Final practice problems

Exam Facts

Wednesday December 13, 8:30-11:30am **Kresge Auditorium** (in the Law School)

There is no alternate exam; everyone is expected to take the exam at the scheduled time. Local SITN students will come to campus for the exam, since that ensures you will be able to ask questions and receive any clarifications that might come up during the exam. SITN folks outside the Bay Area will take the exam at their local site. There is a link to a campus map up on the web page and some information about parking options for those of you coming from off-campus.

Format

The final will be an in-class 3-hour **open-note/closed-book** written exam. We won't ask you to wedge everything you've learned this quarter on one piece of paper, so you can bring class handouts, your own notes from lecture, and printouts of your programming projects to refer to during the exam. You may not use any books or a computer during the exam.

Given it is open-note, the exam will feature fewer short-answer questions whose answers can easily be looked up and more on application of the material, extension beyond the basics, synthesis of bigger concepts, and the like.

Material

The exam will cover the entire quarter, but will focus more on the second half (semantic analysis, code generation, and implementation issues). Material similar to the homework will figure prominently, although you are responsible for all topics presented in lecture and in the handouts. The sorts of things you might expect to see on the exam:

- (see midterm practice handout for list of topics covered prior to midterm)
- Syntax-directed translation —attribute grammars, inherited and synthesized attributes, use of yacc attributes and actions, error-handling
- Type systems — named/structural type-equivalency, strong/weak typing, static/dynamic typing
- Scoping —variable lifetime and visibility, static/dynamic scoping
- Object-oriented issues— type compatibility, polymorphism, inheritance, static/dynamic binding, runtime implementation
- Intermediate representations —abstract syntax trees, TAC, intermediate code generation
- Target machine—processor architectures, memory hierarchies, data representation, instruction sets, assemblers, linkers, loaders
- Runtime issues— function call protocols, parameter-passing, call stack, dynamic memory management, register use
- Optimization—basic blocks, control flow analysis, constant folding and propagation, algebraic identities, operator strength reduction, common sub-expression elimination, code motion
- Language design issues— tradeoffs between efficiency, safety, and expressiveness, CT versus RT decision-making

The rest of this handout is a smattering of problems culled from past CS143 exams and problem sets. Solutions will be posted on our web site later this week. I tried to provide some variety so you could see a range of question formats. These questions mostly focus on second-half topics; for review on pre-midterm material, refer to the handouts, lecture examples, problem sets, and the practice and real midterm questions.

1) The following excerpt from soop.y concerns just function declarations and definitions. An aspiring student working on pp3 is attempting to use an embedded action to push a scope for the parameters before processing the function parameter list and popping that scope at the end of the definition. As expressed below, this code will not work as designed for managing the parameter scope. Briefly explain what the problem is and how to resolve it.

```
FunctionDeclaration : Type T_Identifier '(' ParameterListOrVoid ')' ';'
  {
     $$ = BuildFunctionDecl($2, $1, $4, DeclOnly);
  }
FunctionDefinition : Type T_Identifier { gScopeStack->PushScope(); }
          '(‘ ParameterListOrVoid ')' CompoundStatement
  {
     gScopeStack->PopScope();
     $$ = BuildFunctionDecl($2, $1, $5, FullDefn);
  }
```

2) In handout #17 on grammar design, we argue that it can be preferable to permit “looseness” in the grammar itself and have the semantic analyzer weed out various invalid constructions rather than trying to make the grammar perfectly tight. There are several examples that illustrate this in the SOOP grammar. Choose one and discuss the tradeoffs between trying to restrict the input using the parser instead of the semantic analyzer.

3) In the projects, we suggested you avoid global variables. Believe it or not, this was to make your life easier, not harder. Explain why using global variables to state in a compiler is generally a poor idea and why using the scopestack is a better choice.

4) Let the synthesized attribute *val* give the value of the binary number generated by S in the following grammar. "?

```
S -> L . L | L
L -> LB | B
B -> 0 | 1
```

For example: 101.101, S.val = 5.625. Use synthesized attributes to determine S.val by adding actions to the above grammar.

In case you are a little rusty on binary "fractions": How is it that 101.101 = 5.625? 101 on the left side of the decimal = 5. The ".625" can be calculated by obtaining the decimal value of the (binary) fractional part, and dividing by the next greater power of 2. For example, ".111" is 7 in binary and we divide this by $2^3 = 8$ which is the next power of 2, to give us 0.825. ".11" is 3 which we divide by $2^2 = 0.75$. ".1000" = 8 divided by $2^4 = 0.5$. Finally, the example above ".101" = 5 divided by $2^3 = 0.625$.

5) Draw a picture of the complete SOOP runtime stack and dynamic memory (heap) at the two marked locations in the following program. Show values stored in the stack frames and in memory locations. We do not need to see the register values, and you do not have to worry about temps.

```
class Rectangle {
   int originX;
   int originY;
   int height;
   int width;
   void initRect(int oX, int oY, int h, int w) {
      originX = oX;
      originY = oY;
      height = h;
      width = w;
   }
}

class ColorRectangle: Rectangle {
   int color[3];
   void initColorRect(int oX, int oY, int h, int w, int r, int g, int b)
   {
      color = NewArray(color);
      color[0] = r;
      color[1] = g;        // show state of memory before this line is executed
      color[2] = b;
      initRect(oX, oY, h, w);
   }

   int GetColor(int i) {
      return color[i];     // show state of memory before this line is executed
   }
}

void main(void) {
   class ColorRectangle cr;
   cr = New(ColorRectangle);
   cr.initColorRect(100,100,200,200,255,255,0);
   Print("color is ", cr.GetColor(0));
}
```

6) In pp2 we added the for loop to SOOP, but by the time pp5 rolled around, we no longer included it. However, code generation for it is quite similar to the while loop. Give the sequence of TAC instructions that would emitted from the compiler doing a straightforward (non-optimized) translation of the following for loop:

```
   int i;
   for (i = 0; i < 12; i++)
         Print(i);
   Print("Done.");
```

7) If building a new object is a two-step process where the programmer first allocates memory and then must call a separate routine to initialize it, it is possible for a careless programmer to forget that second step. To help prevent this error, a language can support a unified create-and-initialize operation. In Java and C++, such routines are called *constructors*. They allow you to replace the code on the left with that on the right (the syntax below is from C++):

```
Binky *b = new Binky;
b.Init(3, 2.5, true);
```

```
Binky *b = new Binky(3, 2.5, true);
```

We would like to add constructors to SOOP. To simply matters, there must be exactly one constructor defined in each class, although the programmer can specify the number and type of parameters to the constructor. Inheritance adds some quirks, so we will not deal with this, just assume that you are dealing with classes without inheritance.

a) Describe the syntax and design you are going to use for adding constructors to SOOP. You can adopt the C++ or Java versions or something of your own choosing. Be sure to indicate how the constructor definition for a class is identified and how objects are created. You should also indicate what impact, if any, adding constructors has on the runtime implementation of objects and classes.

b) What changes will you need to make to the scanner to implement constructors?

c) What changes will you need to make to the parser? Be specific in terms of what productions are added, removed, or changed from the original SOOP grammar.

d) What changes will you need to make to the semantic analyzer? Be specific in terms of what validity checks and error messages must be added, removed, or changed. It is preferred that you show this by indicating the actions associated with your productions from part c).

e) What changes will you need to make for TAC code generation? You must generate only ordinary TAC instructions (i.e. you cannot change the TAC instruction set). It is preferred that you show this by indicating the actions associated with your productions from part c).

8) Divide the following TAC code into basic blocks and use arrows to construct the flow-graph.

```
L0:
   w = c + d;
   z = t3 + t4;
   y = t1 + t2;
   IfZ t0 Goto L1;
   t6 = t7 – t4;
   d = t6 + t5;
L2:
   a = 5;
   t8 = a + z;
   Goto L4;
   t6 = t7 – t4;
L1:
   t2 = t3 + t6;
   t6 = t7 – t4;
   Return ;
L3:
   x = t1 + t2;
   t7 = a + z;
   IfZ t7 Goto L2;
```

9) Optimize the basic block of TAC code below in a series of passes using the local optimization techniques described in class: (CP) constant propagation (CF) constant folding, (AI) algebraic identities, (OSR) operator strength reduction, (CSE) common sub-expression elimination, (CPY) copy propagation, (DCE) dead code elimination. Do one optimization in each pass (CP, CSE, etc.), and apply the optimization to the entire block in that pass. Take as many passes as you need to get the code fully optimal, label each pass with the optimization being performed. You do not need to recopy the entire block on each pass, just the sections that change.

```
t0 = 0;
i = t0;
t2 = 4;
t3 = t2 * i;
t1 = a + t3;
t4 = 1;
*t1 = t4;
t5 = 1;
t6 = i + t5;
i = t6;
t8 = 4;
t9 = t8 * i;
t7 = a + t9;
t10 = 2;
*t7 = t10;
t11 = 1;
t13 = 4;
t14 = t13 * t11;
t12 = a + t14;
t16 = 4;
t17 = t16 * i;
t15 = a + t17;
*t12 = *t15;
```