

Name: _____

USC ID: _____

**Midterm Exam
CS 410, Fall 2001 [Bono]**

October 18, 2001

There are 7 problems on the exam, with 80 points total available. There are 8 pages to the exam, including this one; make sure you have all of them. If you need additional space to write any answers, you may use the backs of pages (just direct us where to look).

Put your name and SSN at the top of the exam. Please read over the whole test before beginning. Good luck!

	value	score
Problem 1	8 pts.	
Problem 2	10 pts.	
Problem 3	6 pts.	
Problem 4	15 pts.	
Problem 5	10 pts.	
Problem 6	16 pts.	
Problem 7	15 pts.	
TOTAL	80 pts.	

Problem 1 [8 pts.]

Part A [2]. `bison` creates a _____ parser.
(your answer will be a kind of parser)

(Note: there is more than one answer that will be accepted.)

Part B [2]. `flex` is a tool for automatically building a

_____.
from _____.

Part C [2]. Give a regular expression equivalent to
`a+`
that does not use the `+` operator:

Part D [2]. Give a regular expression equivalent to
`a*`
that does not use the `*` operator (but it may use `+`).

Problem 2 [10 pts.]

Use subset construction to build a DFA equivalent to the NFA below. Show your work.

	a	b	ϵ
1	{ 2 }	{ 3 }	{ 3 }
2	{ }	{ 4 }	{ 1 }
3	{ 4 }	{ 3 }	{ }
<u>4</u>	{ 4 }	{ }	{ }

Problem 3 [6 pts.]

Show that the following grammar is ambiguous.

$$\begin{aligned}
 A &\rightarrow A x A \\
 &| y A \\
 &| z
 \end{aligned}$$

Problem 4 [15 pts.]

Consider again the ambiguous grammar from the previous problem:

$$\begin{array}{l}
 A \rightarrow A x A \\
 \quad | \quad y A \\
 \quad | \quad z
 \end{array}$$

Part A. Do any grammar transformations necessary to make the grammar suitable for LL(1) parsing. Label the resulting grammar(s) with the transformation(s) you applied (or say “same” if no transformations were necessary).

Part B. Attempt to create the LL(1) parse table for the final grammar you gave in part A. Since the grammar is ambiguous there will be one or more conflicts. Show the complete table; for places where we can put multiple values in a table entry, show all the values.

Show your work.

Circle and label your answers to each of the parts.

Problem 5 [10 pts.]

Find the FIRST and FOLLOW sets for each of the non-terminals in the following grammar. Show your work.

(in the grammar below ϵ denotes epsilon, the empty string)

$$\begin{array}{l}
 S \rightarrow B D \\
 \quad | \quad A D
 \end{array}$$

$$\begin{array}{l}
 A \rightarrow Y \\
 \quad | \quad z
 \end{array}$$

$$\begin{array}{l}
 B \rightarrow u B \\
 \quad | \quad \epsilon
 \end{array}$$

$$\begin{array}{l}
 C \rightarrow C v C \\
 \quad | \quad w
 \end{array}$$

$$\begin{array}{l}
 D \rightarrow C x \\
 \quad | \quad B x \\
 \quad | \quad \epsilon
 \end{array}$$

Problem 6 [16 pts.]

Consider the following grammar:

```
E' → E
E → E x T
E → E x
E → y T
T → y T
T → z
```

Part A [14]. Show the DFA for recognizing viable prefixes for this grammar, including the sets of items associated with each state. Number each of the states (such that 0 is the start state).

Note: we have already augmented the grammar for you.

Part B [1]. Which of the states from part A will have **accept** entries in their rows of the action table? (give state numbers)

Part C [1]. Which of the states from part A will have *reduce* entries in their rows of the action table? (give state numbers)

Space for Part A answer:

Problem 7 [15 pts.]

Write the grammar and semantic rules for creating an AST for the Dispatch expression in Cool.

The assumptions you can make:

- Assume we don't have static dispatch or dispatch on an implicit `self`. Here is the syntax for this version of dispatch, using the grammar-like syntax that appeared in CoolAid:

$$Dispatch \rightarrow Expr.id(Expr, *)$$

The special notation $A, *$ means zero or more A 's separated by commas. Comma as a separator means that the last A in the list is not followed by a comma.

- Assume someone else already wrote the rest of the grammar and semantic rules for `Expr` including the rule that generates `Dispatch`: $Expr \rightarrow Dispatch$
- Your semantic rules will compute the attribute `Dispatch.tree`. You may assume the `Expr` rules compute the attribute `Expr.tree`.
- Also, assume that we can get to the semantic value of the lexeme `id`, with the expression `id.value`, and that this value has type `Symbol`.
- Furthermore, assume we have the following constructors to build relevant pieces of the AST:

```
o Expression dispatch(Expression object,
                      Symbol funcName,
                      ExprList actualParams);

// dispatch: creates AST for dispatch expression where
//      -- object is the expression to the left of the dot
//      -- funcName is the name of the method and
//      -- actualParams is a list of actuals in the
//      order they were given.

o ExprList empty(); // returns an empty expression list

o ExprList cons(Expression first, ExprList rest);
// cons returns a list that has "first" prepended to the
// front of "rest".
// Thus the resulting list is one longer than "rest"
```