Programming Languages

Qualifying Exam, January 1996

The exam has four questions, of equal weight; please answer all four.
Read the whole exam before beginning to answer the questions.

Question 1. This question concerns compiling Boolean expressions into machine code. Short circuit evaluation is required, i.e. evaluation is left to right but stops as soon as the value of the expression is known. The logical operators are $\neg$ (not), $\wedge$ (conditional and), and $\vee$ (conditional or). There are Boolean variables, but no relations. The source language is defined by the following ambiguous grammar:

$$E ::= id \mid \neg E \mid E \wedge E \mid E \vee E \mid (E)$$

where the precedence of operators, from highest to lowest, is: $\neg$, $\wedge$, $\vee$. The object language contains sequences of conditional branch instructions of the following form:

```
branchTrue id, label ;    (jump to label if variable id is true)
branchFalse id, label ;    (jump to label if variable id is false)
```

Each instruction may be preceded by 0 or more labels of the form "*label* :". Here's an example.

```
lab1 :  lab2 :  branchTrue x, lab3 ;
```

You may assume the existence of a function *newlab* that generates a new unique label. Compilation of a Boolean expression is parameterized with respect to two parameters: μ and τ. If μ is true (false), the generated code is to branch to label τ if the Boolean expression is true (false) and fall through to the next sequential instruction otherwise.

Question 1a. Define an LL(1) grammar for the source language and argue that it is LL(1). Do not use an extended grammar notation, i.e. do not use braces, brackets, stars, etc.

Question 1b. Define an attribute grammar for an abstract syntax tree (AST) intermediate representation for Boolean expressions. Compute the object code for a Boolean expression as attribute c of the root of the corresponding AST. Use compilation parameters μ and τ to initialize appropriate attributes. Attribute definitions can be written in arbitrary pseudo-code; just be clear and unambiguous.

Question 2. Programming methodology.

Question 2a. Explain the method of formal development of programs, in which a program and its proof are developed hand in hand. Concentrate on the development of loops (with one guarded command). Be complete and detailed but brief. If you wish, this explanation may be interspersed with the answer to the next part.

Question 2b. Use the method of part (a) to write an algorithm for the following problem. Show all your work, so that we see that the method is being followed, but you need not belabor the obvious. You need not perform formal calculations when the results can be ascertained without them.

Given is an integer r, $r > 0$. Desired is a sequence s of pairs of integers (x, y) that satisfies postcondition $R1 \wedge R2$, where:

$R1$: s contains exactly the pairs (x, y) such that $x^2 + y^2 = r \wedge 0 \leq x \leq y$
$R2$: the x-components of pairs of s are in ascending order

For example, for $r = 25$, the answer is $(0,5),(3,4)$.

The algorithm should not use *sqrt* or similar operations —only addition, subtraction, and multiplication. Essentially, it has to search the space of possible pairs (x,y) in an efficient manner. Use $s := ()$ to set s to the empty sequence and $s := s \,\hat{}\, (x,y)$ to append a pair (x,y) to s.

Start your development by introducing a variable X and use the following first approximation $P1 \wedge R2$ to the invariant, where:

$$P1: \; s \text{ contains exactly the pairs } (x,y) \text{ such that } x^2+y^2=r \wedge 0 \le x \le y \wedge x < X$$

The resulting program is likely to be quadratic in r; what do you do to improve efficieny?

Question 3. You are working on a new object-oriented language, LOSS, with single inheritance and strong (static) typing —runtime errors are impossible in LOSS.

As an example of LOSS syntax, here is the declaration of class *Cat*. *Cat* is a subclass of *Animal* (not defined here). *Cat* has an integer member *lives*, which is the number of lives a cat still has. Also, *Cat* has method *play*, which takes a *Toy* as an argument and returns a *Cuteness*.

```
object Cat is Animal;
  lives: Integer
  method play(cattoy:Toy) : Cuteness
end object
```

A LOSS program has an initialization section (its details are not important), in which the methods of a class are bound to code. For example, below, *Cat*'s method *play* is bound and *Cat*'s method *eat* (which is defined in *Animal*) is overridden. As can be seen from the example, a member m or a method m of class A is referenced using the syntax *A.m*; the first argument is the *self* parameter, and a call of a method automatically passes in the correct object for this parameter. Every object has a *parent* member, which has the value of the object's parent.

```
function catplay(self:Cat, cattoy:Toy) : Cuteness
  print "The cat " self.name " plays with " cattoy.name
  return self.actCute()
end catplay

function cateat(self:Cat, food:Food) : Noise
  print self.name " eats daintily"
  return self.parent.eat(food)
end cateat

Cat.play := catplay
Cat.eat := cateat
```

For a class A with method *foo* of n arguments, the statement *A.foo* := *bar* binds *bar* to A's method *foo*. Here *bar* must be a function with $n+1$ arguments — as stated above, the first is used as the *self* parameter. Each method can be bound only once, and only in the initialization section.

Here's some notation: For types (i.e. classes) A and B, $A \le B$ stands for "A is a subclass of B", and $A = B$ stands for "A is the same type as B".

Question 3a. What are the necessary and sufficient conditions for the binding *A.foo* := g to be type-correct? More precisely, if *A.foo*'s signature is $T_1 \times \cdots \times T_n \rightarrow T_r$ and g's signature is $T'_0 \times T'_1 \times \cdots \times T'_n \rightarrow T'_r$, what conditions on the types T_i and T'_i are necessary and sufficient?

You answer should be a set of equations using $\leq$ and $=$. Give a short, informal argument why your conditions are necessary and sufficient.

Question 3b. A new feature is to be added to LOSS, to yield a language DEADLOSS. DEADLOSS allows the binding of code to methods to occur at arbitrary points in the program. For example, a function could change the binding of method *Animal.eat* several times, with the result that the behavior of all animals in existence would change each time.

Your coworker Ranma argues that this feature doesn't add functionality. Since functions are first-class entities in LOSS, one could simply replace a method in a class with a member of the appropriate function type and then do assignments as normal. For example, method *Animal.eat* would be replaced by a function $eat : Animal \times Food \rightarrow Noise$ and could simply assign to this member instead of bothering with the binding.

Is Ranma correct? Explain.

Question 3c. Assuming the compiler writers for LOSS and DEADLOSS are reasonably competent, what impact would the bind-anywhere feature have upon the efficiency of DEADLOSS code compared to LOSS code? Explain.

Question 4. The programming language REG has the following syntactic categories:

$\mathbb{N}$-valued variables: x
Terms: $t ::= 0 \mid x \mid t+1$
Programs: $p ::= (x := t) \mid (t_1 = t_2)? \mid (t_1 \neq t_2)? \mid p_1 + p_2 \mid p_1; p_2 \mid p^*$

A state is a function from the variables to the natural numbers $\mathbb{N}$. Each term t denotes a function $[\![t]\!]$ from the states to $\mathbb{N}$:

$$\begin{aligned} [\![0]\!] &= \lambda\sigma.\,0 \\ [\![x]\!] &= \lambda\sigma.\,\sigma(x) \\ [\![t+1]\!] &= \lambda\sigma.\,[\![t]\!](\sigma)+1 \end{aligned}$$

Each program p denotes a binary relation $[\![p]\!]$ on the states:

$$\begin{aligned} [\![x := t]\!] &= \{(\sigma,\sigma') \mid \sigma' = \sigma[x := [\![t]\!](\sigma)]\} \\ [\![(t_1 = t_2)?]\!] &= \{(\sigma,\sigma) \mid [\![t_1]\!](\sigma) = [\![t_2]\!](\sigma)\} \\ [\![(t_1 \neq t_2)?]\!] &= \{(\sigma,\sigma) \mid [\![t_1]\!](\sigma) \neq [\![t_2]\!](\sigma)\} \\ [\![p_1 + p_2]\!] &= [\![p_1]\!] \cup [\![p_2]\!] \\ [\![p_1; p_2]\!] &= [\![p_1]\!] \circ [\![p_2]\!] \\ [\![p^*]\!] &= [\![p]\!]^* \end{aligned}$$

where state $\sigma[x := n]$ maps x to n and all other variables y to $\sigma(y)$; $\cup$ is set union; $\circ$ is relation composition; and ρ^* is the reflexive, transitive closure of relation ρ.

For a pair (σ,σ') of states, $(\sigma,\sigma') \in [\![p]\!]$ iff program p begun in state σ may terminate in state σ'. For example, consider the program

$$q = ((x = 0)?; (x := 1)) + ((x = 1)?; (x := x+2)^*).$$

When started in a state σ with $\sigma(x) = 0$, q terminates in the state $\sigma[x := 1]$; when started in a state σ with $\sigma(x) = 1$, q terminates, nondeterministically, in some state of the form $\sigma[x := n]$, where n is an odd number n; when started in any other state, q does not terminate.

Question 4a. For the REG program *If*:

$$If = ((t_1 = t_2)?; p_1) + ((t_1 \neq t_2)?; p_2)$$

define $[\![If]\!]$ in terms of $[\![t_1]\!]$, $[\![t_2]\!]$, $[\![p_1]\!]$, and $[\![p_2]\!]$.

Question 4b. Using t_1, t_2, and p, write a REG program *While* so that $[\![While]\!]$ is the least fixpoint of the following monotonic function F from sets of pairs of states to sets of pairs of states:

$$\begin{aligned} F \;=\; \lambda\rho.\, & \{(\sigma,\sigma) \mid [\![t_1]\!](\sigma) \neq [\![t_2]\!](\sigma)\} \cup \\ & \{(\sigma,\sigma') \mid [\![t_1]\!](\sigma) = [\![t_2]\!](\sigma) \wedge (\exists\sigma''.\,(\sigma,\sigma'') \in [\![p]\!] \wedge (\sigma'',\sigma') \in \rho)\} \end{aligned}$$

Very briefly, sketch a proof that $[\![While]\!]$ is the least fixpoint of F (our proof sketch is two sentences).

Question 4c. Give a natural semantics for REG. That is, give derivation rules with conclusions of the form $\langle t, \sigma\rangle \rightarrow n$ and $\langle p, \sigma\rangle \rightarrow \sigma'$ so that $\langle t, \sigma\rangle \rightarrow n$ can be derived iff $[\![t]\!](\sigma) = n$ and $\langle p, \sigma\rangle \rightarrow \sigma'$ can be derived iff $(\sigma, \sigma') \in [\![p]\!]$.