processPersons(students);
```

```
public static <T extends Person> void processPersons(MyStack<T> persons) {
```

1e

Java 8 allows the usage of anonymous functions (or "lambdas"). For instance, we could define `processPersons` to have an argument that is a predicate filtering which person's `name` to print:

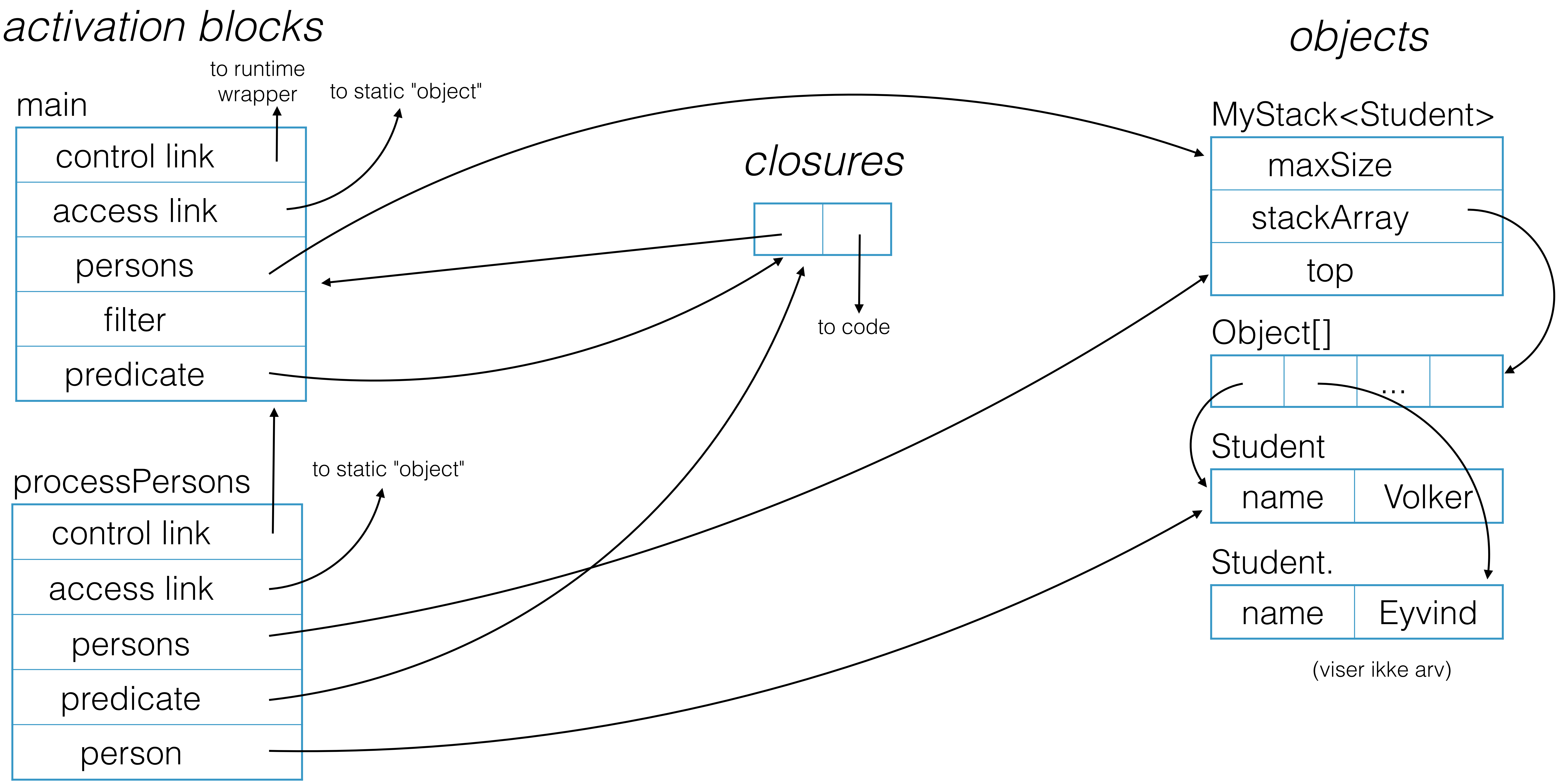
```
public static void processPersons(Stack<Person> persons,
                                  Predicate<Person> predicate) {
    while(!persons.isEmpty()) {
        Person person = persons.pop();
        if(predicate.test(person))
            System.out.println(person.name); // HERE
    }
}
```

`Predicate<T>`, as used in the method above, is a built-in interface in Java 8 that has a Boolean method `test` that can be implemented by an anonymous function. Thus, the `processPersons` method could now be called for instance like the following, to only print "`Volker`":

```
public static void main(String[] args) {
   MyStack<Person> persons = new MyStack<Person>(10);
   persons.push(new Student("Volker"));
   persons.push(new Student("Eyvind"));
   String filter = "V";

   Predicate<Person> predicate = p -> p.name.startsWith(filter);
   processPersons(person, predicate);
}
```

Draw the call stack as it is in the call to `processPersons` when the execution has reached the point marked "// `HERE`" in the code above.

1e


activation blocks
main
control link
access link
persons
filter
predicate
to runtime wrapper
to static "object"
processPersons
control link
access link
persons
predicate
person
to static "object"
closures
to code
objects
MyStack<Student>
maxSize
stackArray
top
Object[]
...
Student
name
Volker
Student.
name
Eyvind
(viser ikke arv)

2a

Evaluate the following ML expressions:

a) `(fn (x,y) => (x+1)::(y@[1])) (2,[3,4]);`

[3, 3, 4, 1]

b) `List.filter (fn x => x > 3) (map (fn x => x+2) [4,1,5,2]);`
where `List.filter = fn : ('a -> bool) -> 'a list -> 'a list` preserves only list elements that satisfy the predicate.

[6, 7, 4]

2b

Assume the standard definition of `foldl`:

```
fun foldl (f: 'a*'b->'b) (acc: 'b) (l: 'a list): 'b =
  case l of
    [] => acc
  | x::xs => foldl f (f(x,acc)) xs
```

With the help of `foldl`, define the function

```
val last = fn : 'a list -> 'a option
```

which returns `NONE` for the empty list, or the last element of the list wrapped in `SOME`.

```
fun last ls = foldl (fn (x, y) => SOME x) NONE ls;
```

Det jeg selv skrev i fjor (funker ikke i SML):

```
val last = foldl (fn (x, _) => SOME x) NONE;
```

2c

In ML, we can define a lookup table as a function from keys to values, using ML's `option` datatype. `NONE` indicates that no element with that key could be found, and `SOME` will be used in the case of reporting an existing entry in the table:

```
type ('k , 'v) table = 'k -> 'v option;
```

1) Define the constant value for the empty table, that is, the table, which will return `NONE` for any key!

```
val emptyT : ('k,'v) table =
```

```
val emptyT : ('k, 'v) table = (fn _ => NONE);
```

2c

In ML, we can define a lookup table as a function from keys to values, using ML's `option` datatype. `NONE` indicates that no element with that key could be found, and `SOME` will be used in the case of reporting an existing entry in the table:

2) Define the function

```
val addT = fn : (('k, 'v) table) -> ('k * 'v) -> (('k,'v) table)
```

which takes as first argument a table `t`, as second argument a key/value pair `(k,v)`, and returns a new table modeled as a function that, when asked for the key `k` that was just added, returns the value `v`, or looks for the key in table `t` otherwise!

[Note that for technical reasons related to polymorphic use of the equality-test, which you should use to compare keys, if you were to try this out in the interpreter, it is not possible to manually annotate the above type on the function. Type inference will still find the correct type, though of course not use the type synonym declared above:

```
val addT = fn : ('a -> 'b option) -> ('a * 'b) -> 'a -> 'b option
```

However, this does not affect your solution.]

```
fun addT t (k,v) =
```

```
fun addT t (k, v) =
  fn key => if k = key
            then SOME v
            else t key;
```

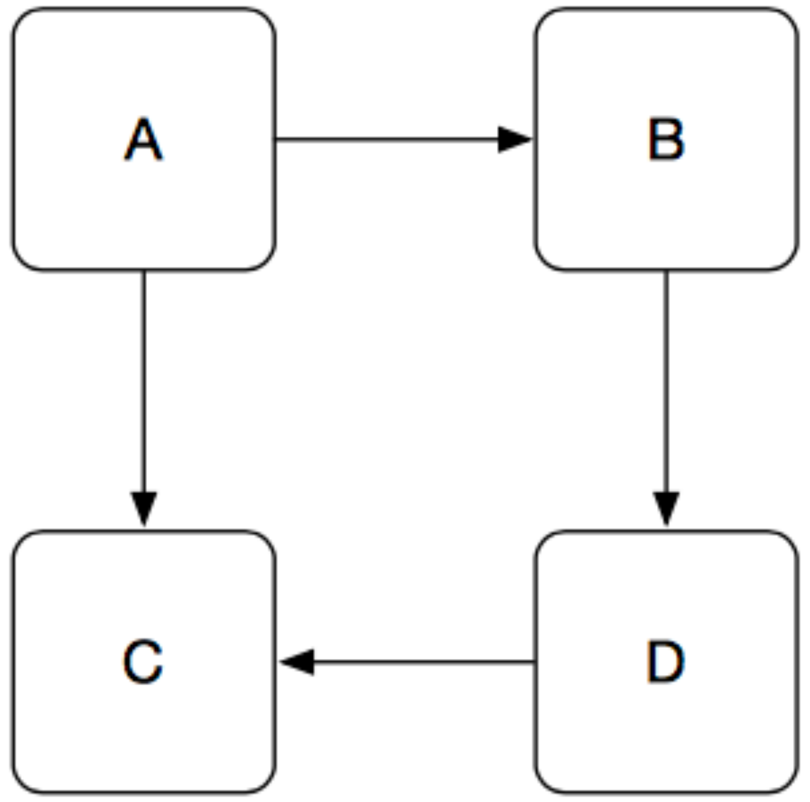
2d We define our own datatype to represent directed graphs as a sequence of vertices (edges) between nodes with the following declaration:

```
datatype 'a graph = Empty
                  | Vertex of ('a * 'a * 'a graph);
```

For example, the expression

```
      Vertex ("a", "b",
              Vertex ("a", "c",
                 Vertex ("b", "d",

                    Vertex ("d", "c", Empty))))
```

represents the following graph:



Write the function

```
val path = fn : 'a -> 'a -> 'a graph -> bool
```

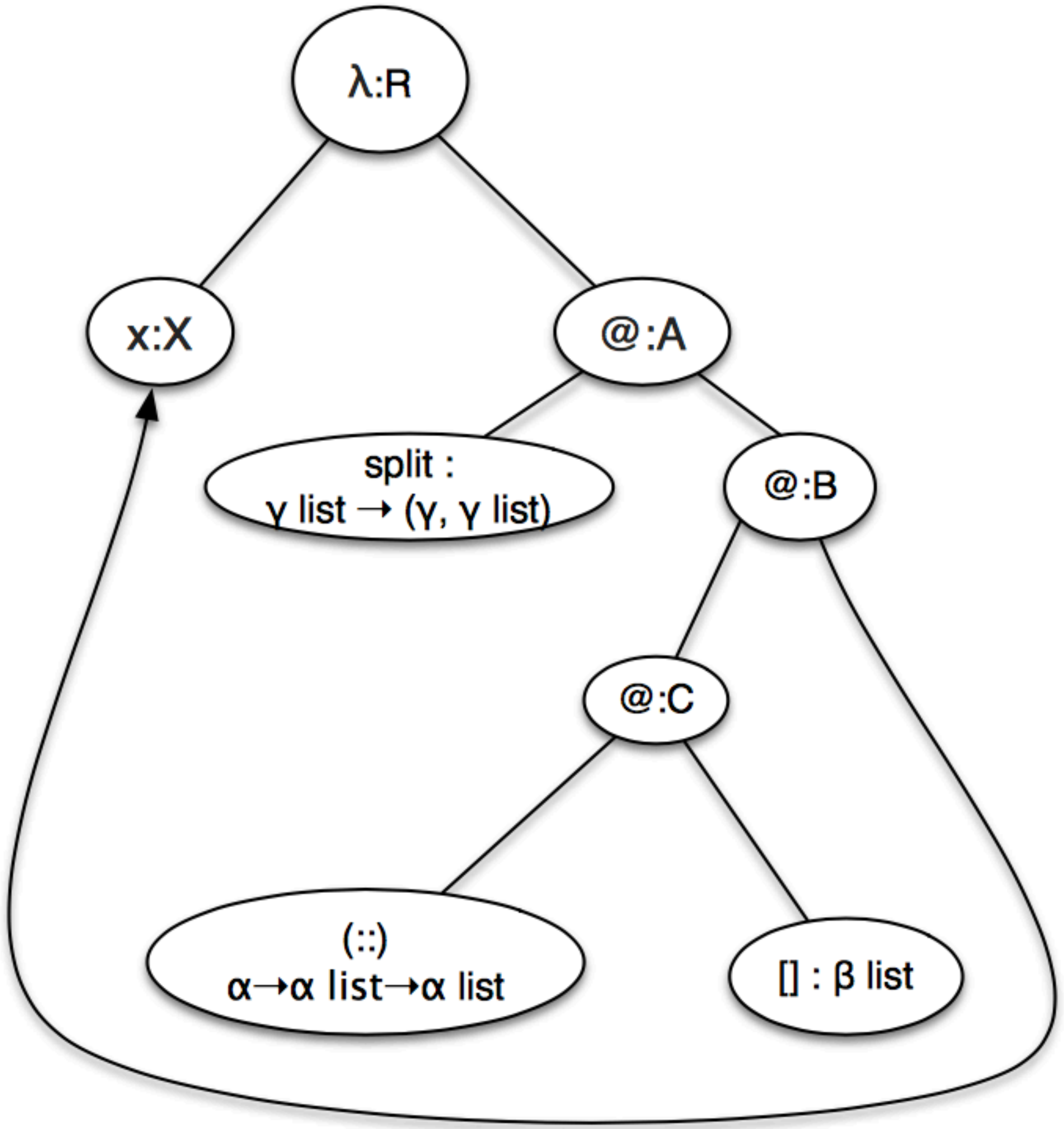
which returns true given two nodes `x`, `y` and a graph `g`, if `x = y`, or if there exists a path from `x` to `y` in `g`. You may assume that all graphs are *acyclic* (that is, they do not contain cycles).

```
fun path x y g =
  if x = y
  then true
  else case g of Empty => false
               | Vertex (f, t, rest) =>
    (x = f andalso y = t)
    orelse (path x y rest)
    orelse (x = f andalso (path t y rest));
```

2e Calculate the type for the following expression according to the ML type inference algorithm:

```
fn x => split ([] :: x).
```

Assume `val split = fn : 'a list -> 'a * 'a list` as given, and the annotated types in the parse graph. Use the provided type variables. Derive the corresponding equations for the parse graph, and solve the resulting equation system to obtain the type of the root node R.



$R = X \to A$

$\gamma\ list \to (\gamma, \gamma\ list) = B \to A$

$C = X \to B$

$\alpha \to \alpha\ list \to \alpha\ list = \beta\ list \to C$

$C = \alpha\ list \to \alpha\ list$

$X = \alpha\ list$

$B = \alpha\ list = \gamma\ list$

$A = (\gamma, \gamma\ list) = (\alpha, \alpha\ list)$

$R = \alpha\ list \to (\alpha, \alpha\ list)$

$\alpha = \beta\ list$

$R = \beta\ list\ list \to (\beta\ list, \beta\ list\ list)$

R = a' list list → (a' list * a' list list)

3a

Give PROLOG's answer (that is, the substitutions for all variables in a query, if the terms can be unified) for each of the queries below, or simply write "no" if no solution exists.

1. `[X|[a,b]] = [Y,a,B].`

   X = Y
   B = b

2. `f(g(0),X) = f(X,g(Y)).`

   X = g(0)
   Y = 0

3. `z(k(0),X) = z(k(X),s(0)).`

   no

4. `c(X,Y) = c(Z,Z).`

   Y = X
   Z = X

3b Given the following persons that each have a name, a mother, a father and a birthday, `person(a,b,c,d)` denotes a person with name `a`, mother `b`, father `c`, and year of birth `d`.

For example:

```
person(anne, sofia, martin, 1960).
person(john, sofia, george, 1965).
person(paul, sofia, martin, 1962).
person(maria, anne, mike, 1989).
```

1. Define a predicate `parents(x,y)`, that is true if `x` and `y` have a child together.
2. Define a predicate `itscomplicated(x)`, that is true if `x` has children with more than one partner.

```
parents(X, Y) :- person(_, X, Y, _).
parents(X ,Y) :- person(_, Y, X, _).

itscomplicated(X) :- parents(X, Y),
                     parents(X, Z),
                     Y /== Z.
```

3c We define a data structure for trees that store values in their nodes in the following way:

`empty` denotes the empty tree. `node(L,V,R)` denotes a tree with value `V` in the node, and left and right sub-trees `L,R`.

1. Define a predicate `depth(T,N)` that is true if the tree `T` has depth `N`. The empty tree has depth 0, and a `node` has depth `1+M`, where `M` is the maximum of the depths of the two subtrees.

```
max(A, B, A) :- A >= B.
max(A, B, B) :- B > A.
```

```
depth(empty, 0).
depth(node(L,_,R), N) :- depth(L, X),
                         depth(R, Y),
                         max(X, Y, M),
                         N is M + 1.
```

3c We define a data structure for trees that store values in their nodes in the following way:

`empty` denotes the empty tree. `node(L,V,R)` denotes a tree with value `V` in the node, and left and right sub-trees `L,R`.

2. Define a predicate `heap(H,N)`, that is true if the tree `H` has the shape of a heap with depth `N`:
   - The empty tree is a heap with depth 0.
   - A node `node(L,V,R)` is a heap, if
     - `L` and `R` are heaps, and
     - if `L` and/or `R` are not empty, then the values at their top are less than or equal to `V`, and
     - If `L` has depth `D`, then `R` has depth `D` or `D - 1`
     - The depth `N` is defined as the maximum depth of the two branches + 1.

Note that through adequate use of the second argument of `heap` you do not actually have to use the predicate defined in part 1)!

3c

We define a data structure for trees that store values in their nodes in the following way:

`empty` denotes the empty tree. `node(L,V,R)` denotes a tree with value `V` in the node, and left and right sub-trees `L`, `R`.

2. Define a predicate `heap(H,N)`, that is true if the tree `H` has the shape of a heap with depth `N`:
   - The empty tree is a heap with depth 0.
   - A node `node(L,V,R)` is a heap, if
     - `L` and `R` are heaps, and
     - if `L` and/or `R` are not empty, then the values at their top are less than or equal to `V`, and
     - If `L` has depth `D`, then `R` has depth `D` or `D - 1`
     - The depth `N` is defined as the maximum depth of the two branches + 1.

```
heap(empty, 0).
heap(node(empty, _, empty), 1).
heap(node(node(LL,LV,LR), V, empty), 2) :-
  heap(node(LL,LV,LR), 1),
  LV =< V.
heap(node(node(LL, LV, LR), V, node(RL,RV,RR)), N) :-
  heap(node(LL, LV, LR), LD),
  heap(node(RL, RV, RR), RD),
  LV =< V,
  RV =< V,
  DD is LD - RD,
  (DD =:= 0 ; DD =:= 1),
  N is LD + 1.
```