CS 536

Final Exam

Wednesday, December 18, 2002

12:25 PM— 2:25 PM

B102 Van Vleck

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 that we add a conditional expression of the form
   ($Expr_1$? $Expr_2$: $Expr_3$)
   to CSX.
   $Expr_1$ is an expression that returns a `bool`. If $Expr_1$ is true, expression $Expr_2$ is evaluated, and its value is the value of the conditional expression. If $Expr_1$ is false, expression $Expr_3$ is evaluated, and its value is the value of the conditional expression.

   Outline the changes that would be needed in a CSX type-checker and code generator to handle conditional expressions. Illustrate your answer using the following example:
   ```
   a = (i != 0 ? j/i : 0);
   ```

2. (a) In CSX we type check a call to method M by first type checking M's declaration. Then the actual parameters in the call to M are type checked. Finally, the number, type and kind of parameters found in the call are compared with the number, type and kind of parameters specified in M's declaration.

   Consider the following alternative. In a declaration of a method P, *no types* are given to P's parameters; they are simply given names. For example,
   ```
   int P(a,b,c) { ... }
   ```
   When a call to P is found, the parameters in the call are type checked (as usual) and then these types are used as *definitions* of the types of P's parameters.

   What changes are needed in your type checking of method declarations and calls to implement this change?

   (b) A potential difficulty in the approach suggested in part (a) is that different calls to P may differ in the types used for a particular parameter. How would you handle a non-unique type for a parameter in different calls to P?

3. Assume we add a new operator to CSX, the *delay* operator. The expression

   `delay expr`

   returns a *suspended* execution of `expr` (essentially a function of zero arguments whose body is `return expr`). The operation

   `force suspension`

   forces the given suspension (created by a `delay` operation) to be evaluated, returning the value of the suspended expression.
   For example,

   `delay delete_file(f)`

   creates a suspended call to `delete_file` with parameter `f`. The `delete_file` operation is not done yet, it is just "set up." At some later time the suspension created by the `delay` may be activated by using the `force` operation. Then the `delete_file` operation is performed.

   Explain how you would implement `delay` and `force` in CSX. How can you be sure that all the variables visible when the `delay` was executed are still accessible when the `force` is executed?

4. Let G be any context-free grammar that contains no λ productions. Let X and Y be any two terminal symbols (tokens). Adjacent(X,Y) is a predicate (boolean-valued function) that determines whether X and Y can ever be adjacent in any program defined by G.

   Explain how to implement Adjacent(X,Y). (Java code is not required; just explain the steps needed to test for adjacency.)

5. (a) Consider the following context free grammar:

   S → Label id ( ) :
   S → Label id = id ;
   Label → id :
   Label → λ

   Is this grammar LL(1)? Why? Is this grammar LALR(1)? Why?

   (b) Consider the following context free grammar:

   S → Label id ( ) :
   S → Label id = id ;
   Label → intlit :
   Label → λ

   Is this grammar LL(1)? Why? Is this grammar LALR(1)? Why?

   (c) Consider the following context free grammar:

   S → Label id ( ) :
   S → Label id (Arg) ;
   Label → intlit :
   Label → λ
   Arg → id
   Arg → λ

   Is this grammar LL(1)? Why? Is this grammar LALR(1)? Why?

6. (a) Code for an if statement is generated on the assumption that the value of the control expression will not be known until run-time. In some cases the value of the control expression is known at compile-time. The simplest such case if when the control expression is just the boolean literal `true` or `false`.
What changes would you make in your CSX code generator for if statements to handle the special case of a control expression that is either the literal `true` or the literal `false`?

(b) The special case handled in part (a) is uncommon. More common is the case in which the control expression is an identifier declared to be a boolean constant with a literal initializer. For example,

```
const debug = true;
  ...
  if (debug) ...
```

What changes are needed to your solution to part (a) to include the case of identifiers declared to be boolean constants?

(c) It may occur that the control expression of an if statement contains boolean operators (&&, ||, !) whose operands are all boolean literals or boolean constants with literal initializers. For example,

```
const debug1 = true;
const debug2 = false;
...
if (debug1 || debug2) ...
```

What changes are needed to your solution to part (b) to include the case of boolean operators whose operands are all boolean literals or boolean constants?

7. Assume we have a Java class

```
class K {
   int a;
   int sum(){
      int b=1;
      return a+b;
} }
```

and the call

```
z = (new K()).sum();
```

Explain the run-time steps needed to call and execute `sum()` (parameter passing, frame manipulation, return address manipulation, etc.)