CS 536 Midterm Exam
Spring 2013

ID: _____

Exam Instructions:

- *Write your student ID (not your name) in the space provided at the top of each page of the exam.*
- *Write all your answers on the exam itself.*
- *Feel free to use the backs of the pages for scratch work. If you need more space for scratch work (or if you'd prefer not to use the backs of the pages), you may use a blue book.*
- *A copy of the Mini grammar is provided to you for reference.*
- *When you are finished, turn in this exam as well as your page of notes (and blue book, if you used one). Make sure your name is on your page of notes.*

	Points	Out of
Question 1 Part (a)		6
Part (b)		6
Question 2		10
Question 3 Part (a)		15
Part (b)		4
Part (c)		4
Part (d)		4
Question 4		15
Question 5 Part (a)		24
Part (b)		12
Total		100

Question 1 (12 points)

This question concerns the language of *words*. A *word* contains vowels and consonants and must follow these rules:

1. Every word must have at least one vowel.
2. No word can have two vowels in a row.

Part (a) Draw a deterministic finite-state machine for the language of words as described above. Label the start state "S" and use a double circle to indicate the final state(s). You may label the edges `vowel` or `cons` (meaning consonant).

Part (b) Write a regular expression for the language of words. You may use `vowel` and `cons` in your regular expression.

Question 2 (10 points)

Suppose we extend the Mini language to allow MATLAB-like `for` loops. A Mini `for` loop statement has the following structure:

```
for (id = begin : incr : end) {
   // body of for loop
}
```

where *id* is an identifier and *begin*, *incr*, and *end* are expressions that evaluate to integers.

The Mini `for` loop works the same as the following Java code:

```
for (id = begin; id <= end; id += incr) {
   // body of for loop
}
```

The increment value (*incr*) is optional (if it is not present, the default increment of the `for` loop is 1), so the Mini `for` loop:

```
for (id = begin : end) {
   // body of for loop
}
```

is equivalent to the Java code:

```
for (id = begin; id <= end; id += 1) {
   // body of for loop
}
```

The body of the Mini `for` loop statement has the same format as the body of a Mini `if` statement.

Suppose we have added the following two tokens to our Mini grammar:

Token name	JLex Pattern
COLON	`":"`
FOR	`"for"`

What new productions will need to be added to the Mini grammar to incorporate this extension? A copy of the Mini grammar is provided. Use lower-case names for nonterminals and use the added tokens as well as tokens already given in the grammar. The productions you add may make use of any nonterminals already given in the grammar.

Write your productions so they are easy to understand; do **not** make them LL(1). Note that, consistent with Mini grammar provided, you do not need to deal with issues of operator associativity or precedence.

Put your answer on the next page.

Put your answer for Question 2 on this page

Question 3 (27 points)

Part (a) Suppose we have defined two new binary operators `?` and `$`. **Write an unambiguous context-free grammar** for the language of expressions with integer literal operands and two operators `?` and `$`. The language should also allow parenthesis, as usual. Here are some examples to let you figure out the precedences and associativities:

expression	equivalent fully-parenthesized expression
`1 ? 2 ? 3`	`(1 ? 2) ? 3`
`1 $ 2 $ 3`	`1 $ (2 $ 3)`
`1 $ 2 ? 3`	`(1 $ 2) ? 3`
`1 ? 2 $ 3`	`1 ? (2 $ 3)`
`1 $ 2 ? 3 $ 4`	`(1 $ 2) ? (3 $ 4)`
`1 ? 2 $ 3 ? 4`	`1 ? (2 $ 3) ? 4`

Part (b) Using the grammar you wrote for part (a), **draw the parse tree** for the input sequence:

```
1 ? 2 $ (3 ? 4)
```

Part (c) If the grammar you wrote for part (a) has any immediate left recursion, apply the transformations learned in class to remove it, and write the result below. Give the *entire* grammar, not just the rules you transformed.

Part (d) If the grammar you wrote for part (b) needs left factoring, do that and write the result below. Give the *entire* grammar, not just the rules you transformed.

Question 4 (15 points)

Below is a non-LL(1) context-free grammar for a simple language of declarations and assignments:

```
program    → varDeclList  stmtList

varDeclList → varDeclList  varDecl
            | varDecl

varDecl    → type  ID  ;

type       → INT
            | BOOL

stmtList   → stmtList  stmt
            | stmt

stmt       → ID  =  exp  ;
            | IF  (  exp  )  {  varDeclList  stmtList  }

exp        → ID
            | INTLIT
            | exp  +  exp
            | exp  *  exp
            | exp  ==  exp
```

One simple optimization a compiler can perform (on languages that don't allow pointers) is to remove code that *declares* or *defines* a variable that never gets *used* in the program. For example,

```
int x;          // x is declared
int y;          // y is declared
int z;          // z is declared
x = 7;          // x is defined
y = x + 3;      // y is defined, x is used
z = 2*x + 4;    // z is defined, x is used
if (z == 3) {   // z is used
   y = 5;       // y is defined
}
// y is never used, so we could remove the declaration of y
// and the two statements that define y
```

In order to perform this optimization, the compiler needs to know which variables get *used* (as opposed to just *declared* and/or *defined*) in a program. You are to define a syntax-directed translation rule for each of the CFG's productions so that the translation of a program is a list of the variables *used* in the program. Assume that the value of an ID token (`ID.val`) is the name of that identifier and the value of an INTLIT token (`INTLIT.val`) is the actual integer value.

You may use the following `List` operations (assume that `L1` and `L2` are lists and `N` is a name):

`new List( )`	creates a new (empty) list
`new List(N)`	creates a new list containing `N`
`L1.add(N)`	adds `N` to list `L1`
`L1.remove(N)`	removes `N` from `L1`
`union(L1, L2)`	creates a new list containing all the names in `L1` and `L2` (with duplicates removed)

Write your translation rules in the table below. (If there is a nonterminal that is not relevant to the translation, you do not need to write a translation rule for grammar productions with that nonterminal on the left-hand side.) For a grammar production with a left-hand side nonterminal *x*, your translation rule should be of the form "*x*.trans = "

Grammar rule	Translation rule
program → varDeclList stmtList	
varDeclList → varDeclList varDecl	
varDeclList → varDecl	
varDecl → type ID ;	
type → INT	
type → BOOL	
stmtList → stmtList stmt	
stmtList → stmt	
stmt → ID = exp ;	
stmt → IF (exp) { varDeclList stmtList }	
exp → ID	
exp → INTLIT	
exp → exp + exp	
exp → exp * exp	
exp → exp == exp	

Question 5 (34 points)

Below is a context-free grammar for a language of simple blocks of code:

block → LCURLY declList stmtList RCURLY

declList → decl declList | ε

stmtList → stmt stmtList | ε

decl → TYPE ID SEMI

stmt → ID EQUALS exp SEMI
| IF LPAREN exp RPAREN block ELSE block
| RETURN exp SEMI

exp → ID | INT

Part (a) Fill in the *FIRST* and *FOLLOW* sets for the nonterminals below:

Non-terminal *X*	***FIRST*(*X*)**	***FOLLOW*(*X*)**
block		
declList		
stmtList		
decl		
stmt		
exp		

Part (b) Recall that we can think of a predictive parser as building the parse tree for its input top-down. Each time the parser's top-of-stack symbol is a nonterminal *X*, it pops *X* and pushes the right-hand side of some grammar rule $X \rightarrow \alpha$. This corresponds to growing the parse tree by giving the leftmost *X* leaf in the tree one child for each symbol in α.

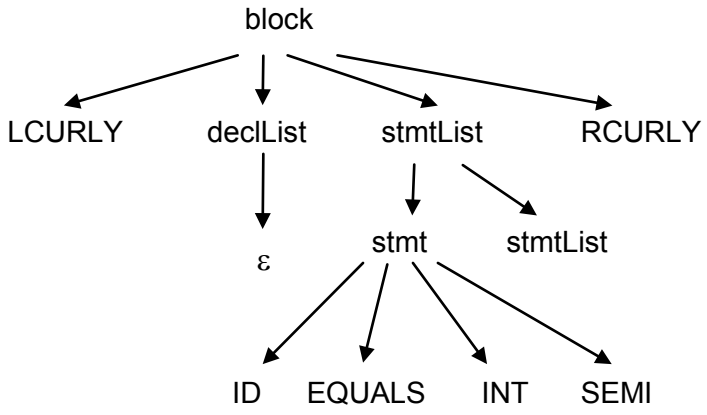
Below and on the next page are three incomplete "snapshots" of moments during a parse of the grammar given on the previous page. Each snapshot includes the input being parsed, the current token, the current symbols in the parser's stack, and the current partial parse tree build by the parser. Your job is to fill in the missing parts of each snapshot.

Snapshot 1:

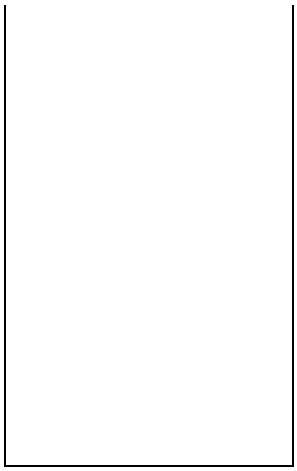
input: LCURLY ID EQUALS INT SEMI RETURN ID SEMI RCURLY

current token: RETURN

partial parse tree



current stack



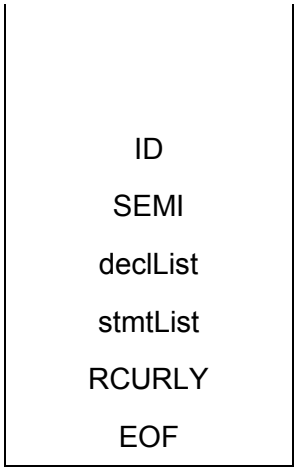
Fill in the contents of the current stack.

Snapshot 2:

input: LCURLY TYPE ID SEMI RCURLY

current token: ID

partial parse tree *current stack*



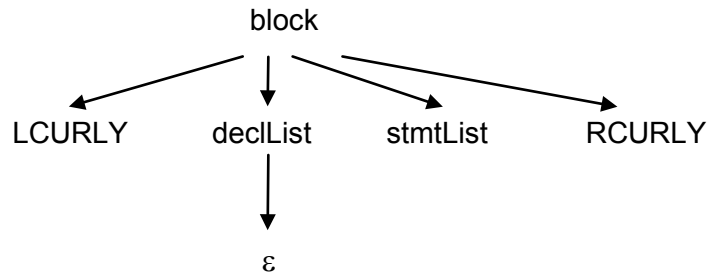
Fill in the partial parse tree.

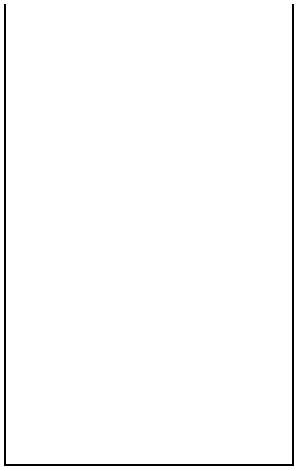
Snapshot 3:

input: _____

current token: RCURLY

partial parse tree *current stack*





Fill in the input and the contents of the current stack.