Comprehensive Exam: Programming Languages Autumn 2006

1. (*21 points*) .... Short Answer

   Answer each question in a few words or phrases.

   (a) (*3 points*) Why is memory usually separated into a stack and a heap?

   Answer: Local data structures used only inside one block can be allocated/deallocated more efficiently using the stack discipline. When the lifetime of an object exceeds the lifetime of the block where it is created, some other mechanism (user deallocation or garbage collection) must be used to deallocated that space.

   (b) (*3 points*) Why is it possible to implement C without creating closures?

   Answer: No C functions are declared in nested scopes. If a function is passed to another function, then the lexical environment of the function that is passed is the global environment of the function it is passed to. Similarly, if a function is returned from another function, its lexical environment is the global environment, not a local environment of the function from which it is returned.

   (c) (*3 points*) What is the difference between overloading and polymorphism?

   Answer: Overloading allows one symbol to have more than one meaning. Overloading is resolved at compile time. Polymorphism allows a single function to be applied to many different types of arguments. Although a polymorphism function may conceptually have many different types, the name of a polymorphic function is associated with one algorithm, not many.

   (d) (*3 points*) Explain the difference between subtyping and inheritance.

   Answer: Subtyping is a relation between interfaces (or types), while inheritance is a relation between implementations.

   (e) (*3 points*) If `A` <: `B` and `B` <: `C` is `A`→`B` <: `A`→`C` ? Why or why not?

   Answer: Yes, by covariance of function types, `B` <: `C` implies `A`→`B` <: `A`→`C`. If f is a function of type `A`→`B`, then $f(x)$ has type `B`. Since `B` <: `C`, the return value $f(x)$ can appear where a value of type `C` is required. Therefore `f` can be used when a function of type `A`→`C` is needed.

   (f) (*3 points*) Explain how the Java compiler treats this statement and, if the statement compiles, what happens at run time?

   ```
   Character c = (Character) new Object();
   ```

   Answer: This statement compiles, because the cast makes it type-correct at compile time. However, a run-time error occurs when the statement is executed.

   (g) (*3 points*) What property of the Java architecture makes it necessary to do bytecode rewriting to get efficient method lookup?

   Answer: Run-time loading and linking of separately compiled class files. More specifically, a class file that calls a method on objects of class `C` may be compiled without knowing what class `C` will look like at run time. Therefore, the class file cannot be compiled to use fixed offsets of methods in the method table of class `C`. However, after `C` is loaded, the bytecode can be rewritten to use a fixed offset.

2. (*9 points*) .... Pointer arithmetic

   Unlike most programming languages, C allows pointer arithmetic. This question asks you about advantages and disadvantages of this language feature, with some parts of the question referring to the following excerpt from a JPEG decoder that performs a discrete cosine transformation using a loop that iterates through an array of elements.

```
void jpeg_fdct_ifast (DCTELEM *data)
{
   ...
   DCTELEM *dataptr;
   int ctr;
   ...
   /* Pass 1: process rows. */
   dataptr = data;
   for (ctr = DCTSIZE-1; ctr>= 0; ctr--) {
      tmp0 = dataptr[0] + ... + dataptr[DCTSIZE-1];
      ...
      dataptr += DCTSIZE;  /* advance pointer to next row */
   }
   ...
```

(a) (*2 points*) Name two disadvantages of pointer arithmetic. In answering the question, consider compiler features, desirable language properties, or language run-time features that are difficult or impossible in languages with pointer arithmetic.

Answer: Type safety, array bounds checking, garbage collection.

(b) (*3 points*) One argument sometimes given in favor of pointer arithmetic is efficiency. In the code example above, each loop iteration processes one row of a DCTSIZE×DCTSIZE matrix whose entries are stored sequentially in memory. Iteration number `ctr` accesses the elements $\texttt{data}[\texttt{DCTSIZE} * ((\texttt{DCTSIZE} - \texttt{1}) - \texttt{ctr})]$ to $\texttt{data}[\texttt{DCTSIZE} * ((\texttt{DCTSIZE} - \texttt{1}) - \texttt{ctr}) + (\texttt{DCTSIZE} - \texttt{1})]$. Explain why this loop above might be more efficient than a loop that uses expressions of the form $\texttt{data}[\texttt{DCTSIZE} * ((\texttt{DCTSIZE} - \texttt{1}) - \texttt{ctr}) + \texttt{i}]$ to index into the data array.

Answer: No need for repeated addition and multiplication to compute the array index.

(c) (*2 points*) Suppose that the omitted code inside the loop accesses memory using $*(\texttt{dataptr} + \texttt{j})$, with `j` set by a `for` loop that only gives `j` values between 0 and $\texttt{DCTSIZE} - \texttt{1}$. Explain why all memory accessed by the loop is memory that is allocated to the program, assuming that the function parameter `data` points to a heap-allocated region of $\texttt{DCTSIZE} * \texttt{DCTSIZE}$ locations.

Answer: A simple calculation shows that for $\texttt{0} \leq \texttt{ctr} \leq \texttt{DCTSIZE} - \texttt{1}$ and $\texttt{0} \leq \texttt{j} \leq \texttt{DCTSIZE} - \texttt{1}$, we have $\texttt{dataptr} = \texttt{data} + (\texttt{DCTSIZE} - \texttt{1} - \texttt{ctr}) * \texttt{DCTSIZE}$ and therefore

$$\begin{aligned}\texttt{data} &\leq \texttt{data} + (\texttt{DCTSIZE} - \texttt{1} - \texttt{ctr}) * \texttt{DCTSIZE} + \texttt{j} \\ &\leq \texttt{data} + (\texttt{DCTSIZE} - \texttt{1}) * \texttt{DCTSIZE} + \texttt{DCTSIZE} - \texttt{1} \\ &= \texttt{data} + \texttt{DCTSIZE} * \texttt{DCTSIZE} - \texttt{1}\end{aligned}$$

(d) (*2 points*) Consider the more general setting of a pair of nested loops of the following form:

```
for (i = 0; i < imax; i++) {
   for (j = 0; j < jmax; j++) {
      ...
      ... data[i*A+j*B] ...
      ...
   }
}
```

i. (*1 points*) Assuming a simple-minded compiler with no loop-related optimizations, what arithmetic operation is used to compute the location `data[i*A+j*B]` in each execution of the inner loop ?
Answer: Two multiplications and an addition.

ii. (*1 points*) Suppose `A` and `B` are constants and `i` and `j` are only used inside array indices. What could an optimizing compiler do to simplify the arithmetic that is needed on each execution of the inner loop?
Answer: The product $\texttt{i} * \texttt{A}$ can be done outside the inner loop. In addition, the expression $\texttt{i} * \texttt{A} + \texttt{j} * \texttt{B}$ can be recognized as a linear function of the loop variable `j` and compiled using repeated addition of `j` on each loop iteration. This eliminates the multiplication $\texttt{j} * \texttt{B}$.

3. (*10 points*) .... Scope and parameter passing

(a) (*4 points*) Consider this simple program.

```
1 { int x;
2   int y;
3   int z;
4   x := 3;
5   y := 7;
6   { int f(int y) { return x*y };
7     int y;
8     y := 11;
9     { int g(int x) { return f(y) };
10       { int y;
11         y := 13;
12         z := g(2);
13       };
14     };
15   };
16 }
```

What value is assigned to `z` in line 12 under static scoping? **Answer:** 33

What value is assigned to `z` in line 12 under dynamic scoping? **Answer:** 26

(b) (*6 points*) What are the values of `y` and `z` at the end of the following block under the assumption that parameters are passed:

i. call by value **Answer:** y = 10, z = 13

ii. call by reference **Answer:** y = 15, z = 15

iii. call by value-result **Answer:** y = 13, z = 13

```
{ int y;
  int z;
  y := 7;
  { int f(int x) {
       x := x+1;
       y := x;
       x := x+1;
       return y
     };
     int g(int x) {
        y := f(x)+1;
        x := f(y)+3;
        return x
     }
     z := g(y);
   };
}
```

Answer: Here are traces for each case.

```
Value:        Reference:    Value-result (assignments in parens from copying-out):
g(7)          g(y)          g(y=7)
  f(7)          f(y)         f(x=7)
    x=8           y=8          x=8
    y=8           y=8          y=8
    x=9           y=9          x=9
    ret 8         ret 9        ret 8
  y=9           y=10         (x=9)
  f(9)          f(y)         y=9
    x=10          y=11       f(y=9)
    y=10          y=11         x=10
    x=11          y=12         y=10
    ret 10        ret 12       x=11
  x=13          y=15           ret 10
  ret 13        ret 15       (y=11)
z=13          z=15           x=13
                             ret 13
                            (y=13)
                            z=13
```

4. (*12 points*) .... Dynamic Lookup

   Answer each of the following questions about dynamic lookup in Smalltalk, C++, and Java in a few concise, carefully worded and legible sentences. Focus on the main points that distinguish these languages.

   (a) (*3 points*) Briefly describe the C++ implementation of dynamic lookup, mentioning the key data structure(s).

   Answer: Uses indirection through the vtable of the class with an offset known at compile-time

   (b) (*3 points*) Briefly describe the Smalltalk implementation dynamic lookup, mentioning the key data structure(s).

   Answer: Method dictionary with a run-time search. If a method is not found in the method diction of a class, the superclass dictionary is searched next.

   (c) (*2 points*) Which implementation would you expect to run faster? Why?

   Answer: C++; the offset of the method in the table is known at compile-time, so there's no need to search the vtable at run-time

   (d) (*2 points*) Could Smalltalk use the C++ implementation of dynamic lookup? Why or why or not?

   Answer: No, smalltalk is not a statically typed language, so different classes might have the same method in different positions in the method dictionary. Therefore, a run-time search is needed.

   (e) (*2 points*) How is Java similar or different from each of these implementations?

   Answer: Java uses the Smalltalk-style search algorithm the first time a method is called from a specific line of the source program. If the class is known at the call site, then subsequent calls use the C++-style lookup.

5. (*8 points*) .... Concurrency and Parallelism

   Graphics processors (GPUs) are one example of a parallel processor that nearly everyone has on their desktop. The frame buffer is a region of memory on the GPU that stores pixel colors. In a single rendering pass, the frame buffer can only be written to and can not be read from. Conversely, the rest of the memory on the graphics card can be read from at any time but can not be written to. After a rendering pass, the frame buffer can be copied into another part of memory where the values can be read in a subsequent rendering pass. These restrictions are enforced by the device driver for the GPU.

   (a) (*3 points*) What general problem with concurrent memory access do we solve by having a write-only frame buffer with all other memory being read-only?

   Answer: Some approach is needed to handle race conditions. These restrictions avoid race conditions without using locks. It also avoids reading pixels that have not been calculated and written yet.

   (b) (*3 points*) Java concurrency primitives allow us to write a multi-threaded software simulation of the parallel processors in a GPU. Suppose we wanted to use our GPU simulator to see what happens when we change the frame buffer from having write-only access to having both read and write access.

   Fill in the blanks in the following Java FrameBuffer class. You may need to write more than word per blank. Explain in one sentence.

```
class FrameBuffer {
  private FBdata[] pixels;
  ...

  public ____________________ read( int addr ) {
    return pixels[addr];
  }

  public ____________________ write( int addr, FBdata data ) {
    pixels[addr] = data;
  }

  ...
}
```

Answer: We'll need to synchronize both reads and writes because assignment is not atomic

(c) (*2 points*) After running several experiments with your Java GPU simulation, you decide that allowing frame buffer reads would be a very useful feature. (This is actually true for some graphics algorithms.) There is a condition, restricting which threads get to write to each pixel, that would allow writes without introducing concurrency problems. Since a hardware implementation of pixel locking is too expensive to be feasible, what condition on access to pixels or groups of pixels could a device driver enforce to prevent concurrency problems? Explain.

Answer: If a program can guarantee that the value read from the frame buffer is only being read by the same process/thread that writes the new value and nobody else (ie the computation that writes pixel[i,j] is the only one that can have read access to pixel[i,j]). This means the modification will not effect any other processing, and no locks are needed.