CSE130 Programming Languages, Final Exam

Tue Mar 21, 2000, 7:00 - 10:00 PM (3 hours)

1. (**2 points**) Consider the type `int->int`. In how many ways can you implement this type in C? Give examples. In how many ways can you implement the type `float->float` in C?

   `int->int` can be implemented in two ways: as an array of ints and as a function.

   `float->float` can be implemented only as a function.

2. (**1 point**) Consider the following recursive function.

   ```
       f(x) = if x>100 then x-10
                       else f(f(x+11))
   ```

   Trace the first four steps in the outermost evaluation of `f(100)`.

   ```
   f(100) = if 100>100 then 100-10 else f(f(100+11))
          = f(f(100+11))
          = if f(100+11) > 100 then f(100+11)-10 else f(f(f(100+11)+11))
      [evaluation of f(100+11)=
      f(100+11) = if 100+11 > 100 then 100+11-10 else f(f(100+11+11))
                = if 111>100 then ...
                = ...
   ```

3. (**3 points**) Consider the following Pascal program.

   ```
   program L(input, output);
   var n, m: integer;

   procedure p1(i:integer);
     var n: integer;
     procedure p2(i:integer);
     begin
       if (i<2) then p2(i+1)
       else
         begin
            n := n+m+i+3;
         end;
     end;

   begin
     n := i; m := m + 1;
   ```

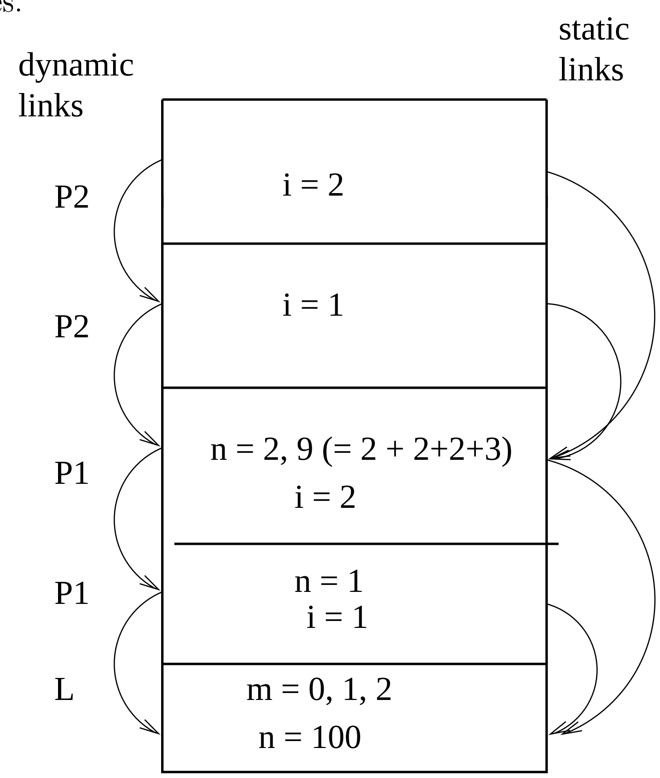
```
  if (i<2) then p1(i+1)
  else
    begin
      p2(1);
      writeln(n);
    end
end;

begin
  n := 100; m := 0;
  p1(1);
end.
```

(a) Show the bindings of the names `m` and `n` in `L`, `p1`, and `p2`.

```
m, n in L = declaration in line 2
m in p1 = m in L
n in p1 = local
m in p2 = m in L
n in p2 = n in p1
```

(b) Draw the runtime stack and show the static and dynamic links for all the activation frames.



(c) What is the output of the program?

9

4. (**1 point**) The following shows a program fragment in C syntax.

```
void swap(int x, int y) {
   int tmp;

   tmp = x;
   x = y;
   y = tmp;
}

int main(void) {
   int i, a[10];

   i = 2; a[i] = 30;
   swap(i, a[i]);
}
```

(a) Trace the program execution assuming that the parameters are passed by reference.

`a[2]` becomes 2 and `i` becomes 30.

(b) Trace the program execution assuming that the parameters are passed by name.

`i` becomes 30 and `a[30]` becomes 2. `a[2]` is unchanged.

(Note: C supports neither call by reference nor call by value.)

5. (**1 point**) C language does not support mutual exclusion of accesses to shared variables. One way of achieving mutual exclusion in C is to use libraries. Java has language support for mutual exclusion. Discuss the merits of each approach.

Advantages of Java: Compiler can check for matching lock and release of semaphores.

Advantages of C: Only those programs which need this feature pay the cost of maintaining semaphores.

6. (**1 point**) Trace the flow of the following Java program. What is the output of the program?

```
class ExceptionA extends Exception {
}
class ExceptionB extends Exception {
}
```

```
class ExceptionC extends Exception {
}
class Excep {
    static void throwThrow() throws ExceptionA, ExceptionB, ExceptionC {
       try {
          throw new ExceptionA();
       } catch(Exception e) {
          throw new ExceptionB();
       } finally {
          throw new ExceptionC();
       }
    }

    public static void main(String[] args) {
        try {
          Excep.throwThrow();
        } catch (Exception e) {
          System.out.println(e);
        }
    }
}
```

The output of the program is `Exception C`.

7. (**1 point**) Recall the SiteManager class you developed for the third assignment. SiteManager provides a remote method `stay()`. Suppose the SiteManager wants to keep track of the number of times this method was used. One possible implementation is

```
class SiteManger extends UnicastRemoteObject implements Site {
   static private int numVisitor=0;
           // num visitor is the number of visitors to the site so far

   public synchronized Object stay(Task task) {
      numVisitor++; // one more visitor!
         /* rest of the stuff */
   }
}
```

(a) Why should accesses to `numVisitor` be synchronized?

If a number of clients invoke the method at the same time, multiple threads are created. Each tries to access `numVisitor` and these accesses should be synchronized.

(b) What is the problem with the synchronization used in the above code?

In the above implementation, the entire `stay` method is synchronized. This implies that only one client can execute the method at a time. All the other requests are blocked. It is better to implement synchronization as

```
synchronized (this) {
    numVisitor++;
}
```

8. (**1 point**) Consider the Prolog program

```
person(socrates).
mortal(X) :- person(X).
mortal(micky).
```

What are the results of the queries

(a) `mortal(socrates).` The result is `yes`

(b) `person(micky).` The result is `no`

Why?

9. (**2 points**) Consider the Prolog program

```
append([], Y, Y).
append([H|X], Y, [H|Z]) :- append(X, Y, Z).
```

Trace the execution of the query `append(X,[3],[1,2,3]).`

```
append(X, [3], [1, 2, 3])
   = append([1|X'], [3], [1|2,3]) :- append(X', [3], [2,3]).
append(X', [3], [2,3])
   = append([2|X''], [3], [2|3]) :- append(X'', [3], [3]).
append(X'', [3], [3])
   = append([], [3], [3]) => X'' = []
                          => X' = [2|[]] = [2]
                          => X = [1|[2]] = [1, 2]
```

10. (**4 points**) Fill the answers in the twelve blanks in the following table. The acceptable answers for times are: "load time", "block entry time", "block exit time", "program termination", "any time", or "other". For accessible locations, you should use: "anywhere except where masked", "in the declaring block", or "other". If you choose "other", you should explain its meaning.

Type of variable	time of creation	time of death	where it is accessible
global variable	load time	program termination	anywhere except where masked
static local variable	load time	program termination	in the declaring block
ordinary local variable	block entry time	block exit time	in the declaring block
dynamic heap object	any time	any time	anywhere if a reference is available

11. (**1 point**) Consider the following ML function for finding the length of a list.

```
fun length(nil) = 0
  | length(_::rest) = 1 + length(rest);
```

Is this function tail recursive? Why? If not, write a tail recursive function which finds the length of a list.

This function is not tail recursive. A tail recursive version is

```
fun length(nil, n) = n
  | length(_::rest, n) = length(rest, n+1);
```

The function call is: `length([1,2,3], 0)`.

12. (**2 points**) Trace the execution of the following ML program.

```
fun plus1 x = x + 1;
fun m x y:int = x*y;
fun mix f g = fn x => g (f x) (f x+3);
val a = mix plus1 m;
a 9;
```

The following is an actual trace.

```
val plus1 = fn : int -> int
val m = fn : int -> int -> int
val mix = fn : ('a -> int) -> (int -> int -> 'b) -> 'a -> 'b
val a = fn : int -> int
val it = 130 : int
val it = () : unit
```

13. (**2 points**) You want to define a datatype *stream* consisting of a possibly infinite sequence of values. An example of a stream is a = *(5 6 7 8 ...)*. Note that the stream a is potentially infinite. Formally, a stream supports three operations described below:

    (a) *head(s)*: return the first element of a stream s.

(b) *tail(s)*: return the rest of s after the first element.

(c) *scons(a, s)*: return a new stream type with head a and tail s. *scons* is the constructor for streams.

For the stream a, we have *head(a)* = *5*, *tail(a)* = *(6 7 8 9 ...)*, and *scons(0, a)* = *(0 5 6 7 ...)*.

The issue here is: how to define a potentially infinite object? One technique needed is recursion. One *possible implementation* of a stream of integers starting with integer n in ML is

```
datatype stream = S of int * stream;
fun scons(n) = S(n, scons(n+1));
fun head(S(a,b)) = a;
fun tail(S(a,b)) = b;
```

But recursion alone is not sufficient.

(a) Does the above definition work with ML evaluation model? If not, why not?

The definition does not work with the ML evaluation model. It causes infinite recursion when the constructor is called.

(b) What language feature is required to make this definition work?

Lazy evaluation of the function `scons` is required to make this work.

14. (**1 point**) Consider the following C++ program.

```
class X {
   int i;
};
class Y : public X {
   int j;
};
class Z : public X {
   float k;
};
class W : public Y, public Z {
   int z;
};
int main(void) {
   W w;
}
```

(a) How many times is the constructor of each class called during the execution of this program? Why?

The constructor of X is called twice, X once, Y once, and W once.

(b) Where is object w allocated?

w is allocated on the stack.

15. (**1 point**) Consider the following C++ program.

```
class X {
   private:
      int x;
   public:
   X() { x = 10; }
};

class Y {
   public:
      int y;
};

int main() {
   X x;
   Y *y = (Y*) &x;
   cout << y->y << endl;
}
```

This program compiles, runs, and prints 10. Assume that classes X and Y were written by different programmers. What is the motivation for class Y? What aspect of C++ does this illustrate?

This is an "attack" on the security provided by the class system. Private fields are not accessible outside the class. If the data layout of the class is known, one can set a pointer to it and retrieve private values.

This illustrates the danger of mixing too many paradigms: procedural and oop, in this case.

16. (**1 point**) Discuss the merits of compile time and runtime checks. The advantages of compile time checks are

(a) speed of checking

(b) some guarantee of correctness before the program is run

The advantages of runtime checks are

(a) Some errors (array bounds, input values, etc.) can be caught only at runtime

(b) security

17. (**1 point**) Consider the following C++ program.

```
int main(void) {
   cout << "Hello world\n";
}
```

How can you make the program print

```
Initialize
Hello world
Clean up
```

by adding additional C++ code but without changing `main`?

Add the following code before `main`.

```
class X {
   X() { cout << "Initialize\n"; }
   ~X() { cout << "Clean up\n"; }
}

X x;
```

18. (**1 point**) When a local method is called, the arguments are pushed onto the stack and when the method returns, the result is popped out of the stack. What are the operations during the call and return in a remote method invocation?

The operations during call are: serialization, marshaling the arguments, and a network call.

The operations during return are: unmarshaling and deserialization.

19. (**1 point**) Consider the following Java program.

```
class X {
   void foo() {
     System.out.println("Class X");
   }
}

class Y extends X {
   void foo() {
```

```
        System.out.println("Class Y");
    }
}

class Z {
    X x;
    /* other stuff */
    void foo() {
        x.foo();
    }
}
```

Suppose `z` is an instance of class `Z`. What is the output of the method invocation `z.foo()`? Why?

The output of `z.foo()` can be either `Class X` or `Class Y`. Because of polymorphism, `x` can reference class `X` or any of its subclasses.

20. (**2 points**)

   (a) What is the motivation for supporting the definition generic types?

   The main motivation for supporting generic types is to facilitate writing of container classes: classes which contain other classes as members. Examples of container classes are: stacks, queues, binary trees, etc.

   (b) Consider the following C++ code.

```
template <class T>
class test {
   T *a, b[10];

public:
   test() { return a/b; }
   T bar() { return a++b; }
   T foo() { return b[300]; }
};
```

   How many errors does it contain? When are the errors detected?

   This contains four errors:

   i. `a/b`: This is detected when the class is instantiated.
   ii. constructor with `return`: This is detected when the class is instantiated.
   iii. `a++b`: This is detected when the member function `foo` is used
   iv. `b[300]`: The size of `b` is 10. This is detected at runtime.