Question 1 (18 points). Regular Expressions & DFAs

(a) (8 points) Give a regular expression that generates all binary strings with at least one 1 and no more than two 0s.

Fine print: You must restrict yourself to the basic regular expression operations covered in class and on homework assignments: *rs* , *r*`|`*s* , *r*`*` , *r*`+` , *r*`?`, character classes like `[a-cxy]` and `[^aeiou]`, abbreviations *name=regexp*, and parenthesized regular expressions. No additional operations that might be found in the "regexp" packages in various Unix programs, scanner generators like JFlex, or programming language libraries are allowed.

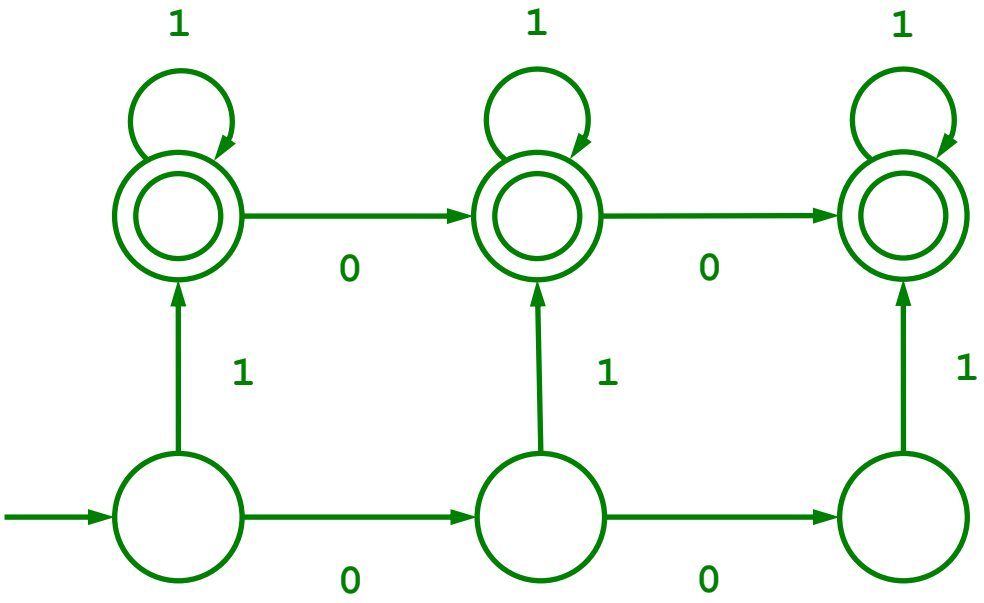
(1+ 0? 1* 0? 1*) | (1* 0? 1+ 0? 1*) | (1* 0? 1* 0? 1+)

Or more concisely,

(1+ 0? 1* 0? 1*) | (1* 0? (1+ 0? 1*) | (1* 0? 1+))

(b) (10 points) Draw a DFA that accepts all binary strings generated by the regular expression from part (a). (It is possible, but not necessary, to do this with only 6 states)

Here is one possibility:



Question 2. (10 points) Scanners. A scanner ought to be able to process any sort of text input, regardless of whether the file contents are actually a program in the language the scanner was intended to process.

What happens if we use a scanner for MiniJava to process the following input?

```
def mult_list(a):
    res_ = []
    while len(a) // 2 > len(res_):
        # /*
        res_.append(a[0] * a.pop())
        # /* */
    return res_
```

Below, list in order the tokens that would be returned by a scanner for MiniJava as it reads this input. If there is a *lexical* error in the input, indicate where that error is encountered by writing a short explanation of the error in between the valid tokens that appear before and after the error(s) (e.g. "UNEXPECTED(%)" would be fine). The token list should include any tokens found after any error(s) in the input, i.e., scanning should continue after discovering an error. You may use any reasonable token names (e.g., LPAREN, ID(x), etc.) as long as your meaning is clear.

A copy of the MiniJava grammar is attached as the last page of the test. You should remove it from the exam and use it for reference while you answer this question. You should assume that the scanner processes MiniJava syntax as defined in that grammar, with no extensions or changes to the language. Also recall that the MiniJava project defines an <IDENTIFIER> as a sequence of letters, digits, and underscores, starting with a letter, and uppercase letters are distinguished (different) from lowercase; and an <INTEGER_LITERAL> is a sequence of decimal digits not starting with 0, or the number 0 by itself, denoting a decimal integer value.

```
ID(def) ID(mult_list) LPAREN ID(a) RPAREN
UNEXPECTED(:)
ID(res_) BECOMES LBRACK RBRACK WHILE ID(len) LPAREN ID(a) RPAREN
UNEXPECTED(#)
RETURN ID(res_)
```

Question 3. (12 points) Ambiguity. We have been asked to help implement a new Hardware Design Language. Part of the language grammar describes operations on bits, including the operations AND and XOR, but not the operation OR:

B:: = `0` | `1` | *B* `&` *B* | *B* `^` *B*

Notes: Whitespace in the grammar is only for readability and is not part of the grammar or the strings generated by it. Similar to OR, XOR (`^`) is commonly understood to have weaker precedence than AND (`&`).

(a) (5 points) Show the grammar is ambiguous. (To do so, you may use any valid method that we have seen.)

One possible solution is to create two parse trees for some string. In this case we use the string "0 & 1 | 0". Students can also show ambiguity using two distinct leftmost or rightmost derivations.

B* => *B* & *B* => *1* & *B* => *1* & *B* ^ *B* => *1* & *1* ^ *B* => *1* & *1* ^ *1
B* => *B* ^ *B* => *B* & *B* ^ *B* => *1* & *B* ^ *B* => *1* & *1* ^ *B* => *1* & *1* ^ *1

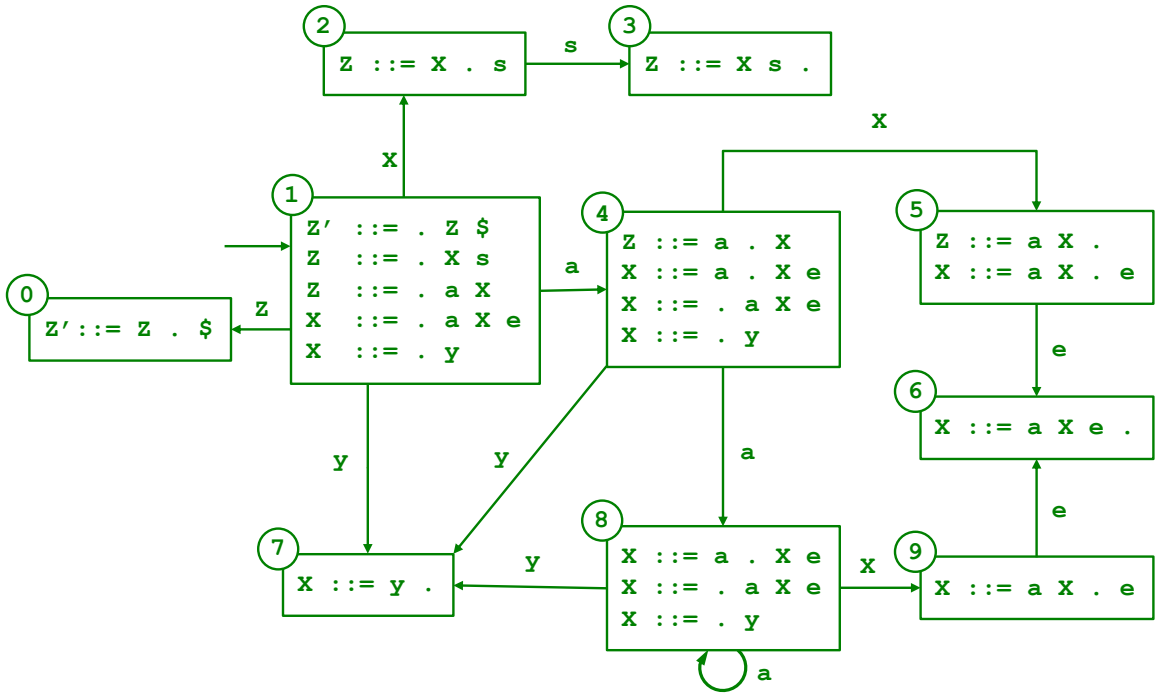
(b) (7 points) Resolve the ambiguity in the grammar by providing an equivalent grammar that generates the same language. Your grammar should follow standard precedence and associativity rules. i.e. AND (`&`) has higher precedence than XOR (`^`); both are left-associative.

B*:: = *B* ^ *A* | *A
A*:: = *A* & *L* | *L
***L*:: = `0` | `1`**

Question 4. (36 points) The "OMG! Not this again!!" parsing question. Consider the following grammar. The nonterminal *Z* is the start symbol of the grammar, and the extra *Z'* ::= *Z* $ rule needed to handle end-of-file in an LR parser has been added for you.

0. *Z'* ::= *Z* $ ($ is EOF)
1. *Z* ::= *X* s
2. *Z* ::= a *X*
3. *X* ::= a *X* e
4. *X* ::= y

(a) (16 points) Draw the LR(0) state machine for this grammar. When you finish, you should number the states in the final diagram in whatever order you wish so that you can use the state numbers in later parts of this question. The state numbers should be successive integers starting with 0, 1, 2, 3, ….



(continued on next page)

Question 4. (cont.) Grammar repeated from previous page for reference:

0. Z' ::= Z $ ($ is EOF)
1. Z ::= X s
2. Z ::= a X
3. X ::= a X e
4. X ::= y

(b) (8 points) Write the LR(0) parser table for the LR parser DFA shown in your answer to part (a). To save time, an empty table is provided below. However, it probably has more rows than you need. Use only as many rows as needed and leave the rest blank.

State #	a	e	s	y	$	*X*	*Z*
0					**acc**		
1	**s4**			**s7**		**g2**	**g0**
2			**s3**				
3	**r1**	**r1**	**r1**	**r1**	**r1**		
4	**s8**			**s7**		**g5**	
5	**r2**	**r2, s6**	**r2**	**r2**	**r2**		
6	**r3**	**r3**	**r3**	**r3**	**r3**		
7	**r4**	**r4**	**r4**	**r4**	**r4**		
8	**s8**			**s7**		**g9**	
9		**s6**					
10							
11							
12							
13							
14							

(continued on next page)

Question 4. (cont.) Grammar repeated from previous pages for reference:

0. Z' ::= Z `$` (`$` is EOF)
1. Z ::= X `s`
2. Z ::= `a` X
3. X ::= `a` X `e`
4. X ::= `y`

(c) (3 points) Is this grammar LR(0)? Explain why or why not. Your answer should describe **all** of the problems that exist if the grammar is not LR(0) by identifying the relevant state number(s) in your answers to parts (a) and (b) and the specific issues in those state(s) (i.e., something like "state 47 has a shift-reduce conflict if the next input is `blah`", but with, of course, correct state numbers and details from your parser). If the grammar is LR(0), you should give a technical explanation why it is (this can be brief).

No. There is a shift-reduce conflict in state 5 on the input 'e'.

(d) (6 points) Complete the following table showing the FIRST and FOLLOW sets and nullable for each of the nonterminals in this grammar. You should include $ (the end-of-file marker) in the FOLLOW set for any nonterminal where it is appropriate.

	FIRST	FOLLOW	nullable
X	**a,y**	**s,e,$**	**no**
Z	**a,y**	**$**	**no**

(e) (3 points) Is this grammar SLR? Give a brief technical explanation why or why not.

Yes. e is not in FOLLOW(Z). Once we omit the reduction in state 5 given input e we no longer have a shift-reduce conflict.

Question 5. (8 points) Top-down parsing. Take another look at the grammar from the previous problem, but omitting the *Z'* ::= *Z* $ rule that was added for LR parsing:

1. *Z* ::= *X* s
2. *Z* ::= a *X*
3. *X* ::= a *X* e
4. *X* ::= y

(Recall that *Z* is the start symbol for this grammar.)

Is this grammar suitable for constructing a top-down LL(1) predictive parser? If so, explain why. If not, explain why not, and, if possible, construct a different grammar that generates the same language that is suitable for a top-down LL(1) predictive parser, or explain why this can't be done.

No, this grammar does not satisfy the LL condition because the Z productions do not have disjoint FIRST sets. We can substitute the X productions into the first production (and group with the second production) to get

Z* ::= a *X* e s | y s | a *X

Then, we can left-factor the productions for *Z* to arrive at

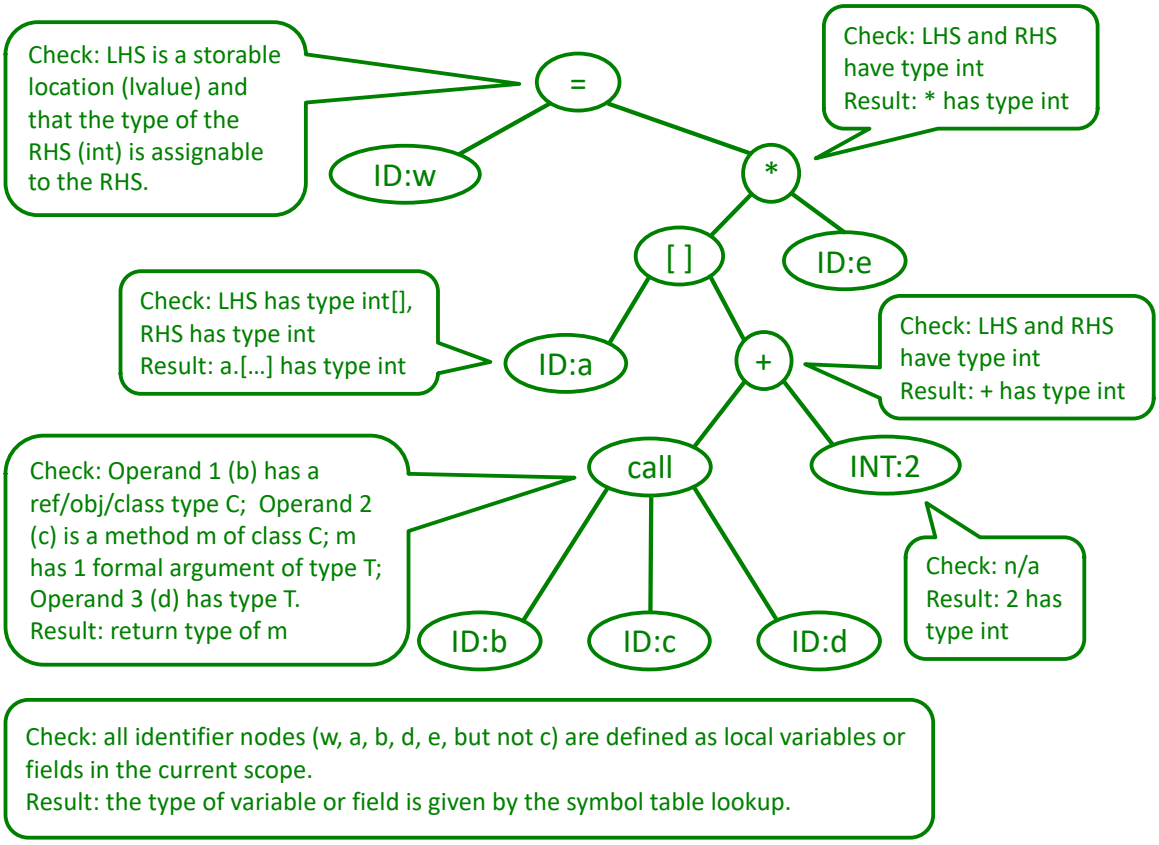
1. ***Z* :: = a *X T***
2. ***Z* :: = y s**
3. ***T* :: = *ε***
4. ***T* :: = e s**
5. ***X* :: = a *X* e**
6. ***X* :: = y**

Question 6. (14 points) Semantics. Suppose we have the following statement in a MiniJava program:

```
w = a[b.c(d) + 2] * e;
```

(a) (7 points) At the bottom of this page, draw an abstract syntax tree (AST) for this assignment statement. You should use appropriate names for the AST nodes, and have an appropriate level of abstraction and structural detail similar to the AST nodes in the MiniJava project AST classes, but don't worry about matching the exact names or details of classes or nodes found in the MiniJava starter code.

(b) (7 points) Annotate your AST by writing next to the appropriate nodes the checks or tests that should be done in the static semantics/type-checking phase of the compiler to ensure that this statement does not contain errors. If a particular check or test applies to multiple nodes, you can write it once and indicate which nodes it applies to, as long as your meaning is clear and readable. You may assume that `int` is the only numeric type in MiniJava, but remember that MiniJava also has `boolean` and object (class) types.



Question 7. (2 free points) (All reasonable answers receive the points. All answers are reasonable as long as there is an answer. ☺)

(a) (1 point) What question were you expecting to appear on this exam that wasn't included?

<this space intentionally left blank>

(b) (1 points) Should we include that question on the final exam? (circle or fill in)

Yes

No

Heck No!!

$!@$^*% No !!!!!

Yes, yes, it *must* be included!!!

No opinion / don't care

None of the above. My answer is _____.