name ________________________________

CSC 533: Programming Languages
Spring 2016

Midterm

1. TRUE/FALSE	(20 pts)	
2. Discussion	(20 pts)	
3. Syntax	(16 pts)	
4. Scoping	(16 pts)	
5. Memory Management	(16 pts)	
6. OOP in Java	(16 pts)	
TOTAL	**(104 pts)**	

I pledge that I have neither given nor received unauthorized aid on this exam.

signed ________________________________

1. TRUE or FALSE? (20 pts)

_________________ ALGOL is considered by most historians to be the first *high-level programming language*.

_________________ Using a *denotational semantics* approach, the meaning of a program is described by specifying an equivalent program in a simpler, well-defined language.

_________________ An *enumeration* is an object that can be used to systematically step through the items in a data structure.

_________________ A *strongly-typed* language is one in which all variables are bound to a type at compile time and those type bindings cannot be changed.

_________________ In C, pointers can be used to access absolute addresses in memory (e.g., memory location 12345).

_________________ In both C++ and Java, the actual *sizes* of the primitive types (e.g., int, float, double) may differ from implementation to implementation.

_________________ Java does not provide `goto`'s, but the `break` and `continue` statements do provide ways of jumping out of a control statement.

_________________ *Top-down design* refers to an approach to software development where key decisions are made by top-level managers and then filtered down to low-level programmers.

_________________ The *IEEE floating-point format* provides a standard structure for representing real values in a programming language.

_________________ For a grammar to be deemed *ambiguous*, every expression in that language must be derivable by multiple parse trees.

2. Discussion (20 pts)

A. In class, we discussed several characteristics of a programming language that contribute to its *readability* (i.e., how easy it is to understand and maintain code in that language). Describe two (2) characteristics that contribute to readability. How does Java rate on these characteristics? Justify your answers.

B. *By-value-result* and *by-reference* are alternative methods for providing in-out mode parameter passing. While by-value-result is intuitive, ambiguities occur in special cases that make its behavior unclear. Describe one such case and how by-value-result parameter passing results in an ambiguity. How is this case handled with by-reference?

C. In FORTRAN, the user does not need to declare variables explicitly – an undeclared variable starting with I, J, K, L, M, or N is assumed to be an INTEGER, while any other undeclared variables is assumed to be a REAL. *Describe one possible advantage* of such implicit declarations. *Describe one possible disadvantage.*

D. Consider a common mistake made by many beginning programmers – reversing the sides in an assignment. For example,

```
    5 = x;
```

In C, this would cause a run-time error with a message something like:

```
  ERROR: l-value expected.
```

What does the term *l-value* refer to with respect to a variable and its bindings? Where is the compiler expecting to see an l-value and failing to do so?

E. One of the striking features of the Mark & Sweep approach to garbage collection is that its cost is inversely proportional to the amount of storage recovered; i.e., the less recovered, the more it costs to perform the garbage collection. *Describe the steps* involved in Mark & Sweep garbage collection. *Which step(s) require more work* if the amount of garbage (i.e., recoverable) memory is small?

3. Syntax (16 pts)

Consider the following BNF grammar rules that define expressions involving addition and prefix/postfix increment.

```
<expr>   →  <pre> '+' <expr> | <pre>
<pre>    →  '++' <post> | <post>
<post>   →  <term> '++' | <term>
<term>   → '(' <expr> ')' | <id>
<id>     →  'A' | 'B' | 'C' | 'D'
```

A. What is the associativity of the addition operator? In particular, how would the expression `A+B+C` be parsed? Justify your answer.

B. Is `A++B` a valid expression (`<expr>`) in this language? If so provide a parse tree and note whether that parse tree is unique (with justification). If not, explain why it is not a valid expression.

C. Is `A+++B` a valid expression (`<expr>`) in this language? If so provide a parse tree and note whether that parse tree is unique (with justification). If not, explain why it is not a valid expression.

D. Suppose we wished to add a factorial operator that could be applied to a single value. The postfix, unary `!` operator should bind more tightly than the other operators, so that `A!+B` would be parsed as `(A!)+B`. Modify the grammar rules (which are reproduced below) to include this new `!` operator. In addition, use your modified grammar rules to provide a parse tree for `A!+B`.

```
<expr>   →  <pre> '+' <expr> | <pre>
<pre>    →  '++' <post> | <post>
<post>   →  <term> '++' | <term>
<term>   → '(' <expr> ')' | <id>
<id>     →  'A' | 'B' | 'C' | 'D'
```

4. Run-time Stack (16 pts)

Consider the following ALGOL-like program:

```
program MAIN;
  int J, K;

  procedure OUTPUT();
  begin
    print "OUTPUT: ", J, K
  end;

  procedure ONE(int K);
  begin
    OUTPUT();
  end;

  procedure TWO();
  int J;
  begin
    J := 5;
    ONE(K-3);
  end;

begin
  J := 9;
  K := 23;
  ONE(14);
  TWO();
end.
```

A. In this program, the subroutine OUTPUT is called twice. Draw the contents of the run-time stack *at each point in the execution after OUTPUT is called.* For each activation record on the stack, be sure to indicate both the static and dynamic links for that record, as well as the values of any parameters and local variables. Assume that parameters are passed by-value.

1st call | *2nd call*

A. What is output by this program if *static scoping* is used?

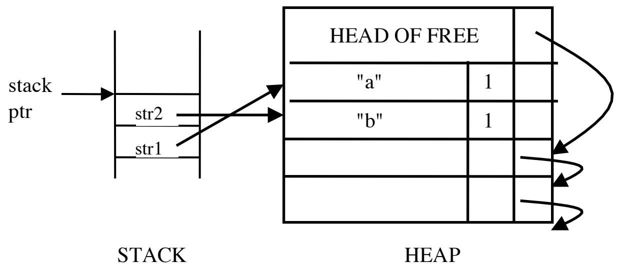
B. What is output by this program if *dynamic scoping* is used?

5. Memory Management (16 pts)

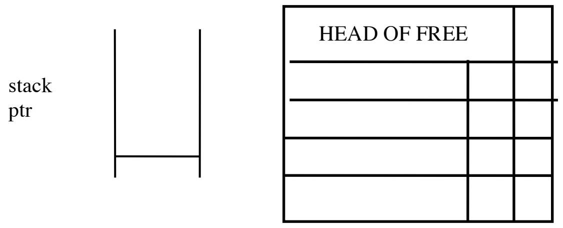
Trace the contents of the stack and heap using *reference counts* to avoid garbage references. In particular, at each checkpoint in the code execution show stack-dynamic variables in the stack and pointers into the heap (with corresponding values in the cells), the reference count for cells in the heap, and the links connecting the free list. The contents of the stack and heap at CHECKPOINT 0 are shown :

```
String str1 = "a";
String str2 = "b";
/* CHECKPOINT 0 */
if (!str1.equals(str2)) {
    String str3 = "c";
    str1 = str3;
    /* CHECKPOINT 1 */
    str3 = str2;
    str2 = str1;
    /* CHECKPOINT 2 */
}
/* CHECKPOINT 3 */
String str4 = "d";
/* CHECKPOINT 4 */
```

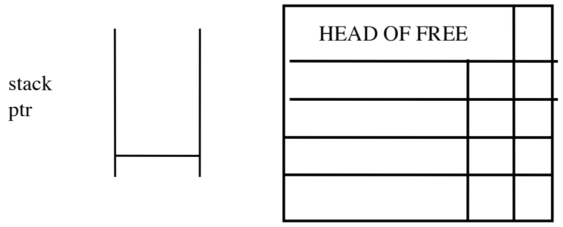
at CHECKPOINT 0:



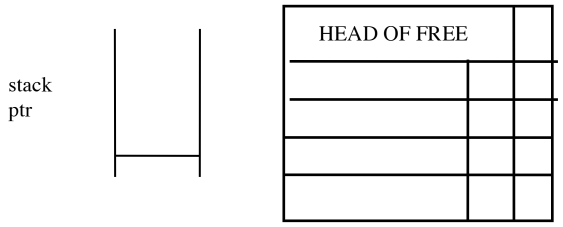
at CHECKPOINT 1:



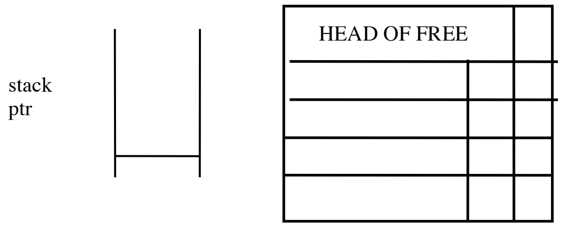
at CHECKPOINT 2:



at CHECKPOINT 3:



at CHECKPOINT 4:



6. OOP in Java (16 pts)

Consider the following class from the SILLY interpreter (HW2):

```
public abstract class Statement {
    public abstract void execute() throws Exception;
    public abstract String toString();
    public static Statement getStatement() throws Exception {
       // implementation not shown
    }
}
```

A. What is the purpose of the keyword *abstract* that appears in the header of the `Statement` class? Why do some components of the class have the *abstract* keyword (e.g., `execute`) and some not (e.g., `getStatement`)?

B. What is the purpose of the keyword *static* that appears in the header of the `getStatement` method? How would removing this keyword affect the behavior of the program?

Consider the following new type of statement that could be added to the SILLY language.

```
public class Mystery extends Statement {
    private Token vbl;

    public Mystery(TokenStream input) throws Exception {
        Token op = input.next();
        this.vbl = input.next();
        if (!op.toString().equals("incr") || this.vbl.getType() != Token.Type.IDENTIFIER ) {
                throw new Exception("SYNTAX ERROR: Malformed mystery statement");
        }
    }

    public void execute() throws Exception {
        DataValue val = Interpreter.MEMORY.lookupVariable(this.vbl);
        if (val.getType() != Token.Type.INTEGER) {
            throw new Exception("RUNTIME ERROR: Illegal value for mystery");
        }
        IntegerValue newVal = new IntegerValue((Integer)(val.getValue())+1);
        Interpreter.MEMORY.storeVariable(this.vbl, newVal);
    }

    public String toString() {
        // TO BE COMLETED IN PART F
    }
}
```

C. Describe the general form that a `Mystery` statement can take in the SILLY language. Give one specific example of a valid `Mystery` statement.

D. Is it possible for a `Mystery` statement to be syntactically valid, and yet still produce a run-time error when executed? If so, give an example and explain why the run-time error would occur. If not, explain why not.

E. Describe in general what a `Mystery` statement does when executed.

F. Complete the definition of the `toString` method, which is intended to return a `String` representation of a `Mystery` statement (i.e., all of the components of the statement in a single `String`).

```
public String toString() {
```