CS 536
Final Exam

Thursday, May 11, 2006

2:45 — 4:45 PM

1221 CSST

Instructions

Answer any *five* questions. (If you answer more, only the first five will count.) Each question is worth 20 points. Please try to make your answers neat and coherent. Remember, if we can't read it, it's wrong. Partial credit will be given, so try to put something down for each question (a blank answer always gets 0 points!).

1. Assume we have a CSX function whose header is

   ```
   int t(int a, int b, int c) {…
   ```

   and a call

   ```
   z = t(0, 1, t(2, t(3,4,5), 6));
   ```

   Show the JVM code you would generate for this call.

   Explain the steps the JVM interpreter performs to actually do the indicated function calls (parameter passing, frame manipulation, return address manipulation, etc.)

2. Assume that we add a "repeat until" loop to CSX:

   ```
   repeat
     stmts
   until (condition);
   ```

   `condition` is a boolean-valued expression that is evaluated at the end of each iteration. Iteration terminates as soon as `condition` becomes true. At least one iteration of the loop always occurs.

   Show the JVM code you'd generate for this kind of loop. Design an AST structure and code generator appropriate for this loop.

3. Assume we modify the structure of a CSX class to separate declarations and implementations. A class begins with class **declarations**. These are variable and constant declarations (exactly the same as in original CSX) as well as method headers (**without** method bodies).

An “**implemented as**” section follows that contains the bodies of each method declared in the class. Each method declared in the class must have a body defined in this section, and no body may be defined unless it has been previously declared. Here is a simple example of this revised class structure:

```
class demo {
      char skip = '\n';
      int f();
      void main();
implemented as
      f:    {return 10;}
      main: { print("Ans =",f(), skip); }
}
```

What are the advantages of structuring a class in this manner (from the programmer’s point of view)?

What changes are needed in the structure of an AST `classNode`? How must CSX type checking and code generation be modified to compile this new class structure?

4. Consider the following three context-free grammars. Which are LL(1)? Why? Which are LALR(1)? Why?

(i)
$S \longrightarrow x\ T\ y$
$T \longrightarrow A\ T$
$T \longrightarrow b$
$A \longrightarrow a$
$A \longrightarrow \lambda$

(ii)
$S \longrightarrow \{\ S\ \}$
$S \longrightarrow \{\ S\ ;\ \}$
$S \longrightarrow$ return

(iii)
S $\longrightarrow$ Mode Access main
Mode $\longrightarrow$ static
Mode $\longrightarrow \lambda$
Access $\longrightarrow$ private
Access $\longrightarrow \lambda$

5. Most modern programming languages, including C, C++, Java and CSX, allow recursive methods. However, most methods that we use *aren’t* recursive. Assume we know that some method P isn’t recursive. How can we simplify the implementation of a call of P?

Assume that we have a CSX program in AST form. How can we determine, by examining this AST, whether any of its methods are recursive? (You may assume that any call within a method is reachable. That is, if a call of Q appears *anywhere* within P, then any method that can call P can also indirectly call Q.)

6. Assume we know that some context-free grammar, G, is LL(1). Moreover, we also know that no non-terminal in G generates *only* λ. If we remove all productions of the form $A \longrightarrow \lambda$ (i.e., all productions that directly derive λ), is the resulting grammar still LL(1)?

   If it is, carefully explain why. If it isn't, present a simple counter-example that shows that LL(1)-ness isn't always preserved.

7. The JVM code we generate for variables and expressions normally pushes (or computes) the value of a variable or expression onto the stack. Call this *stack code*.

   An alternative is *jump code*. In jump code, evaluation of a boolean-valued variable or expression results in a "jump" to either a *true address* (if the value is true) or a *false address* (if the value is false). Jump code is particularly useful for loops and conditionals since we want boolean values to control execution, jumping to one location or another depending on the expression value.

   Assume we have a boolean value (a zero or a one) at the top of the stack. What JVM code should we generate to transform this value into a conditional jump to labels `L_true` and `L_false`?

   Assume we have translated a boolean expression, `e`, into jump code that jumps to `L_true` and `L_false`. How can we transform this generated code into jump code that computes `!e` (where `!` is the boolean not operator)?