CSE 401 - Midterm Review Sample Solutions - Week 6

1. Ambiguity

Here is one possible grammar for logical comparison operators in Java. Note that `||` is used as a single token representing logical or in Java -- it is distinct from the `|` that is used to separate productions.

```
A ::= A || B
A ::= B
B ::= A && A
B ::= id
```

a. Show that this grammar is ambiguous.

Two leftmost derivations of the string "id || id && id":

```
A => A || B => B || B => id || B => id || A && A => id || B && A => id || id && A
=> id || id && B => id || id && id
```

```
A => B => A && A => A || B && A => B || B && A => id || B && A => id || id && A
=> id || id && B => id || id && id
```

b. Give a grammar that generates the same language, but without ambiguity. For this problem, you may use any precedence and associativity that you would like as long as the resulting grammar is unambiguous and will generate the same set of strings as the original grammar.

```
A ::= A || B
A ::= B
B ::= B && id
B ::= id
```

2. Regular Expressions and DFAs

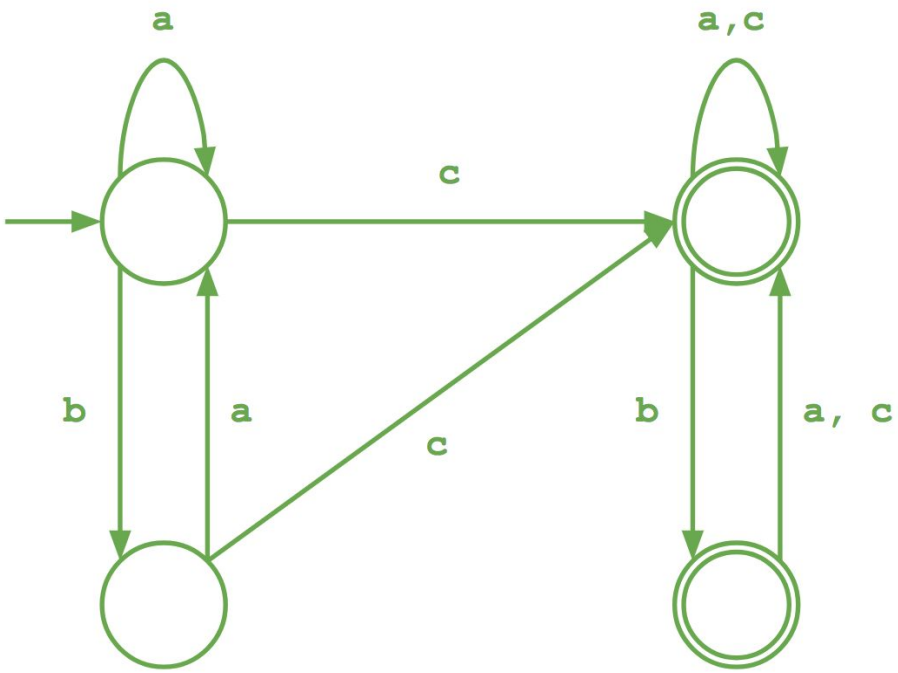
a. Write a regular expression that matches the following set of strings:

All strings of a's, b's, and c's in which there is at least one c, and in which there are no consecutive b's.

You may only use the basic operations of concatenation, choice (|), and repetition (*), plus the simple extensions ? and +, and character sets like [a-z] and [^a-z]. You may also give names to subexpressions like vowels = [aeiou].

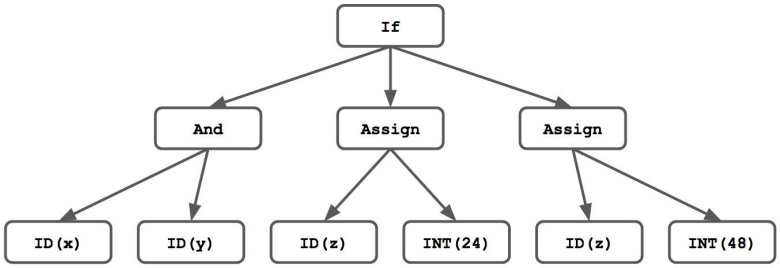
```
<sequence> = (a|c|b(a|c))*b?
<solution> = <sequence>c<sequence>
```

b. Draw a DFA that matches the same set of strings as your regular expression from part (a). It is not necessary to use the formal algorithm for converting from a regular expression -- you may find it easier (and the result more readable) to draw the DFA directly. For this problem, you do not have to worry about an explicit error state.



3. Abstract Syntax and Semantics

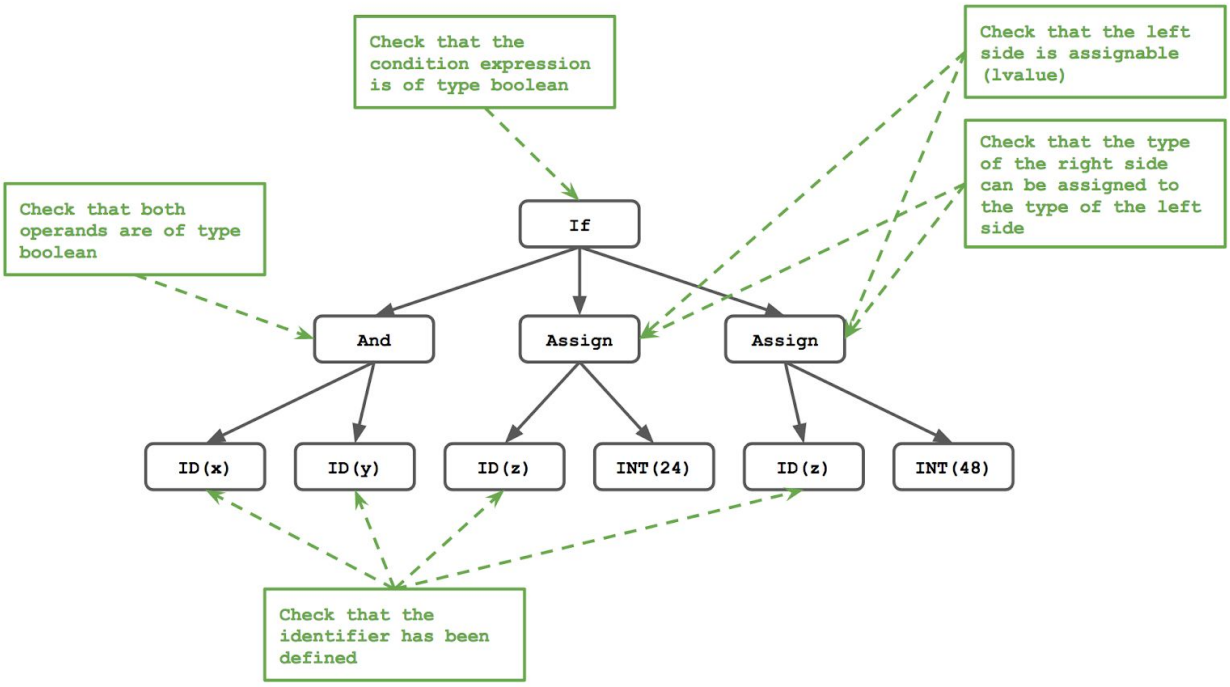
Consider the following Abstract Syntax Tree for a fragment of a MiniJava program:



a. Write the MiniJava source code that would produce this AST, including all necessary punctuation and syntax to make it legal Java code. In other words, write just the portion of the input that could be fed into the scanner and parser so that the resulting overall AST would contain this tree.

```
if (x && y)
    z = 24;
else
    z = 48;
```

a. What checks need to be performed in the static semantics/typechecking phase of the compiler to verify that this abstract syntax is, in fact, legal and contains no type errors or other static semantics problems? You may annotate the diagram itself or reproduce your answer in by listing the checks for each node.



4. Resolving LR(0) Conflicts with SLR

Consider the following grammar and its LR(0) parse table:

```
1. S'::= S $
2. S ::= A x S
3. S ::= A x
4. A ::= y
```

	x	y	$	S	A
1		s4		g2	g3
2			acc		
3	s5				
4	r4	r4	r4		
5	r3	s4,r3	r3	g6	g3
6	r2	r2	r2		

a. What kind of conflict is present in this LR(0) parse table?

A shift/reduce conflict in state 5.

b. Construct the SLR parse table for this grammar, computing nullable and the FIRST and FOLLOW sets for the grammar as necessary. Then describe why the SLR parse table resolves the conflict that is present in the LR(0) parse table. You may indicate the changes for SLR in the existing parse table above, or you may reproduce the new table entirely.

	FIRST	FOLLOW	nullable
S	y		no
A	y	x	no

Since y is not in the FOLLOW set for the non-terminal S, in the SLR parse table for this grammar, it will not be necessary to have r3 appear under the y column in state 5. Therefore, the shift/reduce conflict can be resolved in favor of shifting.

5. Components of a Compiler

The front end of a compiler consists of three parts: scanner, parser, and (static) semantics. Collectively these need to analyze the input program and decide if it is correctly formed. For each of the following MiniJava errors, indicate which stage of the front-end of the MiniJava compiler would normally handle it. If it helps to explain your answer you can give a brief reason why, but that is not required. For this question, you may use a copy of the MiniJava grammar; you are not expected to have it memorized.

a. Using the increment operator from Java, "++", after the name of a variable of type int.

Parser. (The scanner would recognize both plus tokens as being legal, and would not complain. It would be up to the parser to notice that two plus tokens cannot be in a row like that.)

b. Using the increment operator from Java, "++", after the name of a variable of type boolean.

Parser. (For the same reason as (a) -- although there is an additional error that the semantics checker would discover, the parser goes first and does not care about types.)

c. Assigning a value to the integer literal 5, as in 5 = 10;

Parser. (Only identifiers can be assigned to in MiniJava -- this is part of its grammar. However, if a covering grammar were used and any expression were allowed on the left side of an assignment statement, it may be the case that the semantics checker would be responsible for this one)

d. Writing "/*" in a program but never writing the corresponding "*/".

Scanner. (Only the scanner deals with comments; they should be stripped out by the time the data is handed to the parser.)

e. Declaring a variable named "x" when a variable with that name has already been declared.

Semantics.