UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS 164 **P. N. Hilfinger**
Spring 2009

CS 164: Final Examination (corrected)

Name: _____ Login: _____

You have three hours to complete this test. Please put your login on each sheet, as indicated, in case pages get separated. Answer all questions in the space provided on the exam paper. Show all work (but be sure to indicate your answers clearly.) The exam is worth a total of 50+ points (out of the total of 200), distributed as indicated on the individual questions.

You may use any notes or books you please, but not computers, cell phones, etc.—anything inanimate and unresponsive. We suggest that you read all questions before trying to answer any of them and work first on those about which you feel most confident.

You should have 8 problems on 14 pages.

1. _____/10
2. _____/6
3. _____/5
4. _____/10
5. _____/5
6. _____/
7. _____/4
8. _____/10

TOT _____/50

Login of person to your left: _____

Login of person to your right: _____

1. [10 points] For each of the following possible modifications to a fully functional system for compiling and executing our Python dialect, tell which components of the compiler and run-time system would have to be modified: lexical analyzer, parser and tree-generator, static semantic analyzer, standard prelude, intermediate-code generator, machine-language code generator, and run-time libraries. In each case, indicate a *minimal* set of components from this list that would have to be changed, and indicate *very briefly* what change would be needed. When you have a choice of two equal-sized sets of modules that might reasonably be changed, choose the one that makes for the simplest change or whose modules appear earlier in the list (e.g., prefer changing the lexical analyzer to the parser, if either change would be about equally difficult).

a. Allow the following program to execute indefinitely without blowing up:

```
while True:
   L = []
   i = 0
   while i < 1000:
      L.append (i)
      i += 1
```

b. Allow arbitrary delimiters on strings, analogously to PHP, so that

   <<<*DELIM*<*CHARS*>*DELIM*>>>

   is a string containing *CHARS*, where *DELIM* is *any* sequence of characters other than '<' and '>' that does not appear anywhere in *CHARS*.

c. Introduce the built-in function `hex`, which converts any integer into a hexadecimal string.

d. Allow attribute references `E.x` (including in method calls, as in `E.x(...)`) only in cases where `E` is guaranteed either to have the value `None` or have a value for which the `x` attribute is defined, rejecting other references before any execution of the program.

e. Allow programmers to write things like

```
def f():
    body of f
enddef
```

so that one may optionally mark the end of a **def** with the keyword **enddef** at the same indentation level as the **def**.

2. [6 points] The term *extended BNF* refers to ordinary BNF augmented with regular-expression-like constructs. In this problem, we'll look at one form of it and how it can be converted to ordinary BNF. In this version, one can write rules with standard right-hand sides such as

```
A : B ;
A: 'c' D ;
D : ;       /* EMPTY */
A : Q | R 'f' ;
```

plus extended forms such as this:

```
A1 : B* ;           /* "An A1 may be formed from 0 or more Bs" */
A2 : B ( C | D ) ;  /* "An A2 may be formed from a B followed by
                       either a C or a D." */
A3 : B ( C 'd' | E )* ;
```

and so forth. These new forms add no new power, since we can convert them to ordinary BNF. For example, we could rewrite A2's definition as

```
A3 : B Q ;
Q : C | D ;
```

In this problem, you are to write a Bison grammar for this extended BNF notation (a *meta-grammar*) that translates it into regular BNF (that's right, we're going to parse Bison with Bison). To make this easy (or at least easier), we are going to allow your program to work iteratively. There will be a global variable, G, that holds sets of strings and that the semantic actions of your Bison program may modify. Each string in this set is to contain a Bison rule (either in plain Bison BNF or extended BNF). We will use your grammar as follows:

```
def Convert(aGrammar):
    gram = aGrammar
    G0 = the empty set
    while True:
        G = the empty set
        Parse gram with your Bison program
        gram = the grammar resulting from concatenating \
               all the strings in G
        if G == G0: return gram
        G0 = G
```

On the next page, write a grammar that recognizes an extended BNF definition as described above and whose semantic actions set G to an equivalent set of BNF rules (the same set if there are no extended BNF constructs in the input). Assume that the lexer returns the strings it matches as the semantic values (i.e., `$1`, etc.) of all terminal symbols. Assume in particular that the terminal symbol SYM matches any ordinary Bison terminal or nonterminal symbol (yes, that means it would allow input like

```
'c' : 'd' ;
```

but just assume we only care about correct input). Finally, assume that there is a function, `gensym()`, that your actions may call to create new, previously unused nonterminal symbols. We won't be fussy about the notation you use in actions for string or set operations.

```
Grammar: /* Empty */
       | Grammar Rule
       ;

    /* Fill in the rest: All semantic values are strings, and actions
     * may modify the global variable G, a set of strings.   Add any
     * other nonterminals you need. */
Rule:
```

3. [5 points] We'd like to make conservative estimates of the maximum values of integer variables in a program. We'll assume a vastly simplified, statically typed programming language. The only expressions are C (for C any integer literal constant), v (for v an integer variable), and $E_1 + E_2$ (for E_1 and E_2 expressions). The only statements are

Assignments: $v = E$;

Conditionals: `if` $E_1 \leq E_2$ `then` S_1 `else` S_2 `fi`, where S_1 and S_2 are lists of statements;

Exit: `return`.

At any point in the program, we'd like to know (conservatively) the maximum possible values of all integer variables. At any such point, your analysis should give each variable either a constant upper bound (like 42) or ∞, indicating that you cannot determine an upper bound. For example, your analysis should be able to annotate the following program as shown in its comments:

```
  # x <= ∞; i <= ∞
if i <= 99 then
   # i <= 99; x <= ∞
   x = i + 1
   # i <= 99; x <= 100
else
   # x <= ∞; i <= ∞
   x = 2
   # x <= 2; i <= ∞
   i = 102
   # x <= 2; i <= 102
fi
# i <= 102; x <= 100
return
```

a. How might a compiler use the results of such an analysis?

Problem continues on next page.

b. Describe an appropriate analysis along the lines of the flow analyses described in lecture.

c. Give an example in which your analysis gives an overly conservative bound on a variable.

d. What would go wrong with your analysis if we introduced **while** loops into this language?

4. [10 points] Consider the Java code below:

```
class Base {
    public void e() { /*e*/ }
}

class A extends Base {
    public void f(int r) { /*f_a*/ }

    public int x;
}

class B extends A {
    public void f(int r) { /*f_b*/ }
    public void g() { /*g*/ }

    public int y;
    public int z;
}

void call_f(A anA)
{
    anA.f(anA.x);
}

A a = new A();
B b = new B();

call_f(a);
call_f(b);
```

This problem involves the implementation of the object-oriented mechanisms for this program. The language is Java; you should *not* introduce anything into your solution except what is needed for implementing Java (which, unlike Python, is completely statically typed). In part (b) below, we ask you to produce three-address code. Use the following notation, where r_i indicates a register, O_i indicates a virtual register or immediate constant, and L_i indicates a label:

r_0 := O_1	
r_0 := O_1 $\oplus$ O_2	where $\oplus$ is one of the arithmetic operators.
r_0 := *($r_1 + N$)	Load r_0 from memory whose address is $r_1 + N$, where N is an integer constant.
*($r_0 + N$) := O_1	Store O_1 to address $r_0 + N$.
push O_1	pushes argument for function call.
call *O_1	call to function whose address is value of O_1.
if O_1 $\prec$ O_2 goto L_0	conditional jump ($\prec$ is a comparison operator).
goto L_1	unconditional jump.
L_2:	Defines label.

Questions begin on the next page.

a. Show what the objects pointed to by `a` and `b` look like. Also show any relevant virtual method tables. Choose a representation in which virtual method tables are as small as possible—that is, just big enough to implement the classes `Base`, `A`, and `B` in Java.

b. Provide an implementation of the body of function `call_f`, excluding the function prologue and epilogue. Assume that the parameters to a function are in registers p_0, p_1, etc.

c. In Project #3, we did *not* make virtual tables as small as possible, but allocated space for all method names in the entire program. Below we've added the use of an interface to the code sample from above. Fill the blanks to give a definition of class `YourClass`, followed by a use of it (statements or method declarations) that demonstrates how the virtual method dispatch scheme for single inheritance you used for part (a) is insufficient for interface method dispatch. Briefly explain why it is insufficient, and how the strategy used in Project #3 solves the problem.

```
interface I1 {
    public void f(int r);
}

class Base {
    public void e() { /*e*/ }
}
```

```
class A extends Base implements I1 {
    public void f(int r) { /*f_a*/ }

    public int x;
}
```

```
class B extends A implements I1 {
    public void f(int r) { /*f_b*/ }
    public void g() { /*g*/ }

    public int y;
    public int z;
}
```

```
class YourClass _____ {

    _____

    _____

    _____

    _____
}
```

_____

_____

_____

_____

5. [5 points] Consider the following program:

```
while i > 0:
    c = a*b + i
    if c > b + i:
        a = c
    c = 0
    i = i - 1
```

Assume that all quantities are known to be machine integers and that only the variable `a` is live after the loop.

a. Convert this into the three-address code from problem 4 above. Assume that `a`, `b`, `c`, and `i` all denote registers and introduce any new registers you need.

b. In your answer to (a), identify the basic blocks, clearly showing the first and last statement of each, and indicating edges between them in the CFG.

c. Show which registers are alive at each point in the program (that is, at the beginning and before each assignment and each conditional jump). If this information indicates that statements can be eliminated, indicate where.

6. [1 point] Name a poem in which old trees in a forest are compared with ancient Celtic priests.

7. [4 points] In a statically typed language, consider a **let** statement that, as in Scheme, allows any number of variable definitions:

```
let
   x = a * z,
   y = q ** 5
in
   f (x * y)
```

Also, somewhat similarly to Scheme, we evaluate all the right-hand sides between the **let** and **in** keywords independently and then define the left-side identifiers to have those values (as constants) during the evaluation of the body—the expression after the **in** keyword. The value of the body is the value of the **let** statement as a whole. Someone proposes the following typing rules for this construct:

```
typeof(let([],Body), T, Env) :- typeof(Body, T, Env).
typeof(let([bind(X, Expr) | Defs], Body), T, Env) :-
    typeof(Expr,T1,Env), typeof(let(Defs, Body), T, [def(X,T1)|Env]).
```

where, for example, the AST for the sample **let** statement above would be

`let([bind(x,` *AST for a*z*`), bind(y,` *AST for q**5*`)],`
*AST for f(x*y)*)

a. What's wrong with this rule? Give an example where it rejects a program with no type errors or accepts an incorrect one. (Make any reasonable assumptions you want about the rest of the language.)

b. Revise the rule to fix the problem.

8. [10 points] For each of the following questions, provide a short, succinct answer.

a. How can you determine reliably whether a particular input has more than one parse? Will this tell you whether you have an ambiguous grammar?

b. In calling a function in the project, one must generally provide a count of the number of arguments, either as an operand of the CALL instruction (in the intermediate language) or as an explicit parameter (in ia32 code). Why is this necessary in our dialect of Python, and why is it *not* necessary in Java (unless using exotic features such as `java.lang.reflect`)?

c. Consider a statically typed and statically scoped language in which all variables contain either integers or some kind of object pointer (but not functions), and in which function declarations may be nested. What language feature or combination of features still requires that we provide static links (or equivalent mechanism) at execution time and why?

d. Typical Java implementations store a virtual-table pointer in each object and use this for method calls. What language features of full Python require a different approach and why?

e. I'd like to implement a feature of a certain highly dynamic language that allows me to change the set of instance variables and methods of all objects of a given type after these objects have been created. Imagine that in Java, after evaluating `new Foo(···)` some large number of times, I decide to modify the class `Foo` during execution of the program, add some instance variables to it, change some of its methods, and then continue executing so that some of the updated methods get called with objects created before the change. How might I do this? In particular, what existing mechanisms in Java might I be able to use to bring this about?