CSE 401 Final Exam

Name _____ID _____

Answer all questions.
Do your own work.
Show your work.
Check your work.

Do Not Start Exam Until Told To Do So

CSE401 Final Exam

1. [10] Circle X of the *earliest* stage at which each MiniJava rule *could* be enforced?

	Scanning	**Parsing** (w/o extra action code)	**Type-checking**	**IR**
curly braces grouping statements are balanced.	X	X	X	X
either **`e`** or **`E`** is used consistently in scientific notation but not both	X	X	X	X
identifier after `extends` refers to previously defined class	X	X	X	X
`int` is followed by a not-previously-declared identifier	X	X	X	X
a **`break`** statement occurs inside a loop	X	X	X	X

2. [4] Describe Java using two of the terms: static, dynamic, weak, strong

3. [6] The invention of intermediate code (IR) was an important milestone in the development of compilers because it made the compiler writing task easier. What are the principle advantages of intermediate code?

4. The original Fortran language had a statement called the Computed Goto:

`GOTO ( `*<line_label_list>*` ) `*<integer_expression>*

which had the following semantics: the integer expression is evaluated and the result is used as an index into the *<line_label_list>*, and control is transferred to that line. Recall that Fortran has user-defined (numeric) line labels, so an example statement would be

```
      X = 1
      GOTO ( 100, 200, 300 ) X+1
10    ...
```

which would jump to the statement labeled `200`. Note that if `X` < 1 or `X` > list length, the Computed GOTO falls through, i.e. executes line `10` in the example.

Suppose that during type check every user defined line label (e.g. `10` in the preceding example) is given a symbolic name (e.g. `L10`) and entered in the symbol table, and that Fortran's data space is all global.

(a) [8] Give correct (it doesn't have to be optimal in any sense) 3-address IR code for the above Computed GOTO statement. (Use the 3 address code of lectures rather than MiniJava lowering language; the compiler can't do arithmetic on line labels.)

(b) [5] Give a pseudo code algorithm to generate code for Computed GOTO statements; the first step is provided:

1. Code gen <*integer_expression*> leaving the result in temporary t.

2.

(c) [6] A Computed GOTO on a Boolean value `B` of the following form amounts to an if statement

```
     GOTO (20, 40) B+1
20   …
```

Apply the algorithm of part 4(b) to this code.

(d) [5] Say (briefly!) what is wrong with the compiled code of part (c).

(e) [5] Give an optimized alternative to part (c).

5. Consider the code at right. (a) [4] In the conditional statement which are more numerous, the defs or the uses? List each of the uses. [Check your work.]

```
span = ub - lb + 1;
space = span * 12;
ex = span %  16;
if (ex == 0) {
    why = 16;
    zed = span - 256;
}
else {
   amt = 16-ex;
   zed = span - 2^ex;
   why = 0;
   space += 12 * amt;
}
vee = zed + 2ex;
System.out.println(vee +
      why);
```

(b) [6] Draw the control flow graph for the code, keeping the diagram to the left side of the paper.

(c) [8] To the right of the control flow graph entries **neatly** give the live ranges, assuming that `ub` and `lb` are live on entry to this section of code.

(d) [8] Give the interference graph using the data from (c).

(e) [10] Copy your interference graph to this page. Apply the recursive coloring algorithm given in class to the interference graph of (d). Because this is a heuristic algorithm you MUST show all work to determine that your outcome is correct. Use numbers for colors. [Abbreviate node labels or maintain geometry to simply drawing.]

6. [20] Given the source code

```
i = 0;
c[i] = a[i]*b[i];
```

(a) give the equivalent standard, unoptimized 3-address code; number each line of generated code.

(b) Give the most optimized equivalent to part (a), *documenting* each line to explain either that it is "unchanged" or which lines it is derived from and by what optimization. We are grading the documentation.

7. [20] The dynamic memory usage of a program uses basically four word units of memory. The plan is to use a Generational Garbage Collector having two generations and four “birthdays” required to become “old”. The standard garbage collection begins by marking all reachable elements in From with a 1 indicating which cells are having another “birthday.” An example From space is shown after this first step. In the space below, complete the garbage collection, drawing whatever structure(s) you need, if any.

From

Mark	1	0	0	1	1	0	1	1	0	0	1	1	0	0	1	0
Previous birthdays	3	3	2	1	2	0	0	0	1	3	3	2	3	0	0	1
Value	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Address x 16	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F