Programming Languages

Winter 1992

1 hour, 60 points

1. (*15 points*) When designing a statically-scoped language with functions, the designer has three options:

   (a) Functions can only be called.

   (b) Functions can be passed as arguments to other functions.

   (c) Functions can be returned as results of other functions.

   Discuss the tradeoffs between expressiveness and implementation complexity in each option. Give a short code segment to illustrate the expressiveness of options (b) and (c).

   How does the C language avoid making this tradeoff?

2. (*23 points*) Structure assignment in C^{++} was originally implemented as bit-wise copy. More specifically, if type `pair` is declared by

```
struct pair {
   int x;
   int y;
};
```

   and we have pairs `a` and `b`, the assignment `a` = `b` is implemented by bitwise copy of `b` onto `a`. Simple bitwise copy was later replaced by recursive bitwise copy, but the distinction between the two will not matter for the purpose of this question.

   (a) (*3 points*) Suppose that after declaring `pair` as above, we declare `triple` using `pair` as a public base class (`struct` is the special case of `class` with all members public).

```
struct triple : public pair {
  int z;
};

pair p;
triple t;

t.x = 1;
t.y = 2;
t.z = 3;

p = t;
```

   The fourth assignment above is type correct, for a slightly subtle reason that you need not explain. What will be the value of `p.x` and `p.y` after this assignment?

   (b) (*7 points*) The situation becomes more complicated when we have objects with virtual methods. Consider the two classes and two objects declared below. The first class is a class of points, with private `x` and `y` and public functions for setting and reading `x` and

y. The move function on points just moves the point in the x direction. Objects of the second class, "directed" points, have an additional private "direction" dir. In addition to public functions for setting and reading dir, directed points have a move function that moves them at an angle determined by dir. You do not need to examine the code any of the member functions whose names begin with set or read to do this problem. Object p is a point and q is a directed point, both initialized to position $(0, 0)$.

```
class pt {
  int x;
  int y;
public:
  pt(int xval, int yval) {x = xval; y = yval;};
  void setx(int newx) {x = newx;};
  void sety(int newy) {y = newy;};
  int readx () {return x;};
  int ready () {return y;};
  virtual void move (float dif){ x = x + (int)(dif+0.1);};
};
class dp : public pt {
  float dir;
public:
  dp(int xval, int yval, float dval) : (xval, yval) {dir=dval;};
  void setd(float newd) {dir = newd;};
  float readd () {return dir;};
  void move(float dif){
     setx((int)( (float)(readx()) + dir*dif + 0.5 ));
     sety((int)( (float)(ready()) + (1.0-dir)*dif + 0.5 ));};
};
pt p(0,0);       /* p is a pt with x=0, y=0 */
dp q(0,0,0.5);   /* q is a dp with x=0, y=0, dir=0.5 */
```

After executing statements p.move(1.0) and q.move(2.0) what will be the values of the private data of p and q? After doing an assignment p = q, what will be the effect of p.move(1.0)? What alternate behavior might the naive programmer expect from p.move(1.0) after assigning q to p? From an implementation point of view, why can't assignment to p change the way that p.move works?

(c) (*5 points*) Suppose we use pointers to objects instead. More specifically, we declare pointers pp and qq by

```
pt* pp = new pt(0,0);
dp* qq = new dp(0,0,0.5);
```

and then do a pointer assignment pp = qq. What will be the effect of pp− > move(2.0) after this assignment? Explain. (*Hint:* This is different from the effect of p.move in part (b).)

(d) (*8 points*) There are many general reasons that programmers might have for using object-oriented programming languages. Use one such commonly-cited, general reason to explain why most C^{++} programmers manipulate objects through pointers. Your answer should have something to do with other parts of this question.

3. (*22 points*) The concept of *future* was popularized by R. Halstead's article on the language Multilisp for concurrent lisp programming. Operationally, a future consists of a location in memory and a process that is intended to place a value in this location at some time "in the future." More specifically, the evaluation of `(future e)` proceeds as follows:

   (a) An empty cell is allocated and returned immediately.

   (b) A process is created to evaluate `e`.

   (c) When the process evaluating `e` completes, the value of `e` is written into the cell.

   (d) The process that invoked `(future e)` continues in parallel with the new process. If this process tries to read the contents of the cell while it is still empty, the process blocks until the cell is filled.

   For example, if `e` evaluates to the list `(1 2 3)`, then the expression

   `(cons 'a (future e))`

   produces a list whose first element is the atom `'a`, and whose tail becomes `(1 2 3)` when the second process evaluating `e` is completed. The advantage of this construct is that if the list produced by `(cons 'a (future e))` is passed to function that takes the `car` (first element) of the list, then this can be done in parallel with the evaluation of `e`. However, if the program tries to examine the second element of the list before the value of `e` has been placed in the empty cell, then the computation will block until the data is ready.

   (a) (*6 points*) Assuming an unbounded number of processors, how much time would you expect the evaluation of the following `fib` function to take, on positive integer argument `n`?

   ```
   (defun fib (n)
        (cond ((eq n 0) 1) ((eq n 1) 1)
        (T  (plus (future (fib (minus n 1))
                  (future (fib (minus n 2))
   )))
   ```

   We are only interested in running time up to a multiplicative constant; you may use "big Oh" notation if you wish.

   (b) (*8 points*) What two properties of `e` will guarantee that an expression of the form `( ... (future e) ... )` will have the same value as the corresponding expression `( ... (e) ... )` without `future`? Do not be concerned with the efficiency of either computation, or the degree of parallelism.

(*continued on following page*)

(c) (*8 points*) You are part of a language design team that has adopted futures as an approach to concurrency. The head of the team suggests an error handling feature called a "try block." The syntactic form of a "try block" is

```
(try e (error-1 handler-1) (error-2 handler-2) ...)
```

This construct would have the following characteristics:

i. Errors are programmer-defined, and occur when an expression of the form `(raise error-1)` is evaluated inside `e`.

ii. If no errors occur, this expression is equivalent to evaluating `e`.

iii. If the error named by `'error-1'` occurs during evaluation of `e`, the rest of `e` is ignored, `handler-1` is immediately evaluated and its value becomes the return value of the try-block.

iv. Similarly for the error named by `'error-2'`, and so on.

The other members of the team think that this is a great idea and, claiming that it is completely straightforward, tell you that you should add this to the implementation by Friday. You think it might not be as easy or straightforward as they think. What problems or questions does this construct raise? Give short code examples or sketches to illustrate your point(s).

Note: this question does *not* ask you to solve the problems associated with `try` and `future`.