CS 536
Final Exam

Tuesday, May 10, 2005

7:45 AM— 9:45 AM

1325 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. Recall that in building parsers, the **first** relation is key in making parsing decisions. Define InFirst(A,a), where A is a nonterminal and a is a terminal, to be true if it is the case that $A \Rightarrow^+ a\ldots$ (and false otherwise) That is, InFirst(A,a) tells us if a string of symbols beginning with the terminal symbol a may be derived from A.

   Give an algorithm that computes InFirst(A,a) given a grammar G.

2. Many programming languages allow method names to be **overloaded**. That is, the same identifier may be used to name several different methods in the same scope as long as they can be distinguished at the point of call.

   Assume we allow CSX methods to be overloaded as long as each definition that shares the same name differs in the number of parameters it requires. Thus `M()` and `M(x)` and `M(y,z)` could all co-exist in the same class.

   Explain what changes you will need to make in your CSX symbol table and type checker modules to support this limited form of overloading.

3. Assume we have a CSX function whose header is

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

   and a 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.)

4. Assume that we add a simple “for loop” to CSX:

```
    for (id = init; condition)
       stmt
```

   `id` is any integer variable. `init` is an integer-valued expression that is evaluated at the beginning of the loop and assigned to `id`. `condition` is a boolean-valued expression that is evaluated at the start of each iteration. Iteration terminates as soon as `condition` becomes false. Zero iterations are possible. At the end of each iteration `id` is incremented by 1.

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

5. Recall that the JVM evaluates relational expressions like `i==0` or `a > b` using a conditional branch. Show the code your CSX compiler would generate to compute `i != 5`.

   Relational expressions are most commonly used in if statements and while loops. In the case of a statement like `while (i!=5) {` ... `}` it is probably the case that you compute `i != 5` onto the stack (using a conditional branch) and then do a second conditional branch to decide whether to terminate the loop.

   A better approach would avoid directly computing a boolean onto the stack and instead use the conditional branch that implements `i!=5` to also control loop termination. Explain how you’d change your CSX compiler to implement this improved translation scheme. What JVM code would you now generate for `while (i!=5) {` ... `}`?

6. 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)
   $S \rightarrow D\ P\ a$
   $D \rightarrow b$
   $D \rightarrow \lambda$
   $P \rightarrow c$
   $P \rightarrow \lambda$

7. 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.