CS164: Midterm I

Fall 2003

- Please read all instructions (including these) carefully.
- **Write your name, login, and circle the time of your section.**
- Read each question carefully and think about what's being asked. Ask the proctor if you don't understand anything about any of the questions. If you make any non-trivial assumptions, state them clearly.
- Some questions span multiple pages. Be sure to answer every part of each question.
- You will have 1 hour and 20 minutes to work on the exam. The exam is closed book, but you may refer to your two pages of handwritten notes.
- Solutions will be graded on correctness and clarity, so make sure your answers are neat and coherent. Remember that if we can't read it, it's wrong!
- Each question has a relatively simple and straightforward solution. We might deduct points if your solution is far more complicated than necessary. Partial answers will be graded for partial credit.
- Write all of your answers in the space provided on the exam, and clearly mark your answers. Use the backs of the exam pages for scratch work. Do not use any extra scratch paper. Do not unstaple the exam.
- Turn off your cell phone.
- Good luck!

NAME and LOGIN: _____

Your SSN or Student ID: _____

Circle the time of your section: 9:00 10:00 11:00 2:00 3:00

Q1 (30 points)	Q2 (30 points)	Q3 (30 points)	Q4 (10 points)	Total (100 points)

1 Regular Expressions/Finite Automata

Part (a)

Consider a language over the alphabet $\{0, 1\}$ consisting of the strings that meet the following conditions:

- The length of the strings is 6.
- There must be at least one 1 in an odd position (1, 3, or 5) that is immediately followed by a 1 in an even position (2, 4, or 6).

For example, 110000, 000011, and 101111 are all in the language; 00000, 001010, and 100110 are not.

Write a regular expression that defines this language:

Part (b)

Consider a language over the alphabet $\{0, 1\}$ consisting of the strings that meet the following conditions:

- The length of the strings is 6.
- The last two characters must both be zero.

For example, 110000, 001100, and 001100 are all in the language; 000011, 001010, and 111001 are not.

Write a regular expression that defines this language:

Part (c)

Draw two deterministic finite-state machines, one for the language defined in Part (a), and one for the language defined in Part (b). When drawing the state machines, use the appropriate notation for nodes and edges. Organize your node layouts neatly; the layout should reflect the logic of your solution.

Part (d)

Draw a non-deterministic finite-state machine for the language that is the union of the languages defined in Parts (a) and (b). (A string belongs to a language that is the union of languages A and B if the string belongs to A or B or both A and B.)

Part (e).

Draw a deterministic finite-state machine for the language that is the union of the languages defined in Parts (a) and (b). Also, write a string that is *not* accepted by your finite-state machine.

2 First/Follow/LL(1)

This question concerns the context-free grammar given below (where capital letters denote non-terminals, small letters denote terminals).

$$S \rightarrow A\ B \mid B\ A\ B$$
$$A \rightarrow S\ A \mid y\ x$$
$$B \rightarrow x \mid \epsilon$$

Part (a)

Fill in the following table with First and Follow sets for the grammar.

X	$FIRST(X)$	$FOLLOW(X)$
S		
A		
B		
$A\ \ B$		
$B\ \ A\ \ B$		
$y\ \ x$		
x		
ϵ		

Part (b)

In the $LL(1)$ parsing table below, fill in the column headings, and the row corresponding to the non-terminal A:

A			

Part (c)

Is the grammar $LL(1)$? Justify your answer.

3 A Simple Real-World Problem

Background: Recall that Java provides the `++` operator, which can be used either as a pre-increment or as a post-increment operator. That is, `++` can be used to increment the value of a variable either before or after the value of the variable is used. For example, if the value of `x` is initially 1, then the expression `x++` evaluates to 1 and the expression `++x` evaluates to 2.

Also recall that `x++` is a legal Java expression, but `3++` is not. That is, you can increment a memory location (e.g., a variable) but you cannot increment a value. Note that the error in `3++` is detected in the semantic phase of the compiler.

Part (a) Draw an AST for the expression below. Label each AST node clearly with the meaning of the node (for example, "addition," "identifier," etc). Invent new types of AST nodes as necessary.

```
x++ + ++x
```

Part (b) It is interesting to observe that while

```
x++ + ++x
```

is a legal Java expression, the same expression without white spaces, namely

```
x+++++x
```

is not a legal Java expression. That is, the latter expression will cause a compile-time error. This question asks you to identify the phase of the compiler in which the error occurred. Depending on the compiler, the error can be flagged in different stages, and so there is more than one correct answer. (Continued on the next page.)

(i) Was the error found in the lexer? If **yes**, clearly explain the cause of the lexical error, and move on to Part (c) on the next page. If **not**, write the output of the lexer for this input expression and continue to (ii). (Assume that the lexer uses three kinds of tokens: the + operator, the ++ operator, and an identifier.)

(ii) Was the error found in the parser? If **yes**, clearly explain the cause of the syntactic error, and move on to Part (c) on the next page. If **not**, write the output of the parser (i.e., an AST) for this input expression and continue to (iii).

(iii) Was the error found in the semantic checker? If yes, clearly explain the cause of the semantic error. (Clearly, since you got all the way to this part, the answer must be **yes**.) To show the error, all you need to do is evaluate the AST from (ii) under the assumption that the initial value of `x` is 1, and show where and why the evaluation would fail. Evaluate the AST just like you did in your PA1 interpreter.

Part (c) Let's assume that when the programmer wrote the expression `x+++++x`, he really meant to compute the expression `x++ + ++x`. Unfortunately, the first Java expression is not equivalent to the second expression with the whitespaces (as discussed above, the first expression is not even a legal Java expression). Clearly, in Java, white spaces are sometimes very important.

One solution how to make the compiler recognize `x+++++x` as equivalent to `x++ + ++x` without having to insert any whitespaces into the expression is to move part of the functionality of the lexer to the parser.

Specifically, let us propose that the ++ operator won't be recognized in the lexer; instead, the lexer would simply turn each + character into a **plus** token. So, the lexer recognizes only two kinds of tokens: **id** and **plus**. As usual, the lexer discards any white space from the input.

It would then be up to the parser to group **plus** tokens into ++ operators. Below is a grammar for such a parser. This grammar describes a tiny subset of the language of arithmetic expressions (this language includes additions, pre-increments and post-increments):

$$E \rightarrow T \text{ plus } E \mid T$$
$$T \rightarrow \text{id} \mid \text{id plus plus} \mid \text{plus plus id}$$

(i) Show the parse tree for the input string `x+++++x`.

(ii) Let's now show that the proposed solution unfortunately doesn't work — because the grammar is ambiguous. Use the input `x+++x` to show that the grammar is ambiguous.

Why does ambiguity in a grammar for a programming language pose a problem?

4 Lexer Generator

The code below is a fragment of a solution for the second programming assignment (the lexer generator). The code shows a fragment of a visitor object that traverses a Java AST that represents a regular expression and creates an NFA that represents the regular expression. Specifically, the code shows a method that implements the regular expression + operator (i.e., the one-or-more repetition) as well as the * operator (i.e., the zero-or-more repetition). The code for the second operator is not shown. Your task is to fill in this missing code.

```
public boolean visit(PostfixExpression s) {
   // the traversal of the child s fills in the values of start, end
   // which are instance fields of the visitor object.
   s.getOperand().accept(this);
   NFAState childStart = start, childEnd = end;

   //              Java ++ operator corresponds to regexp + operator
   if (s.getOperator() == PostfixExpression.Operator.INCREMENT) {
          start = new NFAState();
          end = new NFAState();
          start.addTransition(NFAState.EPSILON, childStart);
          childEnd.addTransition(NFAState.EPSILON, end);
          childEnd.addTransition(NFAState.EPSILON, childStart);
   }
   //              Java -- operator corresponds to regexp * operator
   else if (s.getOperator() == PostfixExpression.Operator.DECREMENT) {
          // ADD YOUR CODE HERE









   } else
          assert false;
   return false;
}
```