CS 536

Final Exam

Thursday, December 18, 1997

7:45 AM— 9:45 AM

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 a reduced grammar, G. Define CanGen(A), where A is a nonterminal, to be the set of terminals, b, such that $A \Rightarrow^{+} \ldots b \ldots$. That is, CanGen(A) contains all the terminal symbols that may be derived starting with A.

   Give an algorithm that computes CanGen(A) given grammar G.

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

   (i) $S \rightarrow A\ S$
   $S \rightarrow b$
   $A \rightarrow a$
   $A \rightarrow \lambda$

   (ii) $S \rightarrow (\ S\ )$
   $S \rightarrow [\ S\ ]$
   $S \rightarrow [\ S\ )$
   $S \rightarrow a$

   (iii) $S \rightarrow D\ P\ a$
   $D \rightarrow b$
   $D \rightarrow \lambda$
   $P \rightarrow c$
   $P \rightarrow \lambda$

3. Recall that in generating JVM code for methods we “guestimated” that a maximum stack depth of 25 would be sufficient. Give an example of a CSX program that requires a stack depth greater than 25.

   Rather than guessing at the stack depth we need, we should compute the needed stack depth directly. Outline how you’d compute this value as you are generating code for a method body.

4. In type checking array parameters we could not always determine at compile-time whether an assignment of one array parameter to another was valid. Explain why this was the case.

   Outline how we could determine the possible sizes an array parameter might have during execution. How could this information help us in type checking assignments of array parameters?

5. Assume we have a CSX function whose header is

   ```
   int p(int a, int b) {...
   ```

   and the call

   ```
   z = p(1, p(2, p(4,5)));
   ```

   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.)

6. 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)?

7. Assume we extend the CSX + operator to allow string literals and character arrays as operands. The + operator concatenates the operands, returning a character array whose length is the sum of the lengths of the string or character array operands.

   Explain how you would change type checking and code generation for `binaryOpNodes` to handle this extension.