Spring 2017 Programming Languages Qualifying Exam

CODE_____

This is a closed book test.

Correct, clear and precise answers receive full marks

Please start a new page for each question.

There are five (5) questions

1. ***Dynamic Typing of Variables (20 points)***

Many scripting languages utilize dynamic type checking.

a) What is dynamic type checking?

b) How does dynamic type checking differ from static type checking?

c) Provide a reasonable explanation for how dynamic type checking is performed. Please be clear about how a dynamic type is stored and checked.

a) dynamic type checking is a process to improve reliability in operating a computer code. Dynamic type checking is where the type of a variable is checked during run time to ensure that the proper operation(s) are being performed on it. Many "scripting languages" utilize dynamic type checking and especially those that are object oriented. An example includes
X=2 # x is now an integer
X="TEST" # x is now a string
X = new STACK() # is now a stack

at any time, when x is referenced, the runtime system must deploy code to ensure that the operations are proper

b) static type checking is always done prior to runtime – usually during compilation of code. Static type checking also aids in reliability ensuring that operations are coded properly. Usually a static type will be assigned to a memory variable during compilation and the compiler will use rules to ensure that the operations are proper before the program is run

c) dynamic type checking means that the type of the element has to be determined and checked during runtime. The only reasonable place to store this information is in the heap where additional information is allocated during assignment. Use of the data segment is not dynamic and so a dymic type may need more storage that was statically defined. If it is allocated on the run-time stack, then the data can be dynamic, but on function return, no references should be made to the object on the stack as it is likely changed by some other activation records.

2. **Memory Mapping (20 points)**
Consider the following C code which was compiled with gcc and run on a Linux platform. Explain the output as it relates to the standard runtime memory layout. Explain what memory segments the three assignment statements are accessing.

```
char * A="T";
void foo(char *C) {
   static char D[1];
   printf("A  is at %ul\n",A);
   printf("&A is at %ul\n",&A);
   printf("C  is at %ul\n",C);
   printf("&C is at %ul\n",&C);
   printf("D  is at %ul\n",D);
   printf("&D is at %ul\n",&D);
   A[0]=1; // causes Segment Fault
   C[2]=1; // no segmentation fault
   D[2]=1; // no segmentation fault
}
main ()
{
    char B[1];
    printf("B  is at %ul\n",B);
    printf("&B is at %ul\n",&B);
    foo (B);
}
```

```
B   is at 1377329791l
&B is at 1377329791l
A   is at 4196052l
&A is at 6295616l
C    is at 1377329791l
&C  is at 1377329752l
D    is at 6295628l
&D  is at 6295628l
```

A and D are in the DATA segment. Since A is initialized to a constant, which resides in the code segment. B and C are on the runtime stack. B and D are uninitialized and gcc sets their reference back to the same cell, which allows the runtime system to catch this error. C is set to B during the call function. The A reference is attempting to update values in the code segment, which is a runtime error. C and D are referencing space on the stack and data segements, which are allocated space by the OS a page at a time.

3. **Parameters that are Subprograms (20 points)**

Assume S() is a subroutine with a non-local variable called **x** as shown below. Further assume you are using a programming language (like javascript) which allows for embedded subroutines. Give examples of how **x** is resolved in memory when S() is a parameter to a function and then called via the formal parameter

```
...
 function F(G) {
    ....    G(); // somewhere in the context of F,
                 // G is called as a subroutine/function
  }

  function S(){
    print x ;  //x is non-local
   }

... F( S );   // function S()  is a parameter of F.
```

ANSWER:
One such solution can be found in the sebasta book

```
function sub1() {
   var x;
   function sub2() {
      alert(x);  // creates a dialog box with the value of x
      };
   function sub3() {
      var x;
      x=3;
      sub4(sub2);
      };
    function sub4(subx) {
      var x;
      x=4;
      subx();
      };
 x=1;
 sub3();
 };
```

DEEP → 1
SHALLOW → 4
AD HOC → 3

4. **Method Activation (20 points)**

Consider the following C++ code. What is the output?

```
class A {
    public:  void f()  { cout << "A::f "; }
    virtual void g() { cout << "A::g "; f(); }
}; // of class A

class B : public A {
     public: void f() { cout << "B::f "; }
    void g() { cout << "B::g "; f(); }
}; // of class B

int main() {
  A a;
  B b;
  A *x = &b;
  a.g();  cout << endl;
  b.f();  cout << endl;
  x->f(); cout << endl;
  x->g(); cout << endl;
}
```

A::g A::f
B::f
A::f
B::g B::f

5. **PYTHON – Consider the following python code. (20 points)**

a) What is the output of the following python code?

```
List = [6,7,10,5,2,1,8,5]
s = reduce(lambda x,y: x if (x > y) else y, list)
print(s)
```

10

b) Write a lambda function which squares its input.

square_func = lambda x: x2**

c) Write a lambda function which takes a function (F) and an integer (m) as parameters and returns the value of F applied to m

lambda F,m: F(m)

d) What is the output of the following python code?

```
def mystery(L):
    if not L:
        return L
    if len(L) == 1:
        return L
    if reduce (lambda x,y: x or y , map(lambda x: x==L[0], L[1:]), False):
        return mystery(L[1:])
    else:
        return  [L[0]] +  mystery(L[1:])


foo = [2, 2, 2, 3 , 4 , 2]
foo.append(3)
print foo
print mystery(foo)
```

[2,2,2,3,4,2,3]
[2,3,4]