CS143 Midterm
Fall 2008

- Please read all instructions (including these) carefully.
- There are 4 questions on the exam, some with multiple parts. You have 75 minutes to work on the exam.
- The exam is closed book, but you may refer to your four sheets of prepared notes.
- Please write your answers in the space provided on the exam, and clearly mark your solutions. You may use the backs of the exam pages as scratch paper. Please do not use any additional scratch paper.
- Solutions will be graded on correctness and clarity. Each problem has a relatively simple and straightforward solution. You may get as few as 0 points for a question if your solution is far more complicated than necessary. Partial solutions will be graded for partial credit.

NAME: ____

In accordance with both the letter and spirit of the Honor Code, I have neither given nor received assistance on this examination.

SIGNATURE: ____

Problem	Max points	Points
1	20	
2	30	
3	20	
4	30	
TOTAL		

1. **Lexical Analysis** (20 points)
   Consider the following flex rules:

```
\n                { printf("1") }
A                 { printf("2") }
a                 { printf("3") }
d                 { printf("4") }
[b-d]*            { printf("5") }
abc               { printf("6") }
[A-C]*            { printf("7") }
[a-c]*            { printf("8") }
[^a-c]+d[^A-C]+   { printf("9") }
```

   Here + means “one or more” and * means “zero or more”, as usual.

   What will be the output on the following input? Assume there is nothing to the right of the last visible character on each line except a newline character and that the following input is to be scanned all at once:

```
ABCdabc
Aabcd
BCdbA
bc
abccdba
```

```
The correct answer:
926921518531
using the tokenization:
ABCdabc\n A abc d\nBCdb A \n bc \n abcc db a \n

Also accepted:
912641921518531
using the tokenization:
ABCdabc \n A abc d \n BCdb A \n bc \n abcc db a \n

The most common mistakes were forgetting to use maximal munch or rule
precedence.

Note that [^a-c] and [^A-C] both match the newline character, which is why
the second solution above is technically incorrect.  However, newlines are
treated specially in some contexts by lexing tools, and we decided not to
to deduct points for this technical detail.  The exception is that we did
take off points if you were inconsistent --- if you included the
```

newline in [ˆa-c]/[ˆA-C] once but not both times.

We also accepted any solution which did not put an extra newline at the very end of the input (i.e. did not include the final ’1’).

2. **Grammars** (30 points)
Consider the following language: strings that can be split into $k \geq 0$ substrings, where each substring is $i \geq 1$ a's followed by i b's; different substrings may use different values of i. For example, this language includes the strings ϵ, ab, $aabbab$, and $abaaabbbab$.

- Give an LL(1) grammar for this language; prove your grammar is LL(1).

The solution to this problem requires the grammar and an LL(1) parse table for it showing no conflicts. The parse table is needed to prove the grammar is LL(1); a grammar that is unambiguous, is not left recursive and has been left factored might still contain conflicts in its LL(1) parsing table.

One solution is

```
S -> aTbS | e
T -> aTb | e
```

with the table

	a	b	$
S	aTbS		e
T	aTb	e	

An alternate solution is

```
S -> TS | e
T -> aU
U -> b | Tb
```

with the table

	a	b	$
S	TS		e
T	aU		
U	Tb	b	

There were two common mistakes in this problem. First, many people had a grammar that recognized the wrong language, such as S -> aSbS | e. These grammars can recognize strings not in the target language, such as aababb. To recognize the right language, at least two nonterminals are needed.

Second, many people used a grammar that was ambiguous, and thus not LL(1), such as S -> TS | e, T -> aTb | e. This grammar has an infinite number of possible derivations for the string e.

- Give an ambiguous grammar for this language. Prove your grammar is ambiguous.

  The solution to this problem requires the grammar and two different parse trees or their derivations for the same string.

  ```
  S -> TS | e
  T -> aTb | e
  ```

  Two derivations for different parse trees on string e:

  ```
  S -> e
  S -> TS -> S -> e
  ```

  The only common mistake here was giving a grammar that recognized the wrong language, such as S -> SaSbS | e. Some people used the right language in part a) and the wrong language here, or vice versa. At least two nonterminals are needed whether the grammar is ambiguous or not.

3. **Grammars and Semantic Actions** (20 points)

Consider the following grammar and associated semantic actions.

```
S -> ABCD               { x = 11 * x + 1; }
D -> d                  { x = 7 * x + 1; }
C -> cc                 { x = 5 * x + 1; }
B -> Bb                 { x = 3 * x + 1; }
B -> b                  { x = x + 1; }
A -> gBa                { x = 2 * x + 1; }
```

The variable `x` is global—i.e., all semantic actions update the same `x`. Assume `x` is initialized before parsing to 0. What is the final value of `x` in a bottom-up parse of the following input string:

`gbbabbccd`

For full credit, show how you derived your answer.

```
The rightmost derivation is shown in the right column, the production (reduction)
used is shown in the middle column, and the semantic action is shown in the
right column.  Since a bottom-up parser traces a rightmost derivation in reverse,
the actions are performed in order from the bottom to the top of the column.

S          ->       S -> ABCD       11 * 1093 + 1 = 12024 <- answer
ABCD       ->       D -> d           7 * 156  + 1 = 1093
ABCd       ->       C -> cc          5 * 31   + 1 = 156
ABccd      ->       B -> Bb          3 * 10   + 1 = 31
ABbccd     ->       B -> b                  9 + 1 = 10
Abbccd     ->       A -> gBa         2 * 4    + 1 = 9
gBabbccd   ->       B -> Bb          3 * 1    + 1 = 4
gBbabbccd  ->       B -> b                  0 + 1 = 1
gbbabbccd

The most common mistakes were: tracing a leftmost derivation and having a
rightmost derivation but evaluating the semantic actions in the wrong order.
While it was unnecessary to construct the parsing automaton to answer the
question, some people did so and we did not penalize that.  Similarly, quite a
few people made arithmetic errors, which we did not penalize if the order of
the semantic actions was clear to us.
```

4. **Short Answer** (30 points)
For each of the following questions, give a short, concise answer. We are definitely looking for simplicity and clarity in addition to correctness.

- Give a regular expression and a CFG that accept the same infinite language of your choice.

```
There are many possible answers, but a simple one is:

regular expression:   a*
context-free grammar: S -> Sa | epsilon

A common mistake was writing a grammar that had no strings in the language at
all because it lacked a base case where a non-terminal was replaced by
only terminals (e.g., the grammar S -> Sa).
```

- For any n, show how to construct a CFG that has a single string x in its language but at least n derivations for x.

```
There are many possible answers.  A straightforward one is
S -> A1 | ... | An
A1 -> x
...
An -> x

Another common class of answers gave grammars with infinitely many derivations:
S -> Ax
A -> A | epsilon
```

- Give a regular expression that, for any $n \geq 0$, has 2^n strings of length n in its language.

```
(0+1)*

An answer such as (ab)* is incorrect --- the language has only even
length strings.
```

- Give two implementations for deciding whether an NFA accepts an input string.

```
There are at least three possible implementations:
(1) Convert the NFA to an equivalent DFA and run the DFA on the input.

(2) Simulate the execution of the NFA, making arbitrary choices for
non-deterministic choices and backtracking to undo choices that do not lead
to acceptance of the input string.

(3) Simulate the execution of the NFA, keeping track at each step of the set
```

of states the NFA could be in.

- True or false: The smallest DFA that accepts a given language may be exponentially larger than the smallest NFA that accepts the same language. Why?

  `True. The subset of states construction can result in exponentially many more states in the DFA than in the NFA. (Note this argument is just that, an argument and not a proof.)`