CS 164, Fall 1999
Midterm #2
Professor Aiken

Problem #1: Parametric Polymorphism

Consider the functional language grammar used in class:

```
   Program -> Definition; Definitions | ∈
Definition -> Id(Id) = Expr
      Expr -> Id | (Expr Expr) | (fun Id expr) | i | (Expr
              + Expr) | [Expr, Expr] | (first Expr) |
              (second Expr)
```

In this language, the construct $(\texttt{Expr}_1\ \texttt{Expr}_2)$ is a function application where $\texttt{Expr}_1$ is the function and $\texttt{Expr}_2$ is the argument. The expression `(fun Id Expr)` is an anonymous function with argument `Id` and body `Expr` (the equivalent of *lambda* in Scheme). A `i` is an integer constant, `[Expr, Expr]` creates a pair of the values of the two expressions, and `first` and `second` are selectors for the first and second for the first and second components of pairs, respectively.

For each of the following types, give an expression (from the language above) for which this is the principal type.

a. `int -> int`
b. `((α × β) × γ) -> (γ × (β × α))`
c. `α -> ((α -> β) -> β)`

Problem #2: Code Generation

Consider a `for` construct

$$\texttt{for i = } e_1 \texttt{ to } e_2 \texttt{ by } e_3 \texttt{ do } e_4$$

The subexpressions are integer-valued. A `for` is evaluated according to the following rules. The first three subexpressions are evaluated once at the start of the loop in the order e_1, e_2, then e_3. The expression e_4 is evaluated once per iteration of the loop.

The index variable `i` is initialized to the value of e_1. The loop bound is the value of e_2 and `i` is incremented by the value of e_3 after each iteration. The loop terminates before executing an iteration where the value of `i` is greater than the loop bound. The value returned by a `for` loop is 0.

Show how to generate stack machine code for this construct. You may use any reasonable and clear pseudo-code assembly language. You may assume:

- Integer values are represented by a single machine word.
- Expressions put their results in register $a0.
- The index variable `i` is kept at the address 0($fp).
- The stack pointer is in register $sp.
- Expressions have no net effect on the stack (e.g., expressions pop any temporary values).

Problem #3: Attribute Grammars

Consider the following language:

```
 Def -> def Id(Id, ..., Id) = Expr
Expr -> for Id = int, int do Expr
      | Id = Expr
      | Expr; Expr
      | Expr * Expr
      | Expr + Expr
      | Id
      | int
```

The *cost* of an expression is the cost of evaluating subexpressions plus the cost of the expression itself. For example, the cost of a + is 1 plus the cost of the expressions; for clarity we will simply say that the additional cost of a + is 1. The additional a * is 2. The additional cost of a statement sequence `Expr; Expr` is 0. The additional cost of an assignment is 1. The additional cost of a `for` loop is 3, plus an additional 2 plus the cost of the subexpression for each iteration of the loop. The additional cost of a function definition is 2. The cost of variable references `Id` and constants `int` is 1.

a. Give an attribute grammar to compute the cost of function definitions. You may assume there is an attribute `val` that contains the lexeme of a token. You may define any other attributes you wish.
b. Now assume that parameter passing is by name. Can you modify your attribute grammar to compute cost in this case? If so, how? If not, why not?
c. Are the attributes you defined (those other than `val`) synthesized or inherited?

Problem #4: Overloading

Java allows methods to be overloaded. The Java overloading resolution rule is: When given a choice of multiple methods in a class with the same name, the method with the most specific argument types is selected. (It is a compile time error if there is no unique method with the most specific argument types.) For example, consider an expression `a.f(b)` where *a* has type `Foo` and assume class `Foo` has two `f` methods `f(Object x)` and `f(Bar x)`. Then if `b` has type `Bazz`, where `Bazz` is a subtype of `Bar`, then the overloading is resolved to the `f(Bar x)` method.

Java resolves overloading at compile time using the static types, while dynamic dispatch selects methods to execute at run time using the dynamic types. This is all as usual, but overloading has tricky interactions with dynamic dispatch. Consider the following program skeleton:

```
// All access restrictions (public/protected/private) omitted for clarity

class Point{
    boolean equal(Point x){...}   //*1*
}

class ColorPoint extends Point{
    boolean equal(ColorPoint x){...}   //*2*
}

Point p1 = new Point();
Point p2 = new ColorPoint();
ColorPoint cp = new ColorPoint();
```

The overloading in this example is in the subclass `ColorPoint`, which has two `equal` methods (the one in the class and the inherited one from `Point`).

For each method call, say which version of `equal` is called and why.

a. `p1.equal(p1);`
b. `p1.equal(p2);`
c. `p2.equal(p1);`
d. `p2.equal(p2);`
e. `cp.equal(p1);`

f. `cp.equal(p2);`
g. `p1.equal(cp);`
h. `p2.equal(cp);`
i. `cp.equal(cp);`
