

CSE 401 Final Exam

December 15, 2011

Name _____

You may have one sheet of handwritten notes plus the handwritten notes from the midterm. Otherwise, the exam is closed book, closed notes, closed electronics, closed neighbors, open mind,

Please wait to turn the page until everyone has their exam and you have been told to begin.

If you have questions during the exam, raise your hand and someone will come to you.

Legibility is a plus as is showing your work. We can't read your mind, but we'll try to make sense of what you write.

1	/ 9
2	/ 22
3	/ 16
4	/ 12
5	/ 12
6	/ 12
7	/ 6
8	/ 10
9	/ 1
Total	/ 100

Question 1. (9 points) Compiler phases. For each of the following tasks, indicate the *earliest* stage of the compiler that can *always* perform that task or detect the situation. Assume that the compiler is a conventional one that generates native code for a single target machine (say, x86-64) for a source language with a static type system like C++ or Java. Use the following abbreviations to identify the stages:

scan – scanner

parse – parser

sem – semantics/type check

opt – optimization

instr – instruction selection & scheduling

reg – register allocation

run – runtime (i.e., when the compiled code is executed)

can't – can't always be done during compilation or execution

_____ Detect circular inheritance hierarchies (class A extends B and class B extends A)

_____ Detect possible use of an uninitialized variable in a Java method

_____ Detect that an integer constant is too large to be stored in a variable of type `int`

_____ Decide that using a `leaq` instruction would be the best choice to multiply a value by 5 instead of using an `imul` instruction

_____ In a method call `x.f(...)`, verify that the class of the object referenced by `x` actually does contain a method `f` with the correct number and types of parameters

_____ In a method call `x.f(...)`, verify that `x` actually refers to an object and is not `null`

_____ Discover that the program source file ends in the middle of an incomplete assignment statement like `x=a*b+` (i.e., end of file after the `+` symbol)

_____ Discover that registers do not need to be saved and restored in a function because all calculations can be done with registers the function is allowed to change without restoring

_____ Warn the programmer that the declaration of a method in a subclass is overloading a method declaration in a superclass and not overriding the superclass method as the programmer may have intended

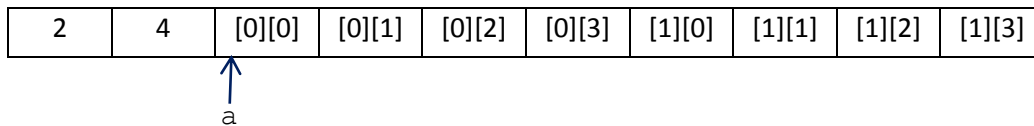
Question 2. (22 points) Compiler hacking: a question of several parts. Vulture Ventures, a new startup in South Lake Union, would like to buy our MiniJava compiler, but only if we add 2-dimensional arrays to it. To do this we will make the following additions to the existing rules in the MiniJava grammar:

```
Type ::= "int" "[" "]" "[" "]"
Expression ::= "new" "int" "[" Expression "]" "[" Expression "]"
Expression ::= Expression "[" Expression "]" "." "length"
Expression ::= Expression "[" Expression "]" "[" Expression "]"
```

This adds a new type for 2-D arrays (`int [] []`), another version of “new” to allocate 2-D arrays, a way to access the number of columns in a 2-D array (`a [row] .length`), and an expression to access individual elements of a 2-D array (`a [row] [col]`). As in regular Java, `a .length` accesses the number of rows in the array. The elements in a 2-D array are 8-byte integer values, as is true of all other integer values in MiniJava.

Our customer requires us to store 2-D arrays as a single block of memory, not as an array of pointers to individual array rows as in regular Java. The array elements are stored in row-major order, as in C, C++, and other languages. When an array is allocated, the `new` operator returns a pointer to element `[0][0]` of the array. The number of rows and columns in the array are stored in the two preceding 8-byte quadwords right before element `[0][0]`.

For example, if we execute `a = new int [2] [4];` the variable `a` will refer to element `[0][0]` in a new array that has 2 rows of 4 elements each. The number of rows (2) and number of columns (4) are stored just before the array elements.



(a) (3 points) What new lexical tokens, if any, need to be added to the scanner and parser of our MiniJava compiler to add these new 2-dimensional arrays to the original MiniJava language? Just describe any necessary changes; you don’t need to give JFlex or CUP specifications or code.

(continued next page)

Question 2. (cont.) (b) (6 points) What changes are needed to the MiniJava abstract syntax tree (AST) data structures to add these new 2-dimensional arrays to MiniJava? Again, you do not need to give any Java or CUP code, just describe the changes (what kinds of new or changed nodes, what children would they have, etc.).

(c) (4 points) Suppose a reference to an array element $e1[e2][e3]$ appears as a value in a MiniJava expression. Describe the checks that need to be performed to verify that this array element expression has no type errors or other static semantic errors.

(continued next page)

Question 2. (cont.) (d) (9 points) Describe the x86-64 code shape that the compiler would generate to evaluate the array element expression `e1 [e2] [e3]` and load the value of the correct array element into register `rax`. Be sure to show where the code to evaluate expressions `e1`, `e2`, and `e3` should appear in the generated code. You should assume that the code generated for expressions `e1`, `e2`, and `e3` will leave the value of the corresponding expression in `rax`, as we did in the MiniJava project. Also, assume that the stack is aligned on a 16-byte boundary at the beginning of the code sequence, and if you change the size of the stack you need to be sure this alignment is preserved if any of the expressions `e1`, `e2`, or `e3` contain a method call.

Use the Linux/AT&T/gcc x86-64 assembler syntax for your code.

Hint: Remember that elements of the array occupy 8 bytes each, and the number of rows and columns in the array are stored in the 8-byte quadwords just prior to array element `[0] [0]`.

Question 3. (16 points) A bit of coding. Consider the following Java function that computes the greatest common divisor of its arguments. This code uses the Java and gcc convention that `long` is a 64-bit integer type.

```
/* return gcd(x,y) for x, y > 0 */
long gcd(long x, long y) {
    if (x > y) {
        return gcd(x-y, y);
    } else if (y > x) {
        return gcd(x, y-x);
    } else /* x==y */
        return x;
    }
}
```

This question involves translating this function to x86-64 assembler code. Ground rules:

- You must use the Linux/AT&T/gcc assembly language, and must follow the x86-64 function call, register, and stack frame conventions.
 - Argument registers: `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9` in that order
 - Called function must save and restore `rbx`, `rbp`, and `r12-r15` if they are used
 - Function result returned in `rax`
 - `rsp` must be aligned on a 16-byte boundary when a `call` instruction is executed
- Pointers and long ints are 64 bits (8 bytes) each.
- Your x86-64 code must implement all of the statements in the original function, including the recursive function calls. You may *not* rewrite the function into a different form that produces equivalent results (i.e., no tail recursion optimizations). Other than that, you can use any reasonable x86-64 code that follows the standard function call and register conventions.
- You are not required to use register `rbp` as a frame pointer register. If you do use it, you must use it correctly.
- Please include *brief* comments in your code to help us understand what the code is supposed to be doing (which will help us assign partial credit if it doesn't do exactly what you intended.)

(You may detach this page from the exam if convenient.)

Question 3. (cont.) Translate function `gcd` to x86-64 assembly language. Be sure to follow the rules given on the previous page (hint: read the rules again after you've finished your answer.) Code repeated for reference.

```
/* return gcd(x,y) for x, y > 0 */
long gcd(long x, long y) {
    if (x > y) {
        return gcd(x-y, y);
    } else if (y > x) {
        return gcd(x, y-x);
    } else /* x==y */
        return x;
    }
}
```

Question 4. (12 points) Register allocation. Considering the following code:

```
a = read();
b = read();
if (a > b) {
    c = read();
    e = c - b;
} else {
    d = b;
    e = a + d;
}
print(b);
print(e);
```

(a) Draw the interference graph for the variables in this code. You are not required to draw the control flow graph, but it might be useful to sketch it out to help find the solution and to leave clues about what might have happened if the graph is not quite correct. (The next page contains additional blank space if you would like to use it, but please put your final answer on this page.)

(b) Give an assignment of (groups of) variables to registers using the minimum number of registers possible, based on the information in the interference graph. You do not need to go through the steps of the graph coloring algorithm explicitly, although that may be helpful as a guide to assigning registers. If there is more than one possible answer that uses the minimum number of registers, any of them will be fine. Use R1, R2, R3, ... for the register names. (Again, there is additional work space on the next page if needed, but please write your answer here.)

Question 4. (cont.) Extra blank space if needed.

Question 5. (12 points) Thinking optimally. In most production compilers, optimization is done using intermediate code that consists of simple 3-address instructions like the following:

$$r5 = r3 + r7$$
$$r6 = r5 * 8$$

Some of the instructions in the intermediate code may be *dead code* because the results of the instructions are never used later in the program. An important optimization in compilers is *dead-code elimination*, where we remove such instructions to save space and execution time.

(a) (8 points) What *data flow analysis* should be used to discover which instructions are dead code? Describe the appropriate data flow problem to use and explain how to use its results to identify dead instruction(s).

(b) (4 points) After performing the data flow analysis in part (a) and removing the dead code that it identifies, will there be any dead code in the remaining instructions? If so, why, and what needs to be done to remove them? If not, why not?

Question 6. (12 points) SSA. Here is a simple program that reads two numbers and prints their greatest common divisor as long as both numbers are positive.

```
x = read();
y = read();
while (x != y) {
    if (x > y) {
        x = x - y;
    } else { /* y > x */
        y = y - x;
    }
}
print(x);
```

Draw a control flow graph for this program (nodes = basic blocks, edges = possible control flow), using Static Single Assignment (SSA) form as described in section. You should include ϕ -functions where needed to merge different versions of the same variable. For full credit you should only include necessary ϕ -functions and not have extraneous ones scattered everywhere, but we will give generous partial credit if all of the necessary ϕ -functions are included and there are a minimal number of extra ones.

Hint: it might be easiest to sketch the flow graph first without converting it into SSA, then add the needed variable version numbers and ϕ -functions to get the final answer.

Write your answer on the next page. You can remove this page from the exam if you wish.

Question 6. (cont.) Write your answer here.

Question 7. (6 points) A little concurrency. In section we discussed the interactions between concurrency and compiler optimizations. Suppose we have the following function that adds the elements of an integer array to the value in global variable `sum`.

```
int sum = 0;

void addarray(int[] a) {
    if (a.length <= 0) return;
    for (int k = 0; k < a.length; k++) {
        sum += a[k];
    }
}
```

We would like to optimize the code by keeping a copy of `sum` in a local register `temp` as follows:

```
void addarray(int[] a) {
    if (a.length <= 0) return;
    temp = sum;
    for (int k = 0; k < a.length; k++) {
        temp += a[k];
    }
    sum = temp;
}
```

(a) (1 point) Is this optimizing transformation always correct and safe in single-threaded code? (circle one)

yes no

(b) (1 point) Assuming that the variable `temp` is a local register that is not shared with other threads, is this optimizing transformation always correct and safe in multi-threaded code (i.e., when other threads are running in the same address space)? (circle one)

yes no

(c) (4 points) Give a brief explanation of your answer to (b) describing why this optimization is or is not safe in multi-threaded code.

Question 8. (10 points, 2 each) The party's almost over. Time to clean up the garbage...

We looked at three major garbage collection algorithms in class: classic mark-sweep collectors, semispace copying collectors, and generational collectors. For each of the following statements or properties, circle the name of the algorithm that is the best match. If more than one algorithm is an equally good "best answer", circle all of the correct answers.

(a) Fastest allocation of new objects (new/malloc/cons):

mark-sweep copying generation

(b) Searches entire heap for reachable objects during a routine collection:

mark-sweep copying generation

(c) Best heap locality (arrangement of objects for good cache and paging behavior) after a collection:

mark-sweep copying generation

(d) Largest percentage of examined objects reclaimed during a routine collection:

mark-sweep copying generation

(e) Best overall performance:

mark-sweep copying generation

Question 9. (1 point) The best Star Trek television series is clearly (circle):

- a) The Original Star Trek
- b) Star Trek: The Next Generation
- c) Star Trek: Deep Space Nine
- d) Star Trek: Voyager
- e) Star Trek: Enterprise
- f) All of the above
- g) Just give me my free point, please

Best wishes for the holidays and New Year!

The CSE 401 staff