CS 536

Final Exam

Friday, May 15, 1998

7:45 AM— 9:45 AM

2080 Grainger Hall

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 new kind of conditional statement to CSX, the `signtest`. Its structure is

```
signtest ( exp ) {
        neg:  stmts
        zero: stmts
        pos:  stmts
}
```

   The integer expression `exp` is evaluated. If it is negative, the `stmts` following `neg` are executed. If it is zero, the `stmts` following `zero` are executed. If it is positive, the `stmts` following `pos` are executed.

   Show the AST you would use for this construct. Explain how you would generate code for it. Illustrate your answer using the following example.

```
signtest (A+B-3) {
        neg:  print("Expression is negative.\n");
        zero: print("Expression is zero.\n");
        pos:  print("Expression is positive.\n");
}
```

2. 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.

3. Many programming languages, including Java and C++, allow multiple declaration of methods with the same name as long as each declaration differs in the number or type of its arguments. This is called *overloading*.

   Assume we add overloading to CSX by allowing more than one method to be defined with the same name as long as each such definition differs in the number or type of arguments it allows. Explain what changes you'd need to make in your symbol table and type checker modules. Illustrate your answer using the following program:

```
class test {
      int V=0;
      int max(int A, int B){
          if (A > B)
                 return A;
          else  return B;
       }
      bool max(bool A, bool B){
          if (A > B)
                 return A;
          else  return B;
       }
       main() {
          V = max(1,23);
  }    }
```

4. 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 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.)

5. Assume we have a context-free grammar, G. Let A be a nonterminal in G. Explain how to determine whether A generates any terminal string that is exactly n tokens long.

6. Consider the following grammar:

   Prog $\longrightarrow$ { Decls Procs Stmts }
   Decls $\longrightarrow$ INT id ; Decls
   Decls $\longrightarrow$ λ
   Procs $\longrightarrow$ INT id ( ) { Stmts } Procs
   Procs $\longrightarrow$ λ
   Stmts $\longrightarrow$ id ( ) ; Stmts
   Stmts $\longrightarrow$ λ

   This grammar is *not* LALR(1). Explain carefully why it is not LALR(1). Transform this grammar into an equivalent grammar that *is* LALR(1).

7. A common compiler optimization is *inlining*. A call to a method M is inlined when the call is replaced with the body of M, with formal parameters initialized to the actual parameters used in the call.

   What are the advantages of inlining? What are the disadvantages of inlining? Under what circumstances should inlining *not* be performed?

   Given a CSX program in AST form, how would you inline a a particular call of some method M?