Name_____

CS471, Programming Language Structure I
Spring, 2003

Final Examination

This is a take-home exam. You may work with others in the class on all or part of the exam., but you *must* indicate your helpers' names on your answers. If you do not, and there is evidence of copying of answers, you may receive zero for the whole exam. The total points for each question or part of a question follows it in parentheses, thus: *(12 pts)*

1

Consider the Pascal program below.

a. State the rule for setting the static pointer of an activation record to point to the correct static ancestor record on the stack. *(5 pts)*

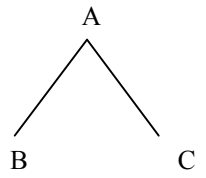
b. Consider the Pascal code fragment below. Draw a scope diagram, and a calling diagram for the program. *(5 pts)*

c. Draw a stack diagram just before execution of the print statement in B takes place. The activation records should show, at least, variables, parameters and static and dynamic pointers. Explain, in detail, how the static pointer is set for every activation record on the stack. *(15 pts)*

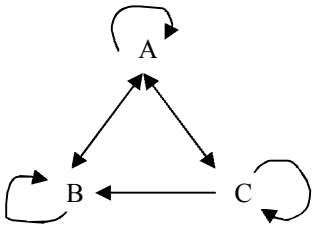
```
procedure A;
  var x : integer;
  procedure B;
  begin
    print(x)
  end;
  procedure C(y : integer);
    var x : integer;
  begin
    x := y + 1;
    if y < 3 then
      C(y + 1)
    else
    begin
      B
    end
  end;
begin
  x := 0;
  C(1)
end;
```

a. Subtract the static depth of the callee's static parent from the static depth of the caller. Take this many hops up the static pointer chain from the caller's activation record.

b.

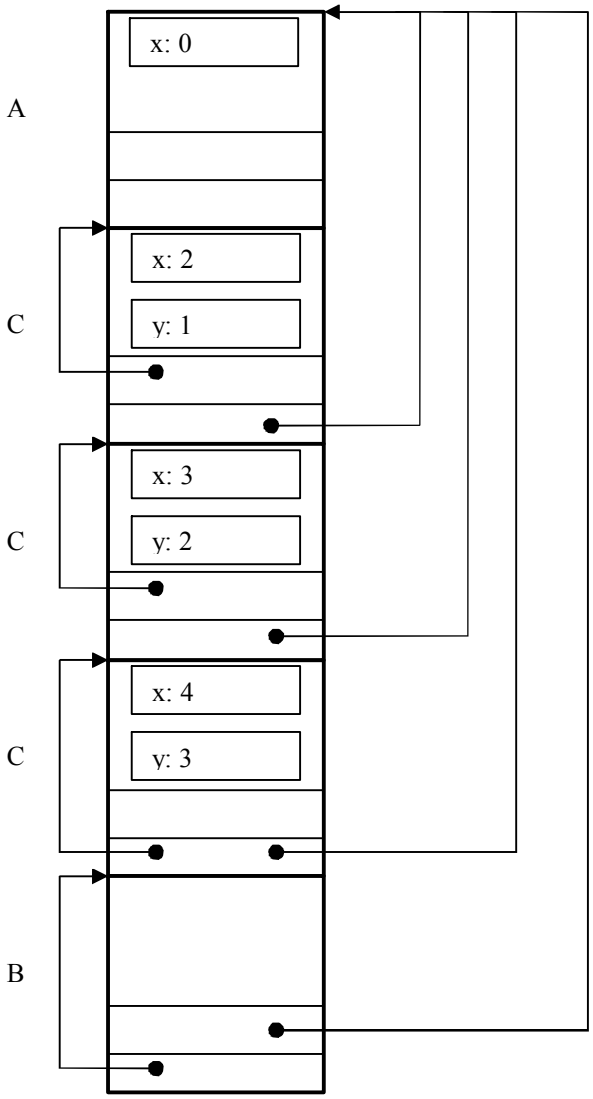


Scope diagram



Calling diagram

c.



1. **A's activation record is placed on the stack when A is called. It has has undetermined pointers.**
2. **A calls C. C's dynamic pointer points to A. Number of hops is static depth of callee's static parent, A minus callers depth. Since these are the same (C's static parent is A, and the caller is A), this is zero, so C's static pointer points to A as well.**
3. **C calls itself. Now the caller's depth is n+1 (A's depth is n), and the callee's static parent's depth is n, so the difference is 1. The new C's static pointer is ths ame as the pervious one, ie.e it points to A.**
4. **The same thing happens with the next two calls, i.e. each successive call to C uses it's caller's static pointer to A.**
5. **When C calls B, the depth of the caller is n+1 and the depth of the callee's static parent is n, so again the difference is 1. B's static pointer then points to A as well, since it is taken from C's copy.**

2

a. Java has reference semantics for objects. Explain this statement by answering the questions put in comments in the program below. *(2.5 pts each)*

```
public class Box {
  private int value;
  public Box() { value = 0; }
  public void setValue(int v) { value = v; }
  public int getValue() { return value; }
}

public class BoxTest {
  static public void main(String [] args) {
    Box x = new Box();                  // what happens here?
    x.setValue(7);                      // what happens here?
    Box y = x;                          // what happens here?
    y.setValue(11);                     // what happens here?
    System.out.println(x.getValue());   // what is printed?
    System.out.println(y.getValue());   // what is printed?
  }
}
```

1. **an object of type Box is created on the heap and a reference to it is placed in x.**
2. **value inside x is set to 7**
3. **variable y contains a copy of the reference to x**
4. **since y and x points to the same object, the 7 is overwritten with 11**
5. **both prints produce 11**

b. If the following method is added to class Box:

```
public Object copy() {
  Box b = new Box();
  b.setValue(getValue());
  return b;
}
```

and the line in BoxTest in boldface (**`Box y = x;`**) is changed to:

```
Box y = (Box)x.copy();
```

what does the main method in BoxTest now print? *(10 pts)* [Note that all classes in Java implicitly extend Object if none is mentioned.]

Since y now contains a reference to a copy of the object referred to by x, y.setValue(11) sets value to 11 inside y, elaving the 7 inside x untouched. The prints now produce 7 and 11, respectively.

3

Consider the following Scheme function:

```
(define (mystery f L)
   (cond ((null? L) '())
         ((f (head L)) (mystery f (tail L)))
         (else (cons (head L) (mystery f (tail L))))))
```

a. Write out a derivation (a trace) of the call:

`(mystery (lambda (x) (= (modulo x 2) 1)) '(1 1 2 3 5))` *(20 pts)* [modulo is the standard arithmetic function ('remainder') on integers]

Let m = (lambda (x) (= (modulo x 2) 1))

(mystery m '(1 1 2 3 5))

= (mystery m '(1 2 3 5)) since (m 1) is true

= (mystery m '(2 3 5)) since (m 1) is true

= (cons 2 (mystery '(3 5))) since (m 2) is false

= (cons 2 (mystery '(5))) since (m 3) is true

= (cons 2 (mystery '())) since (m 5) is true

= (cons 2 '()) since the list is now empty

= (2)

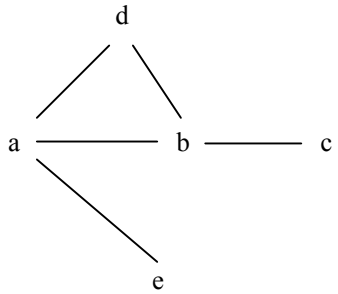
b. In a sentence, what does the function mystery do? *(5 pts)*.

Function mystery filters the list by removing items that return true when passed to th function f.

4

A graph consisting of nodes with links between them ("edges") can be described in Prolog with a set of facts about these edges. For instance the following set of facts represents the graph on the right:

```
edge(c,b).
edge(b,d).
edge(b,a).
edge(a,d).
edge(a,e).
```



A path from one node to another, following the links, may be defined by:

```
path(S,F) :- edge(S,F).
```

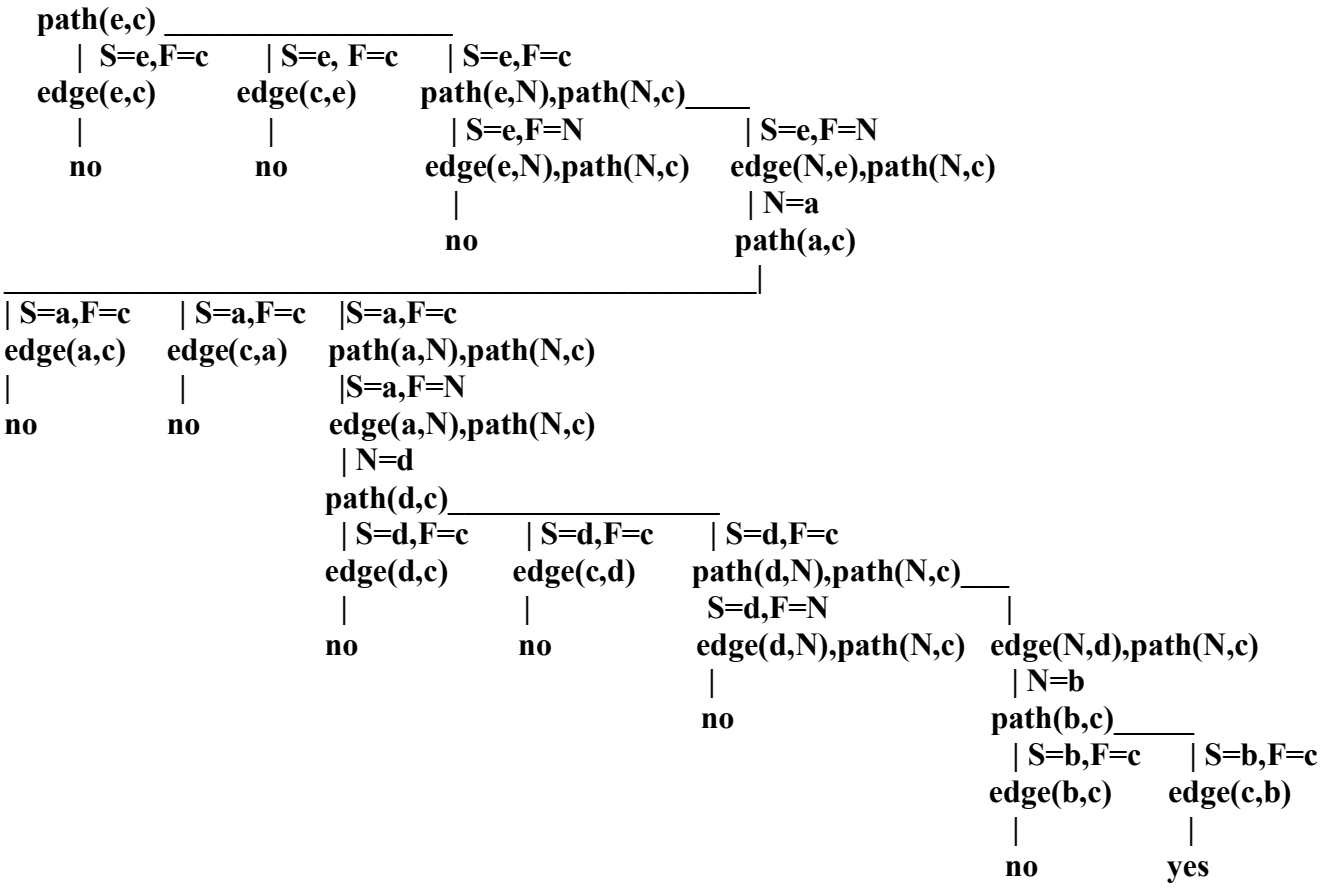
```
path(S,F) :- edge(F,S).
path(S,F) :- path(S,N), path(N,F).
```

a. Show, by drawing a goal tree, that Prolog can verify that there is a path from e to c. *(15 pts)* i.e. use the query:

```
?- path(e,c).
```

b. The reverse query : `path(c,e)` has a problem. What is it? *(10 pts)*.

BONUS: How might the problem be fixed? *(5 pts)*.



The reverse query, path(c,e) will find the edge(c,b) and then try path(b,e). The first edge found is (b,d), leading to path(d,e). Because edge(b,d) comes before edge(a,d) in the facts, the path will be retraced to b, leading to path(b,e) which has already been tried. This is a recursive loop which will never succeed.

One way to correct this is to test for recursive loops, by passing all the nodes visited along the path down to every subgoal. This would bypass b, finding a, and completing the path.