COS 441 Take-Home Final
Fall 1998

Kim Bruce, instructor

Due noon, 1/15/99

Instructions: This is an open book exam. You may consult your notes, all homework you turned in, the text, any papers handed out in class, the on-line lecture notes, links to documents from the course web pages, and the instructor. You are allowed to access the links to documentation on ML, Java, and GJ on the Schedules and Readings web page and the Handouts web page available through the course page. You may use ML, Java, and GJ compilers while working on this exam. You may not talk with anyone else about the exam, or consult any other materials (including other material in the library or your classmates' notes - unless you have copied them before the exam).

Please give complete (but not redundant) answers. Good Luck and don't hesitate to ask if you have any questions. I will (tentatively) have office hours Monday-Thursday, 1/11-14 from 2 to 3 p.m. each afternoon, and Friday between 10 and 11 a.m. I will also read e-mail regularly (i.e., at least daily) during the exam. I will send e-mail to you with any changes in office hours or corrections on the exam. Be sure to read e-mail before turning in your exam to make sure that you have any corrections.

Please turn in paper copies of all of your answers (including printouts of all programs written as part of your solutions) to me or to Tina McCoy in room 410. *Also* turn in electronic copies of all programs. Please write "I pledge that I have not violated the honor code on this examination" with your signature on your solutions.

1. Consider the following Pascal-like program segment where `sum`, `count`, and `n` are integer variables:

```
sum := 0;
count := 0;
while count < n do
   sum := 2*sum + 1;
   count := count+1
end
```

   Using the formal definition of axiomatic semantics given in lecture, prove the program correct with respect to the following pre and post-conditions:

   Pre: $n > 0$
   Post: $sum = 2^n - 1$

Be sure to give a formal proof which depends only on the definitions of axiomatic semantics given in class (i.e., an informal proof by induction will net you very few points). Cite all axioms and rules used at each step.

Hint: Finding a loop invariant relating `sum` and `count` will help you get started!

2. In class we provided encodings for objects, classes, and subclasses, and have shown how to send messages, access instance variables (inside classes), and generate new objects. However we haven't yet shown how to send messages to superclasses. That is, if class `D` extends class `C`, and both have method `m`, then the definition of `m` in `D` often begins with `super.m(..)` followed by some code to take care of the extra instance variables in the subclass. Here is a simple example using `CellClass` from the lecture notes.

```
class CellClass{
   x: int = 0;
   get(): int { return self.x }
   set(nuVal:int): void { self.x := nuVal }
   bump(): void { self.set(self.get()+1) }
 }

class ColorCellClass inherit CellClass{
   color: ColorType;
   getColor(): ColorType {return self.color}
   set(nuVal:int): void {super.set(nuVal);
                         self.color := red;}
}
```

   (a) Please explain how to take encode "super" calls in a way compatible with the encodings given in class.

   (b) Show this translation in detail for the method `set` of `ColorCellClass`.

   Hint: Suppose the superclass is of the form `c = class(inst,meth)` with type `ClassType(IV,M)`, and the subclass is of the form `class inherit c (sinst,smeth)` of type `ClassType(sIV,sM)`. The key to handling `super.m` is to consider the method suite from the superclass and the way in which we built *meth'* from *meth*, but this time build it expecting the instance variables from the subclass.

3. In problem 3 of assignment 2, I asked you to write an ML program to evaluate arithmetic expressions formed with the following grammar.

```
e ::= n | ~e | e + e | e - e | e * e | e / e | (e)
```

   where `n` stands for an integer, `~e` is the negative of `e`, the next four terms are the sum, difference, product, and quotient of expressions, while `(e)` is used to determine the order of evaluation of expressions (e.g. `3-(5+6)`). You might want to go look again at that assignment to see what was expected of you there.

For this problem I would like you to write a program in GJ (or Java) to evaluate parse trees for arithmetic expressions from this language. As usual, I would also like the expressions to have a method `toString` which returns a String representing the (unevaluated) fully parenthesized form of the expression.

In ML, you created a datatype definition which included all of the parts of the grammar. In object-oriented languages, you should create an interface `Expression` which publicizes the methods available to be used on all expressions. Each of the different kinds of terms will be represented by a class which implements `Expression`. (No class is necessary for the last type of term – parenthesized expressions – since we are only dealing with fully parsed expressions.)

A value representing expression "3-(5+6)" can be created via something like `new Minus(new Num(3),new Plus(new Num(5),new Num(6)))`. You *do not* have to write a program which parses arithmetic expressions. Just like the earlier ML program, you need only manipulate the values built from the constructors.

Rather than just implementing each of the classes for the specialized terms, I would also like you to write an abstract class `BinaryOp` which will serve as a superclass for classes representing addition, subtraction, multiplication and division. It should include the implementation of a `toString` method which should not be overridden in any of the subclasses. To make this possible I suggest you include a String-valued instance variable in `BinaryOp` which can be set to a representation of the operator in the constructors of the subclasses. `BinaryOp` should also define instance variables `left` and `right` representing the arguments of the operator. The method `evaluate` should be implemented in its subclasses.

Division should be implemented as integer division (i.e., the same result as using "/" on two integers in Java). A complication that you must deal with is the possibility of dividing by zero. If evaluating an expression results in a divide by zero, it should raise the exception `DivideByZero`.

Test your program by writing a class `RunIt` whose main program will build several expressions, print them out, and then evaluate them. Please include somewhere in your program the evaluation of the expression `(3/(4/5))+1` followed later by the evaluation of expression `6/(3-1)`. If evaluating any expression results in throwing the `DivideByZero` exception then the program should catch the exception and print out a message explaining which subexpression caused the exception to be thrown (e.g., in the example above, evaluating `3/(4/5)` caused the exception to be thrown). Throwing the exception should have no impact on the evaluation of other expressions in the main program.

To receive full credit for this problem you should build an interface and class structure exactly as described here. While other solutions are possible (and arguably as good), this solution includes many of the features of object-oriented languages that I'd like to be sure you understand.

4. Functional and object-oriented programs have different strengths and weaknesses with respect to extensibility. In the previous problem you wrote an interpreter for simple

arithmetic expressions. As pointed out there, you had earlier solved the same problem for ML.

(a) Suppose you wish to add a new arithmetic expression representing raising an integer expression to an integer power. Would it be easier to add this expression in the functional solution or the object-oriented solution. Why? As part of your answer, sketch what must be done in each situation. I'm not interested in details, more in what parts of code must be changed or written from scratch. (Consider how much code must be rewritten and recompiled as opposed to just adding new code which does not require changes to old code.)

(b) Suppose you wish to add a new operation to be applied to all arithmetic expressions. For example, suppose rather than printing fully-parenthesized expressions, we wish to print them in a two-dimensional form as parse trees. Would it be easier to add this expression in the functional solution or the object-oriented solution. Why? As part of your answer, sketch what must be done in each situation.

(c) The Visitor pattern for object-oriented languages is designed to make it easy to add new operations with object-oriented languages. For example, we can define the interface for an expression visitor (or processor) as follows:

```
/* Interface for arithmetic expression processors */

public interface ExpProcessor<Result>{
  Result forTimes(Times te);
  Result forPlus(Plus pe);
  Result forDivide(Divide de);
  Result forMinus(Minus me);
  Result forNum(Num ne);
  Result forNeg(Neg ne);
}
```

Now add a parametric method with the following signature to `Expression`

```
<Result> Result process( ExpProcessor<Result> ep);
```

Each class implementing `Expression` should implement `process` by having it call the corresponding "for" method of `ExpProcessor` with `this` as its argument. For example, the `Times` class should contain

```
public <Result> Result process(ExpProcessor<Result> ep){
  return ep.forTimes(this);
}
```

It is now possible to define new "visitors" that operate on expressions without going back and adding new methods to the original classes. To define a new visitor, simply define a class implementing `ExpProcessor` so that the "for" methods perform the appropriate operations.

Rewrite your solution to the previous problem to use visitors rather than to build in `evaluate` and `toString` methods in the classes implementing `Expression`. In this new version, only the `process` method should be included in interface `Expression`. As before, build a class implementing `Expression` for each type of expression in the grammar. Again, all must contain `process` methods, but must not contain `evaluate` or `toString`.

Create two new classes `EvalProcessor` and `PrintProcessor`. Evaluating a term of the form `exp.process(new EvalProcessor())` should return the value of the expression `exp`, while `exp.process(new PrintProcessor())` should return a string that represents the fully parenthesized (unevaluated) form of the expression. The class `PrintProcessor` should implement `ExpProcessor<String>`, while `EvalProcessor` implements `ExpProcessor<Integer>` (unfortunately, polymorphic classes cannot be instantiated with `int` in GJ). Test this new program as you did in the previous problem.

Warning: Don't forget that you must again worry about division by zero in `EvalProcessor`. Set things up so that the result of dividing by zero is handled as in the previous question. You may also find that you must make some things public in the expression classes that would normally be kept protected or private. (Your solution should work whether or not the visitor is a member of the same package as the classes implementing Expression.)

(d) Compare briefly the ease of adding new operations using the visitor pattern in object-oriented languages as opposed to the mechanisms available in ML. Carefully enumerate advantages and disadvantages. of each.

5. Please do problem 42 on page 545 of the text by Louden.

6. Many people feel that object-oriented languages do not need modules because classes provide the same facilities.

   (a) Please compare the facilities available in ML structures, signatures, and functors, with Java and GJ's (parameterized) classes and interfaces. Be sure to highlight the differences.

   (b) Object-oriented languages like GJ seem to require F-bounded polymorphism (or something equivalent) for expressiveness. How is the equivalent of bounded polymorphism supported in ML functors? How about in Ada packages? Why don't these languages need F-bounded polymorphism?