Question 1. (14 points) Regular expressions. We would like to process strings that represent simple polynomials. A polynomial is the sum of a sequence of one or more terms, and each term consists of a positive coefficient, the variable x, and an exponent. There is a ^ character between each x and the corresponding exponent. The last term (only) in a polynomial can be a coefficient without the variable x and its exponent. Some examples:

2x^17+3x^2+42   1x^1+5x^3   2x^2+3x^1+1x^2+5   17

Simplifications and restrictions:

- Polynomials have at least one term (i.e., are not an empty string).
- The only variable in these polynomials is the single letter x.
- Coefficients and exponents are strings of decimal digits 0 through 9 that do not start with a 0, i.e., all coefficients and exponents are non-zero positive integers with no leading 0s.
- The last term (only) can be a coefficient (integer) by itself. All other terms must have an x and an exponent.
- Exponent values may appear in any order and may be repeated in different terms in the same polynomial (i.e., 2x^2+5x^3+1x^2 is legal).
- Coefficients may not be omitted (i.e., x^2 is not legal, 1x^2 is)
- If x appears, it must have an exponent.
- There are no negative coefficients or exponents, and no − operator.
- There is no whitespace (blanks, tabs, etc.) in a polynomial string.

Some strings that are not polynomials according to these rules:

x^2 (no coefficient); 3x (no exponent); 17+1x^2 (only last term can be an integer without a x^… following); 3x^0, 3x^01, 0, 03x^2, 3x^5+017 (leading 0s not allowed), 2^3 (no x).

As with homework problems, you must restrict yourself to the basic regular expression operations covered in class and on homework assignments: r s, 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 language libraries are allowed.

Write your ….

answers on ….

the next page.

Remove this page from the exam and do not include it when you hand in your exam. It will not be scanned or graded.

Question 1. Write your answers here. Hint: it may be useful to work on the regular expression and the DFA parts simultaneously.

(a) (7 points) Give a regular expression (or collection of regular expressions) that generates all valid polynomials according to the above rules.

<int> = [1-9][0-9]*

<term> = <int>x^<int>

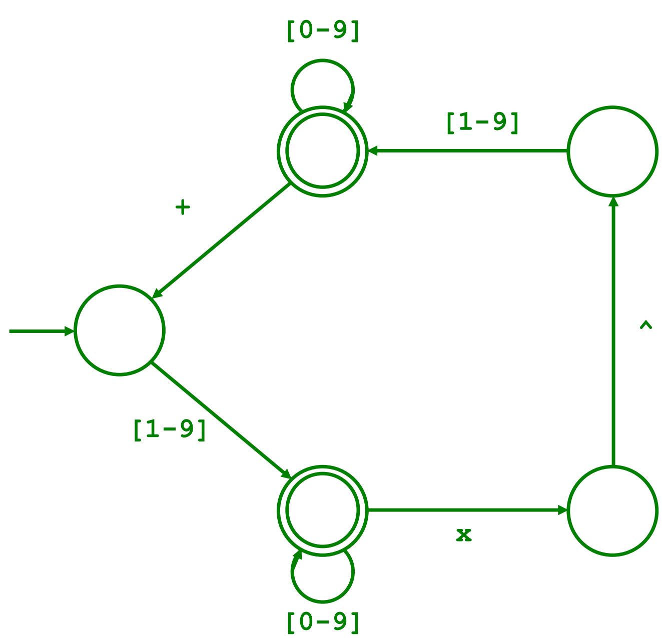
<polynomial> = <int> | <term> (+ <term>)* (+ <int>)?

Another way to write the last line that also works is:

<polynomial> = (<term> +)* <int> | (<term> +)* <term>

Of course, any other correct solution also received full credit.

(b) (7 points) Draw a DFA that accepts all valid polynomials according to the above rules.



Question 2. (10 points) Scanners and tokens. To see what would happen, we ran our MiniJava scanner using a file containing the following Ruby code fragment as input:

```
if a <= 1 do
     data = {:while} // to return
end
```

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) (something brief like "illegal character #" if a "#" was found in the file would be fine). The token list should include additional tokens found after any error(s) in the input. 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 may remove it for reference while you answer this question. You should assume the scanner implements MiniJava syntax as defined in that grammar, with no extensions to the language.

IF ID(a) LESS EQUALS INT(1) ID(do)

ID(data) EQUALS LBRACE

Invalid character ":"

WHILE RBRACE ID(end)

Other token names received full credit as long as it was clear what was intended. Several solutions had a deduction, though, because it appeared that the invalid character was treated as an "error token" to be returned to the parser (i.e., tokens like ERROR(";")).

Question 3. (12 points) Ambiguity. Consider the following grammar:

A ::= `x` `=` *B* **`;`**
B ::= *B* + *B*
B ::= y

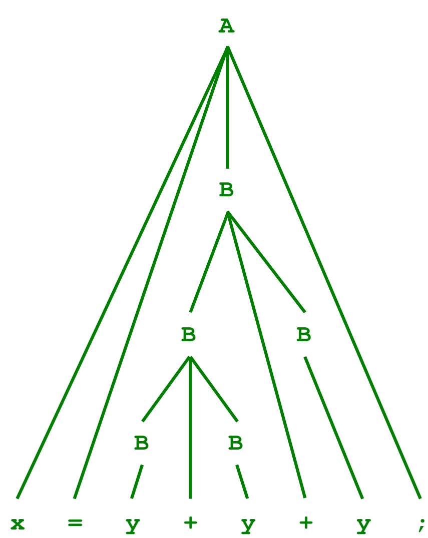
(*A* and *B* are non-terminals, `x`, `y`, +, =, and **`;`** are terminals)

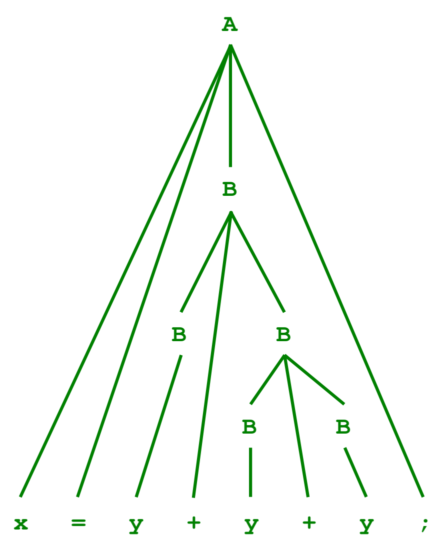
(a) (6 points) Is this grammar ambiguous? If so, give a proof that it is by showing two distinct parse trees, or two distinct leftmost (or rightmost) derivations, for some string. If not, give an informal, but precise argument why it is not ambiguous.

Here are two solutions. Two leftmost derivations of the string x = y + y + y:

A => x = B; => x = B+B; => x = y+B; => x = y+B+B; => x = y+y+B; => x = y+y+y;
A => x = B; => x = B+B; => x = B+B+B; => x = y+B+B; => x = y+y+B; => x=y+y+y;

Two parse trees for the string x = y + y + y:





(b) (6 points) If your answer to part (a) is that the grammar is ambiguous, give an unambiguous grammar that generates the same language as the original grammar. If there are several possible solutions, give one where precedence and associativity are handled the same as in Java (i.e., + is left-associative, etc.) if that is possible.

If the grammar in part (a) is unambiguous, you may leave this part of the problem blank to receive full credit for it.

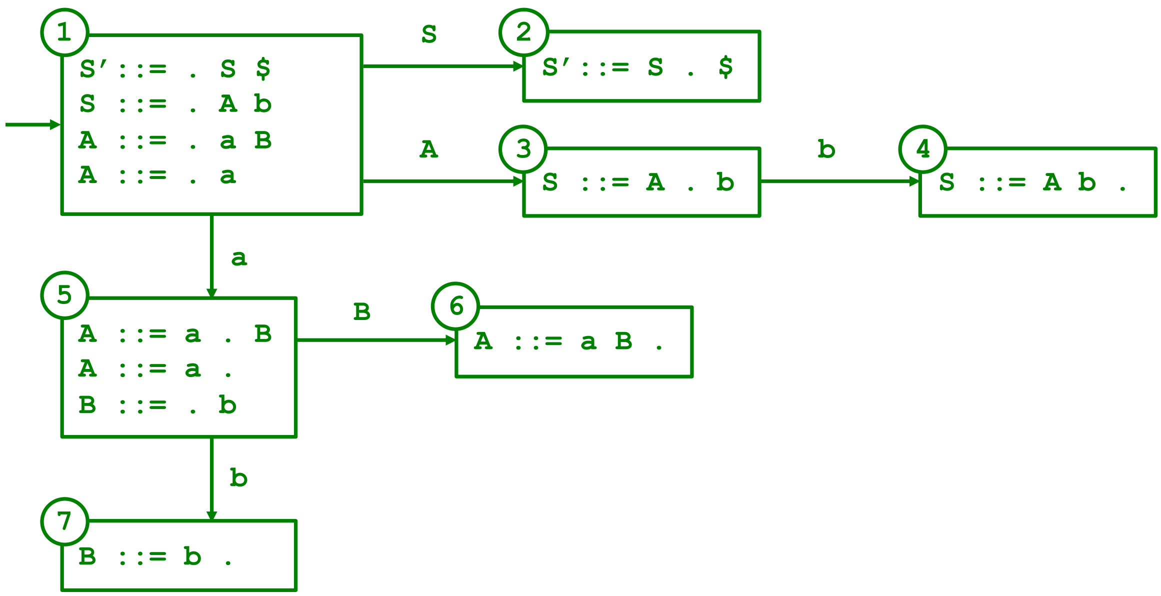
A ::= x = B ;
B ::= B + y
B ::= y

Notes: This solution gives + higher precedence than assignment (=) (as was true in the original grammar), and + is now left associative.

Question 4. (34 points) The LR parsing question that always has a funny(?) slogan. Here is a tiny grammar.

1. S' ::= S `$` (`$` represents end-of-file)
2. S ::= A `b`
3. A ::= `a` B
4. A ::= `a`
5. B ::= `b`

(a) (12 points) Draw the LR(0) state machine for this grammar.



(b) (8 points) Compute *nullable* and the FIRST and FOLLOW sets for the nonterminals S, A, and B in the above grammar:

Symbol	nullable	FIRST	FOLLOW
S	**false**	**a**	**$**
A	**false**	**a**	**b**
B	**false**	**b**	**b**

(continued on next page)

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

1. $S' ::= S$ `$` (`$` represents end-of-file)
2. $S ::= A$ `b`
3. $A ::=$ `a` B
4. $A ::=$ `a`
5. $B ::=$ `b`

(c) (10 points) Write the LR(0) parse table for this grammar based on the LR(0) state machine in your answer to part (a).

	a	b	$	S	A	B
1	s5			g2	g3	
2			acc			
3		s4				
4	r2	r2	r2			
5	r4	s7, r4	r4			g6
6	r3	r3	r3			
7	r5	r5	r5			

(d) (2 points) Is this grammar LR(0)? Explain why or why not.

No. State 5 has a shift-reduce conflict.

(e) (2 points) Is this grammar SLR? Explain why or why not.

No. The problem with state 5 is that if the next input symbol is `b` the state contains both a shift to state 7 and a reduction using the rule A ::= `a`. In the SLR table we should remove the reduction if `b` is not in FOLLOW(A), but since `b` is in that set the reduction remains and the shift-reduce conflict is still present.

Question 5. (15 points, 5 each) LL grammars. For each of the following grammars indicate if it satisfies the LL(1) condition, i.e., it is possible to construct a predictive parser using the grammar. If the grammar is not LL(1) explain why not. (Hint: you may find it helpful to determine FIRST, FOLLOW, and nullable for some or all of the non-terminals. But the answers can probably be figured out without having to go through all the details of the algorithms to compute those sets, and you do not need to do that.)

(a) *P* ::= `a b` *Q* `c` *R*
Q ::= `c` | *R* `b` | ε
R ::= `a`

No, this grammar is not LL(1). Q is nullable so we need to also look at FOLLOW(Q) to decide which production to pick when we expand Q. FOLLOW(Q) contains `c`, as does FIRST(Q). Since those sets contain a common element we cannot pick the proper Q production when `c` is the next symbol in the input (there is no way to decide whether to pick Q::=`c` or Q::=ε).

(b) *P* ::= `a b` *Q* `c` *R*
Q ::= `c` | *R* `b`
R ::= `a`

Yes. The FIRST sets for the productions of each non-terminal are disjoint and none of the non-terminals are nullable. (Reason not required – we forgot to ask for it in the question, but probably should have.)

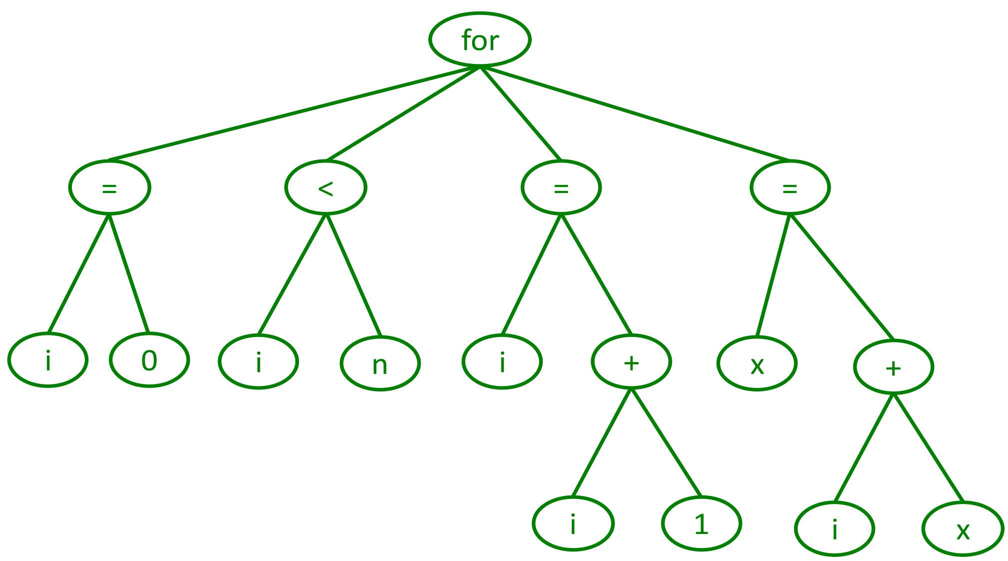
(c) *P* ::= `a b` *Q* `c` *R*
Q ::= `c` | *R* `b`
R ::= `a` | `c`

No. Since `c` appears in FIRST(R), then `c` is in the FIRST set for both Q productions and we cannot pick the correct production to expand Q if `c` is the next input symbol.

Question 6. (15 points) Semantics. Bowing to popular demand, we've decided to add a `for` loop to MiniJava. The syntax is `for(`*init*`;` *test*`;` *update*`)` *Statement*. The *init* and *update* parts are arbitrary *Statement*s; the *test* part is an expression that must evaluate to `true` or `false`. As in C or Java, the *init*, *test*, and *update* parts of the `for` statement do not have to be related to each other, e.g., `for(i=0; x<y; n=17) b=false;` is legal (although it is terrible style).

(a) (7 points) Given the statement `for(i=0; i<n; i=i+1) x=i+x;`, draw an appropriate Abstract Syntax Tree (AST) for that statement below. Don't worry about matching the exact structure of the MiniJava AST classes – just be sure your drawing shows a reasonable AST for this statement.

(b) (8 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 any errors. You do not need to specify an attribute grammar – just show the necessary tests. If a particular 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.



(b) There are only a handful of node types in the AST and the same semantic checks are needed for each occurrence of the same type of node. Here's the list:

`for`: verify that the 2nd child has type Boolean.
Identifier nodes: verify identifier is declared and has type int. Type of node is int.
Integer constant nodes: type of node is int.
+ (addition): verify both operands have type int. Type of + node is int.
< (comparison): verify both operands have type int. Type of < node is Boolean.
= (assignment): verify both operands have type int. Verify that the left operand designates a location (lvalue).

Note: the `for` node has 4 children. Three are statements and one is an expression. The only check needed in that node is to verify that the expression is Boolean.