Question 1. (18 points) Regular expressions and DFAs. In the US, currency amounts are normally written with a leading $, one or more digits, optionally with one or more leading 0's, giving the number of dollars, and, optionally following that, a decimal point with two digits giving the number of cents. If there are more than 3 digits in the dollar part, they are separated into groups of three digits with commas, counting from the right.

Examples of legal strings using these rules: $1, $01, $1,234.56, $17.42, $1.23, $01.23, $00,017.42, $12, $12.50, $1,024, $8,820,000,000 .

Examples of illegal strings: 1.23 (no leading $), $12.5 (only one digit after decimal point), $12,34.56 (must be a group of three digits after each ","), $.17 (no digit before the "."), $1234.56 (leading digits not separated into groups of 3 with commas), $1.235 (more than 2 digits after ".").

(a) (9 points) Give a regular expression that generates all strings that are legal currency amounts according to these rules.

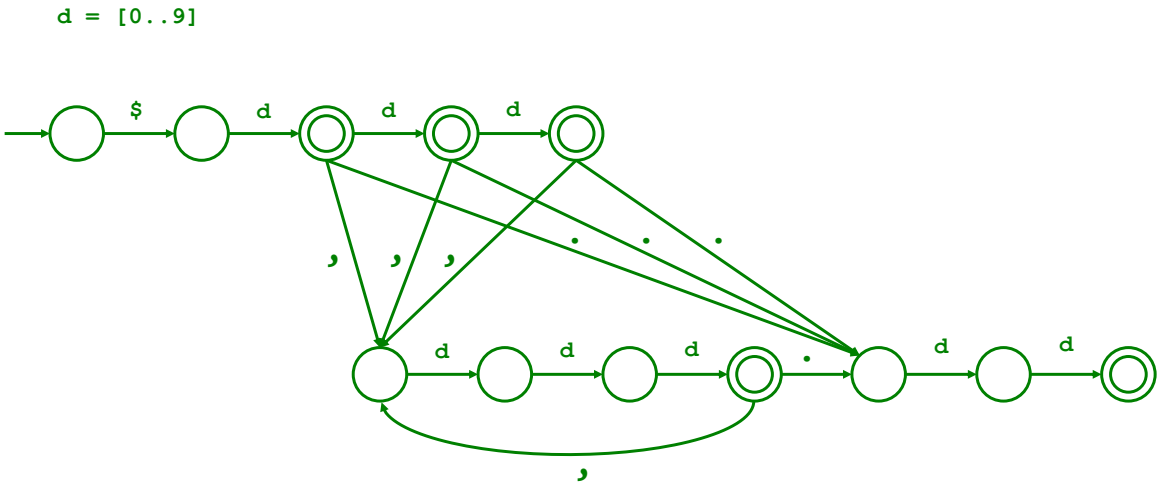
Fine print: You must restrict yourself to the basic regular expression operations covered in class and on homework assignments: *rs* , *r*`|`*s* , *r*`*` , *r*`+` , *r*`?`, character classes like `[a-cxy]` and `[^aeiou]`, abbreviations *name=regexp*, and parenthesized regular expressions. No additional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

d = [0-9]

$ d d? d? (, d d d)* (. d d)?

(continued on next page)

Question 1 (cont.) (b) (9 points) Draw a DFA that accepts all valid currency strings described above and generated by the regular expression from part (a).



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 (which was taken from the Haskell programming language website)?

```
[x | x <- xs, x `mod` p /= 0]
```

Note that the punctuation marks surrounding `mod` above are backquotes (reverse quotes).

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 @" 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.

LBRACK ID(x)

Illegal character '|'

ID(x) LESS MINUS ID(xs) COMMA ID(x)

Illegal character '`'

ID(mod)

Illegal character '`'

ID (p)

Illegal character '/'

EQUALS INT(0) RBRACK

Question 3. (14 points) Ambiguity. The classic Unix/Linux typesetting system includes a program named *eqn* to format equations. The input to *eqn* describing equations with subscripts and superscripts is given by the following grammar:

E ::= *E* `sub` *E*
E ::= *E* `sup` *E*
E ::= `x`

The `sup` operator designates superscripts while `sub` designates subscripts. So, `x sub x` is typeset as x_x, while `x sup x` is x^x.

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

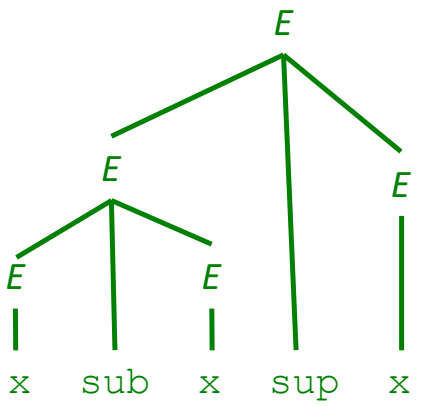
Notes: whitespace in the grammar is only for readability and is not part of the grammar or the strings generated by it. Each of the words like `sub` or `sup` in a grammar production should be treated as a single terminal symbol, not as individual letters.

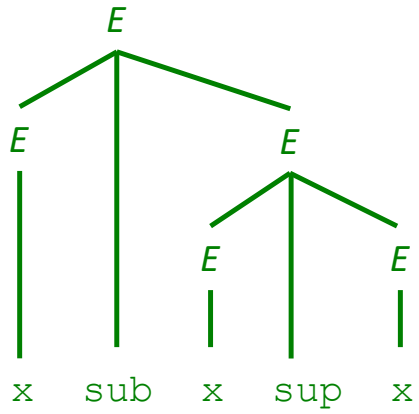
Yes. Two distinct leftmost derivations of x sub x sup x:

E => E sup E => E sub E sup E => x sub E sup E => x sub x sup E => x sub x sup x

E => E sub E => x sub E => x sub E sup E => x sub x sup E => x sub x sup x

It also is possible to show the ambiguity by drawing the respective parse trees.



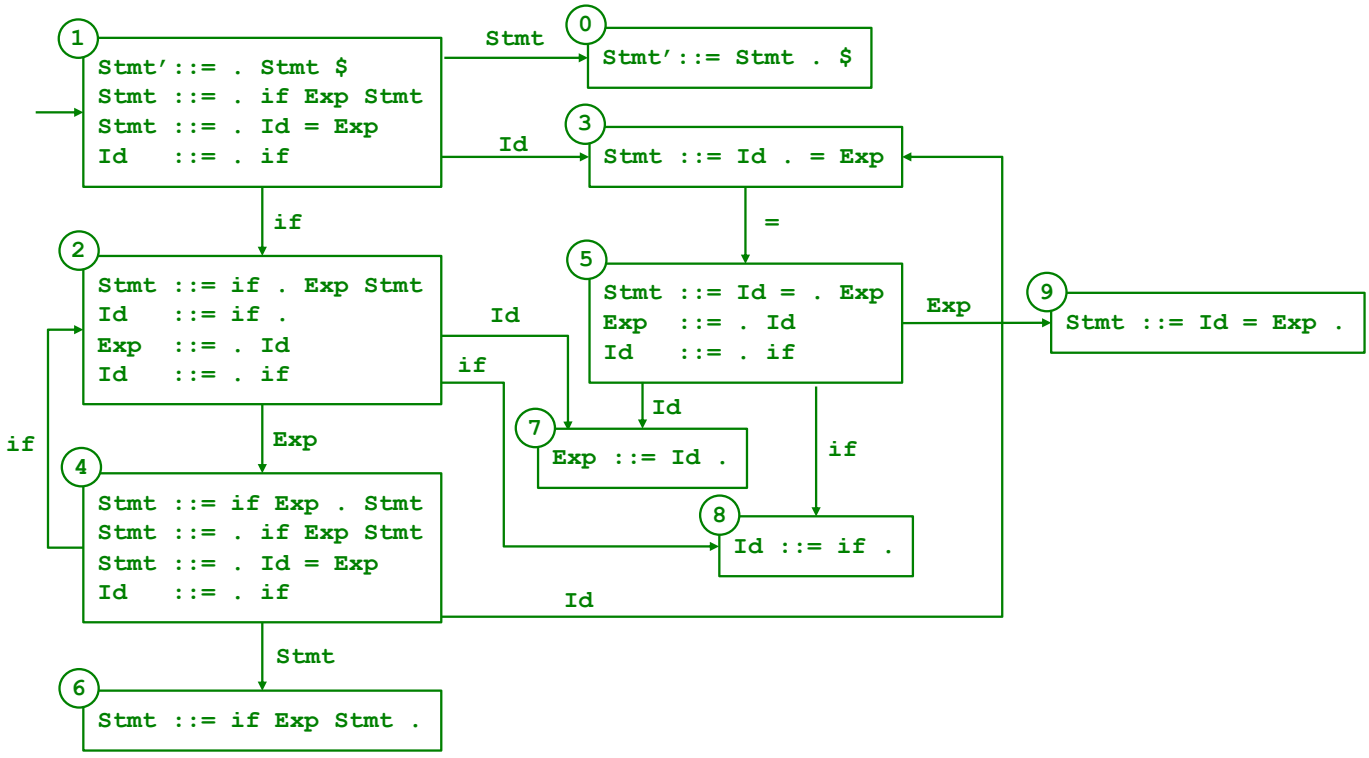


Question 4. (40 points) The "OMG! Not this again!!" parsing question (but with a surprise twist at the end!). One of the summer interns has decided that programming languages would be better if they did not contain reserved words that cannot be used as identifiers. After all, why shouldn't someone be able to have variables named `while` or `for` in a program?

To see what the implications of this might be, the following is a grammar for a very small language with a conditional and assignment statement, and expressions that consist of identifiers. The terminal symbol `if` can be used as an identifier, and it also appears at the beginning of a conditional statement. The extra *Stmt'* ::= *Stmt* `$` rule needed to handle end-of-file in an LR parser has been added for you. As is usual, whitespace in the grammar is only for readability and is not part of the grammar or the strings generated by it. Terminal symbols like `if` are single symbols, not strings of letters.

0. *Stmt'* ::= *Stmt* `$` (`$` is EOF)
1. *Stmt* ::= `if` *Exp Stmt*
2. *Stmt* ::= *Id* = *Exp*
3. *Exp* ::= *Id*
4. *Id* ::= `if`

(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.



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

0. *Stmt'* ::= *Stmt* `$` (`$` is EOF)
1. *Stmt* ::= `if` *Exp Stmt*
2. *Stmt* ::= *Id* = *Exp*
3. *Exp* ::= *Id*
4. *Id* ::= `if`

(b) (8 points) Write the LR(0) parser tables for the LR parser in your answer to part (a).

State	`if`	=	$	*Stmt*	*Exp*	*Id*
0			acc			
1	s2			g0		g3
2	s8, r4	r4	r4		g4	g7
3		s5				
4	s2			g6		g3
5	s8				g9	g7
6	r1	r1	r1			
7	r3	r3	r3			
8	r4	r4	r4			
9	r2	r2	r2			

(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 `foo`", but with, of course, state numbers and correct details from your parser). If the grammar is LR(0), you should explain why (this can be brief).

No. There is a shift-reduce conflict in state 2 on input `if`.

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

0. *Stmt'* ::= *Stmt* `$` (`$` is EOF)
1. *Stmt* ::= `if` *Exp Stmt*
2. *Stmt* ::= *Id* = *Exp*
3. *Exp* ::= *Id*
4. *Id* ::= `if`

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

	FIRST	FOLLOW	nullable
Stmt	**if**	**$**	**no**
Exp	**if**	**if $**	**no**
Id	**if**	**if = $**	**no**

(e) (3 points) Is this grammar SLR? Explain why or why not.

No. `if` is in FOLLOW(*Id*), so we can't remove the reduce conflict in state 2 if the next input is `if`.

(f) (4 points) (The surprise!) Is the grammar given above, but omitting the *Stmt'* ::= *Stmt* $ rule that was added for LR parsing, suitable for constructing a top-down LL(1) predictive parser? If it is, your answer should give a technical explanation why it is. If not, your answer should give a technical explanation describing the problem or problems with this particular grammar that prevent it from being suitable for a LL(1) predictive parser.

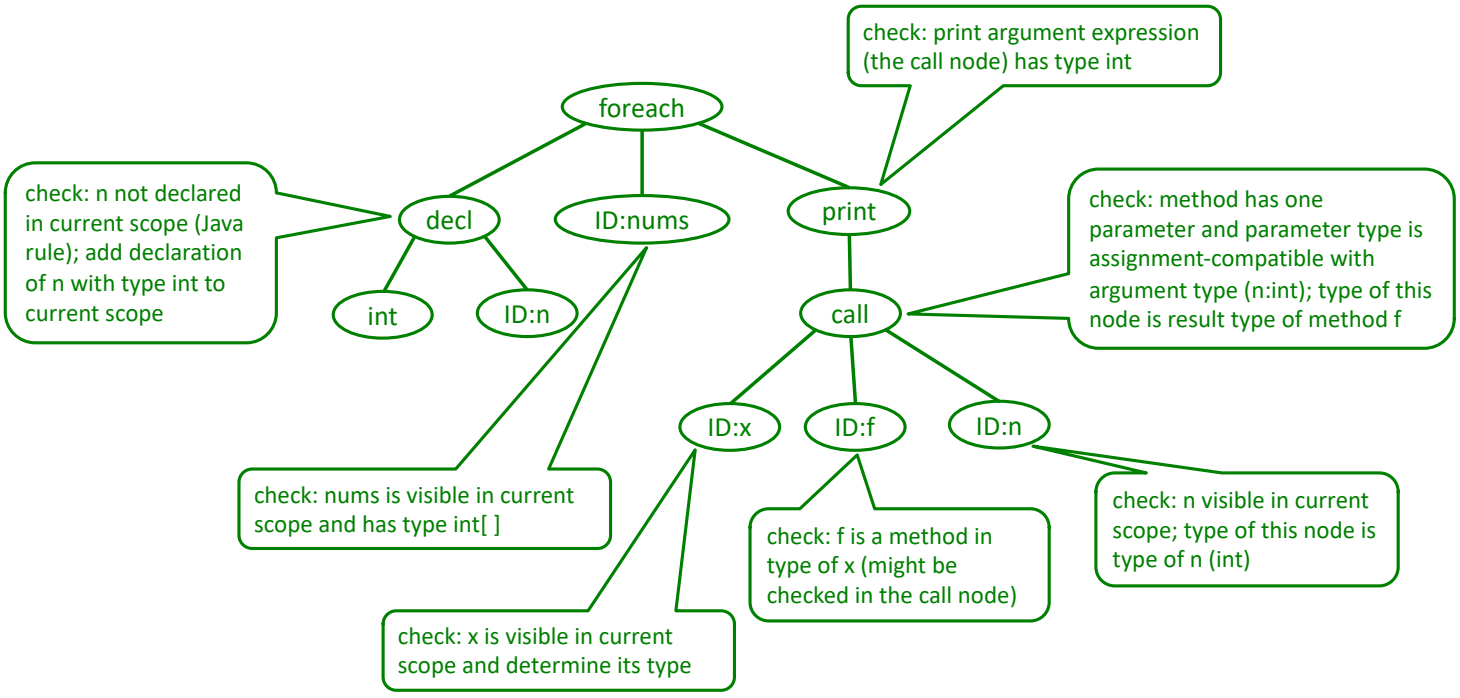
No. The two productions for *Stmt* both have `if` in the FIRST set of the right-hand side.

Question 5. (16 points) Semantics. We would like to add a *for-each* loop to MiniJava to provide a more compact way to process MiniJava arrays. Here is a program fragment that uses this new loop to print the results of calling method `x.f(n)` for each of the integer elements `n` of the array `nums` (recall that the `int n` at the beginning of the loop is the declaration of `n` as a variable local to the loop, and since MiniJava only has arrays with `int` values, this is the only possible type for the local variable in this loop):

```
for (int n: nums) System.out.println(x.f(n));
```

(a) (8 points) Draw an abstract syntax tree (AST) for this statement and its children at the bottom of this page. 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 code.

(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 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.



Note: There are many possible ways to draw the tree and as long as the answer captured the essential ideas it received credit. The above semantic checks are all necessary, but, as with the tree nodes, as long as they were indicated in appropriate places in the actual tree (or were listed below) they received credit.

Question 6. (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?

(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 _____.