CS 164: Final Examination

Name: ________________________________  Login: ______

Please do not discuss the contents of this test with anyone before the test on Saturday, 22 May 1999.

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 65+ points (out of the total of 200), distributed as indicated on the individual questions.

You may use any notes, books, or computers you please—anything inanimate. 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 6 problems on 14 pages.

1. __________/13  
2. __________/13  
3. __________/13  
4. __________/13  
5. __________/ 
6. __________/13

TOTAL __________/65
1. [13 points] For each of the following possible modifications to a fully functional Java system, tell which components of the compiler and run-time system would have to be modified: lexical analyzer, parser and tree-generator, static semantic analyzer, code generator, and runtime libraries. In each case, indicate a minimal set of modules 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 modules that might reasonably be changed, choose the one that makes for the simplest change or the earlier in the list (e.g., prefer changing the lexical analyzer to the parser, if either change would be about equally difficult).

   a. Allow abstract classes to be instantiated, but cause any call to an abstract method (one to which no body has ever been given) to raise a run-time exception (that is, some unchecked exception that need not be included in throws clauses).

   b. As in C and C++, allow adjacent string literals (but not variables or other string-valued expressions) to mean the same thing as their concatenation. For example, allow programmers to write `System.out.println("Hello" "world")` in place of `System.out.println("Hello world");`

   c. Introduce expressions like `x/y for ArithmeticException ? 1`, meaning “evaluate `x/y` and yield its value, but if it throws `ArithmeticException` (as for integer divide by 0), yield the value 1 instead.”
d. Introduce the notation ‘for (int i = 0, N-1, 1)...’ as a synonym for ‘for (int i = 0; i < N-1; i += 1)....’

e. When A is an array, cause A.toString () and "A is " + A to render A as something like ‘{1,2,3}’ rather than the current situation, where it looks like ‘[I@1fa4d40d’.

f. Disallow the use of plain field names (e.g., ‘currentSize’) in methods, requiring instead that one always write, e.g., this.currentSize (when currentSize is a field of this, that is).
2. [13 points] We can represent grammars written in a simplified version of Bison with abstract syntax trees as follows:

- A grammar with $n$ terminal symbols and $m$ rules is represented by a tree of the form

$$\text{grammar} \ (\text{tokens} \ T_0 \ T_1 \cdots T_{n-1}) \ (\text{rules} \ R_0 \cdots R_{m-1})$$

where each $T_i$ is an identifier, and each $R_i$ is a tree representing a rule. [Here, I am using Lisp list representation for trees: the operator comes first (e.g., grammar), followed by 0 or more children. Leaves are simply identifiers.]

- The rule \( L \rightarrow X_1 \ X_2 \cdots X_k \) is represented by the tree

$$\text{rule} \ L \ X_1 \ X_2 \cdots X_k$$

where $k \geq 0$, and $L$ and each $X_i$ are identifiers.

- Identifiers are sequences of letters, digits, and underscores, starting with a non-digit.

The input corresponding to these trees is simplified from the full Bison syntax as follows:

- All symbols (terminal and non-terminal) are identifiers; tokens such as \( ';' \) are not allowed.

- There are only \%token declarations before the first \%symbol, as in

\[
\%token \ IDENTIFIER \ NUMBER \ LE \ GE \ EQ \ NE
\]

(There can be any number of such lines.) There are no \%left, \%right, \%union, or other declarations.

- There are no \%prec clauses.

- There are no semantic rules (in \{\} braces).

- There is no second \% symbol.

- As in real Bison, the semicolon at the end of each rule is optional (an obscure, but true fact).

- Comments (\/* */ style) are allowed anywhere whitespace is allowed.

Problems begin on the next page.
Login:

a. Assuming that all semantic values returned by the lexer are strings that simply contain the text of the tokens themselves, write an appropriate set of rules (Lex/Flex-style) for the lexical analyzer.
b. Write a Bison grammar for these simplified Bison programs (that is, describe simplified Bison using regular Bison). Supply it with semantic actions to build the trees described above. Assume that you have the following list-building operations:

- **nil()** the empty list.
- **cons(X, L)** where $X$ is a string or list, and $L$ is a list, the list whose first item is $X$ and whose remaining items are $L$.

**WARNING:** The abstract syntax trees described above make no provision for the ‘|’ symbol. You must somehow translate, e.g., ‘A: B | C’ into the two rules ‘(rule A B)’ and ‘(rule A C)’.
c. Suppose that we restrict ourselves to LL(1) grammars as inputs to our simplified Bison system. Give an algorithm for converting the abstract syntax trees coming out of such a parser into a recursive-descent parser (written in a language like C). I am looking only for reasonably high-level pseudo code here, things like

```plaintext
Convert (Grammar G) {
    T = the set of all identifiers in the "token" list of G;
    N = the set of all left-hand side symbols in the "rules"
        list of G;
    ...
}
```

or

```plaintext
for every symbol, s, in N,
    output the header "void s () {";
    ...
```

You can assume, in other words, that you've got things like lists, sets, strings, and other data structures already defined. Assume also that the lexical analyzer is written for you; choose an interface to it and describe it. You are allowed to assume that there are library procedures for computing FIRST and FOLLOW from a set of rules.
3. [13 points] Consider a vastly simplified version of the abstract syntax trees used in your projects, in which the only statement and expression nodes are PlusNode, EQNode, ArrayAccessNode, NameNode, ObjectNode, AssignNode, BlockNode, IfStmtNode, EmptyStmtNode, StmtDeclNode, IntLitNode, TrueNode, FalseNode, and MethodCallNode; there is only one kind of declaration node, AnyDeclNode; and the only type nodes are IntTypeNode, BoolTypeNode, ArrayTypeNode, MethodDeclNode, and TypeVarNode. You can’t represent entire programs with this, but we’re not going to worry about that. An AnyDeclNode has a name (simpName) and a type (dtype) and nothing else. A TypeVarNode has a type (TypeNode) attribute called binding, which is initially null (indicating unbound); it represents a type variable and its binding can be set to any type.

The problem is to write a method findTypes() that can take a statement or expression from the subset described above (since BlockNode is included, this includes sequences of statements as well) that may or may not include StmtDeclNodes for all the variables used in it and check that all variables and functions are used consistently. For example, these trees can represent the following Java program fragment:

```java
x = y;  y = y + 1;
if (y == z)
    q[3] = f(r);
else
    q = x;
int x;
```

Unlike regular Java, we allow declarations to follow uses (as shown by the last declaration). Your findTypes() method should figure out that x, y, and z are integers, that f is a method taking a single argument (all methods will be static), whose type is the same as that of r. It should also figure out that there is an error, since q must be both an array and an integer, which is not possible.

Assume that initially all names have an appropriate defn attribute that points to an AnyDeclNode (so you don’t have to worry about resolving names). Assume also that the dtype of each AnyDeclNode is an (unbound) TypeVarNode (a different one for each AnyDeclNode), and that the type attributes of all expression nodes are also unique, unbound TypeVarNodes. All that findTypes() has to do, in other words, is bind TypeVarNodes as much as possible and issue error messages. Assume that there is a function unify(T₁, T₂) that returns true iff T₁ and T₂ (which must be kinds of TypeNodes) unify, and as a side effect, may bind unbound type variables (TypeVarNodes) in T₁ and T₂ as needed to make them unify (once a TypeVarNode is bound, it retains its binding and never becomes unbound).

Define findTypes() for all the relevant node types. Feel free to use any abbreviations you want for creating new nodes and accessing children; we are not going to be fussy. For example, you can use the pre-defined types TypeInt and TypeBool from Project 3. Just use ERROR() to indicate that an error message should be produced. Do not change any children; unify will produce all the side-effects you need for this problem.
void TrueNode::findTypes () {
    if (! unify (type (), TypeBool))
        ERROR ();
}

void FalseNode::findTypes () {
    if (! unify (type (), TypeBool))
        ERROR ();
}

void PlusNode::findTypes () {
    }

void EQNode::findTypes () {
    }

void ArrayAccessNode::findTypes () {
    }

void ObjectNode::findTypes () {
    }

void AssignNode::findTypes () {
    }

void BlockNode::findTypes () {
    }

Continued on next page.
void EmptyStmtNode::findTypes () {} 

void IfStmtNode::findTypes () {
    void StmtDeclNode::findTypes () {
        void MethodCallNode::findTypes () {
            if (!unify (type (), TypeInt))
                ERROR ();
        }
    }
}

void IntLitNode::findTypes () {
    void MethodCallNode::findTypes () {
    }
}
4. [13 points] The Java virtual machine’s instructions for inputting and outputting from fields, for reading elements of arrays, and for calling methods all include checks that the pointer value being used is not null. In an interpreter, this isn’t such a bad thing, but in a system that generates machine code from Java, these null-pointer checks can require extra code that causes significant slowing of the translated programs’ execution speed. Consider a program fragment such as

```java
if (a.m > z)
    a.x = 0;
    a.r = null;
    a = f();
    a.m = a.m + 1;
```

Each of the ‘a.’ accesses is supposed to throw a NullPointerException if a is null. However, only the first reference to a (in the if condition) and the right-hand side reference to a in a.m+1 actually need to be checked. All of the other ‘a.’ accesses will succeed if these succeed.

For concreteness, assume that programs have been converted into basic blocks containing the usual sorts of three-address statements:

1. r1 = r2 op r3 // op any arithmetic operator
2. r1 = r2
3. r1 = C // C a constant
4. r1 = r2.f // where f is a field name
5. r1.f = r2
6. if r1 op r2 goto L // op is a relational operator
7. goto L
8. r1 = call g // g a function name (don’t worry about parameters)

Assume that all local variables are in registers.

Turn the problem of figuring out what null-pointer checks are unnecessary into a data-flow problem, as follows:

a. Determine what information must be passed around (i.e., what kind of values the IN and OUT functions should have);
b. Justify your choice for (a) by showing how, given the information you compute at the beginning of any statement that loads from or stores into a field, we can use it to figure out if the compiler needs to generate null-checking code;

c. Show what data-flow equations to use to compute the information, and how to define the appropriate KILL and GEN functions for all kinds of statements.
5. [1 point] Who was Doroteo Aranga?

6. [13 points] For each of the following questions, provide a short, succinct answer.
   
   a. Give the smallest DFA you can that recognizes the same language as the following grammar (start symbol ‘S’, lower-case letters are terminals):

   \[
   S \rightarrow P \\
   S \rightarrow a \ S \mid b \ S \\
   P \rightarrow a \ a \ b
   \]

   b. Java differentiates primitive types (int and so on) from reference types (those derived from Object), so that although one can assign any kind of reference value to a variable of type Object, there is no type of variable to which one can assign either an Object or an int. Numerous irregularities result: for example, you can write \(x\).toString() if \(x\) is a reference value, but must write \(String.valueOf(x)\) if \(x\) is an int. What considerations kept the language’s implementors from treating primitive types and reference types more uniformly?
c. The C and C++ languages have *untagged union types*, so that one can define, for example,

```c
union { int x; double y; } U;
```

and then assign integers to `U.x` and floating-point values to `U.y`, but can’t use both `x` and `y` at the same time. `U` is of a fixed size that is big enough to hold either an integer or a floating-point value. The programmer must keep track of which field of `U` is currently valid. Exactly how do union types interfere with garbage collection? Without eliminating the feature entirely, how could it be changed so as not to interfere?

d. Typical debuggers allow one to step through a program one statement at a time and to examine the value of local variables at any point in the program. However, typical debuggers either do not work at all on optimized programs or behave strangely. Give two examples of “classical” optimizations that might make it difficult for debuggers to perform their typical functions.