CSE P 501 Exam 8/5/04

Name _____

There are 7 questions worth a total of 65 points. Please budget your time so you get to all of the questions. Keep your answers brief and to the point.

You may refer to the following references:

Course lecture slides and notes
Homework assignments and sample solutions
Your primary compiler textbook (presumably Appel)

No other books or other materials.

Please wait to turn the page until everyone is told to begin.

Score _____

_____ 1

_____ 2

_____ 3

_____ 4

_____ 5

_____ 6

_____ 7

1. (10 points) Write a regular expression or regular expressions that generate the following sets of strings.

(a) (5 points) All strings containing a's, b's, and c's with at least one a and at least one b.

(b) (5 points) All strings of 0's and 1's with at most one pair of consecutive 1's.

2. (8 points) Pascal defined real (floating-point) numeric constants as follows:

digit ::= [0-9]
digits ::= *digit*+
real ::= *digits* . *digits* | *digits* . *digits* E *scalefactor* | *digits* E *scalefactor*
scalefactor ::= *digits* | (+ | -) *digits*

Draw a DFA that accepts real constants as defined above. (You don't need to construct an NFA and use algorithms to convert it to a DFA – just draw a suitable DFA diagram.)

3. (12 points) (LR parsing) In languages like Scheme, arithmetic expressions are written as parenthesized lists beginning with an operator followed by the operands. Here is a simple grammar for expressions involving addition and subtraction and the single integer constant 1.

exp ::= *int* | (*op exp exp*)
op ::= + | -
int ::= 1

(a) (10 points) Construct the LR(0) state machine for this grammar.

(b) (2 points) Is this grammar LR(0)? Why or why not?

4. (6 points) Some experimental programming languages include an *n*+1/2 loop with the following syntax:

```
repeat
  statement1
while ( exp )
  statement2
end
```

The semantics of this loop is that *statement1* is executed then the conditional expression *exp* is evaluated. If *exp* is false, the loop terminates, otherwise, if it is true, *statement2* is executed and the loop repeats starting with *statement1*.

Give the x86 code shape for this loop using a style similar to that used in lecture for other statements and conditional expressions.

5. (12 points) Suppose we want to add the following conditional statement to MiniJava:

```
ifequal (exp1, exp2)
  statement1
smaller
  statement2
larger
  statement3
```

The meaning of this is that *statement1* is executed if the integer expressions *exp1* and *exp2* are equal; *statement2* is executed if *exp1* < *exp2*, and *statement3* is executed if *exp1* > *exp2*.

(a) (5 points) Give context-free grammar production(s) for the `ifequal` statement that allows either or both of the "`smaller`" and "`larger`" parts of the statement to be omitted. If both the "`smaller`" and "`larger`" parts of the statement appear, they should appear in that order.

(b) (5 points) Is the grammar with your production(s) from part (a) ambiguous? If not, argue informally why not; if it is ambiguous, give an example that shows that it is.

(c) (2 points) When compiling this statement, what rule(s) or condition(s) should the type checker verify?

6. (5 points) Suppose we have two classes `B` and `D`, where `D` is a subclass (derived class) of `B`, and these classes contain the methods shown below.

```
class B {
   void f() { … }
   void g() { … }
   void h() { … }
}

class D extends B {
   void k() { … }
   void g() { … }
}
```

Recall that our convention is that in generated x86 assembly code, the label for a method m in class `C` is `C$m`.

Show what the generated virtual method dispatch tables for classes `B` and `D` would look like in x86 assembly language. (Hint: this is supposed to be an easy question – don't over-analyze it.)

7. (12 points) Write an x86 assembly-language version of the following C function. Your answer doesn't need to use exactly the same code shape presented in class for the various statements and expressions (i.e., it only needs to be legal x86 code that works properly), but you do need to use the C function-calling conventions properly, i.e., push arguments onto the stack, set up a new stack frame, etc., and you must include assembly language code for all statements given here (i.e., don't omit the assignment to the local variable `a`, for example).

```
int factorial(int n) {
   int a;
   a = n;
   if (a <= 1) {
      return 1;
   } else {
      return a * factorial(a-1);
   }
}
```