More Assigned Reading and Exercises on Syntax (for Exam 2)

1. Read sections 2.3 (Lexical Syntax) and 2.4 (Context-Free Grammars) on pp. 33 – 41 of Sethi.

2. Read section 2.6 (Variants of Grammars) on pp. 46 – 48 of Sethi.

3. Read section 2.5 (Grammars for Expressions) on pp. 41 – 45 of Sethi.

Exercises C (Grammars, Parse Trees): Do problems **2.4**, **2.6***, and **2.14**[†] on pp. 49 and 51. You should be able to do these after reading pp. 33 – 38.

*Problem 2.6 can be done after you have read pp. 33 – 38, but might be easier after you have read Sec. 2.5. It is therefore included both in Exercises C and in Exercises E.

[†]In problem 2.14 there is a misprint: The first production with `SL` on the left should read "`SL ::= S`" rather than "`SL ::= S ;`".

Exercises D (EBNF): Do problems **2.9** and **2.16a**** on pp. 50 – 52. You should be able to do these after reading the **Extended BNF** subsection of Sec. 2.6.

**Re 2.16: The first line of the EBNF specification on p. 52 uses [ ] to mean an empty string.

Exercises E (Derivations, Grammars for Expressions, Syntactic Ambiguity, Syntax Charts): Do problems **2.5**[¶], **2.6**, **2.10**[‡], **2.11**, **2.12**[§], **2.13**[§], **2.15a**, **2.15b**, and **2.16b** on pp. 49 – 52.

[¶]Re 2.5(a): A ***leftmost*** derivation is a derivation in which the leftmost nonterminal is replaced at each step. A ***rightmost*** derivation is a derivation in which the rightmost nonterminal is replaced at each step. The derivation of 2 1 . 8 9 from *real-number* that is shown on p. 41 is a leftmost derivation. For example, its second step replaces the *leftmost* nonterminal (*integer-part*) in the string *integer-part* . *fraction* with *integer-part digit*, and then its third step replaces the *leftmost* nonterminal in the string *integer-part digit* . *fraction* with *digit*. The second step of a rightmost derivation of 2 1 . 8 9 from *real-number* would replace the rightmost nonterminal (*fraction*) in the string *integer-part* . *fraction* with *digit fraction*.

[‡]Hint for 2.10(b): Put the unary – operator in the same precedence class as binary + and binary –.

[§]Even though problem 2.12 does not say this, the grammar you write for that problem must be able to generate expressions that involve ! (the logical negation operator) as well as the binary operators shown in Figure 2.9; otherwise you will not be able to do parts b and c of problem 2.13. ! is a right-associative unary prefix operator and has higher precedence than any binary operator.

Solutions to Assignments on Syntax (Sections C, D, and E)

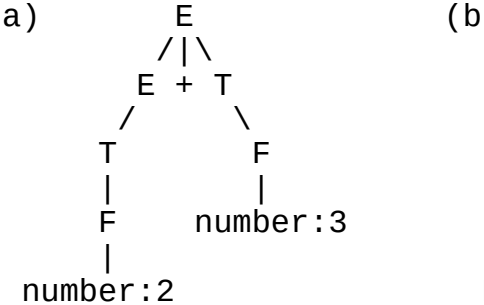
Section C

2.4 (a)
```
<str> ::= <item> | <str> <item>
<item> ::= blank | tab | newline
```

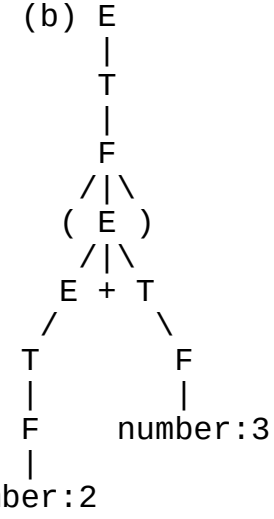
(b)
```
<seq> ::=   <letter>
          | <seq> <letter>
          | <seq> <digit>
<letter> ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
<digit> ::= 0|1|2|3|4|5|6|7|8|9
```

(c)
```
<real num> ::= <int part> . <frac> | <int part> . | . <frac>
<int part> ::= <digit> | <int part> <digit>
<frac>  ::= <digit> | <digit> <frac>
<digit> ::= 0|1|2|3|4|5|6|7|8|9
```

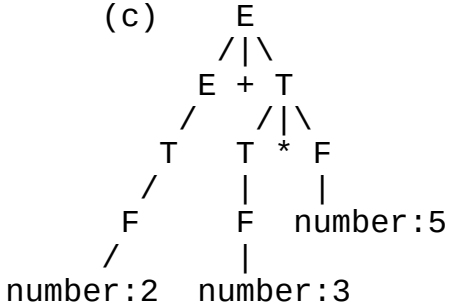
2.6 (a)



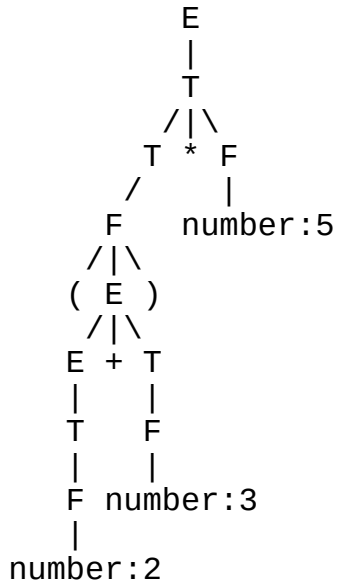
(b)



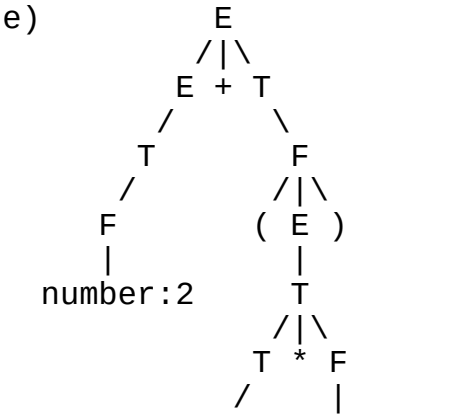
(c)



(d)



(e)



2.14 The productions for SL should read SL ::= S | S ; SL (Pascal statements need not end with semicolons; it would in fact be impossible to derive b from the given productions.)

We will use a "sideways" representation of parse trees. This is obtained from a preorder traversal of the tree by writing out each node when it is visited, on a new line with an indentation that is proportional to the depth of the node in the parse tree. [TinyJ Assignment 1 uses the same representation of parse trees, except that it gives each internal node of the tree an additional rightmost child that is a leaf labeled "... node has no more children".] In this sideways representation of trees, the parent of each non-root node is the node on the closest previous line that has lower indentation.

It would be a GOOD (and EASY) EXERCISE for you to draw each of the following three parse trees using the ordinary representation of trees instead of the sideways representation that is used here:

```
(a)    S
         while
         expr
         do
         S
           id
           :=
           expr

(b)    S
         begin
         SL
           S
             id
             :=
             expr
         end

(c)   S
        if
        expr
        then
        S
          if
          expr
          then
          S
          else
          S
```

Section D

2.9 (a) One or more <name>s, separated by commas.
    (b) One or more occurrences of "<field>;" (i.e., one or
        more <field>s, with each <field> followed by a semicolon).
    (c) Zero or more occurrences of "<statement>;" (i.e., zero
        or more <statement>s, each followed by a semicolon).
    (d) One or more <factor>s, separated by *s.
    (e) Either the empty string, or "var" followed by one or more
        occurrences of "<name-list> : <type>;" (thus each
        occurrence of "<name-list> : <type>" is followed by
        a semicolon).
    (f) Either the empty string, or "const" followed by one or
        more occurrences of "<name> = <constant>;" (thus each
        occurrence of "<name> = <constant>" is followed by
        a semicolon).

2.16 (a) S ::=   id := expr
                | if expr then SL elsif-part else-part end
                | loop SL end
                | while expr do SL end
                | <empty>

          SL ::=   SL ; S
                 | S

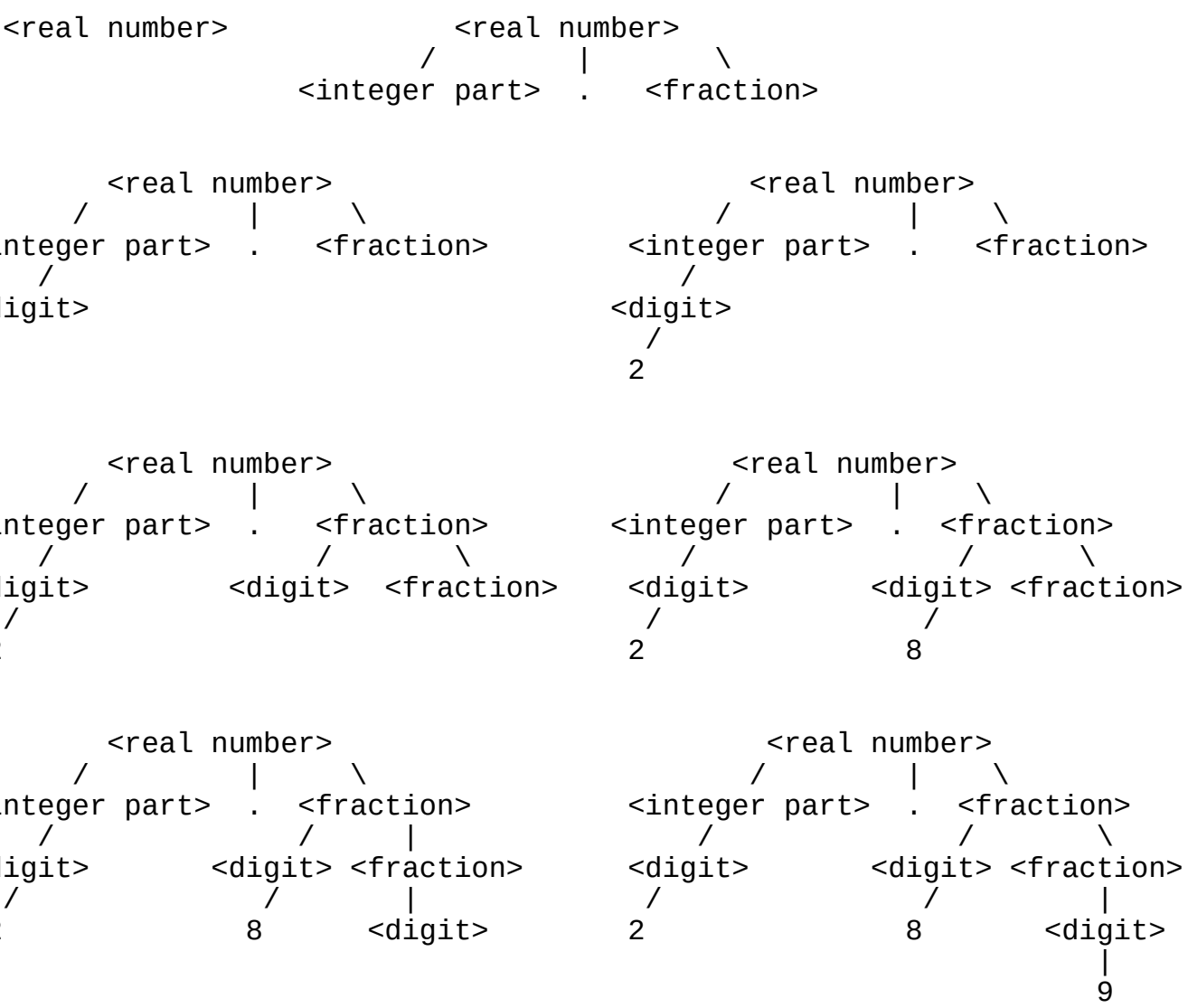
          elsif-part ::=   elsif-part elsif expr then SL
                         | <empty>

          else-part ::=   else SL
                        | <empty>

Section E

2.5 (a) <real number> => <integer part> . <fraction>
                      => <digit> . <fraction>
                      => 2 . <fraction>
                      => 2 . <digit> <fraction>
                      => 2 . 8 <fraction>
                      => 2 . 8 <digit>
                      => 2 . 8 9

(b)



2.6 SEE THE SECTION C SOLUTIONS!

2.10 (a)
```
<class 1 prefix op> ::= not
<class 2 binary op> ::= * | / | div | mod | and
<class 3 binary op> ::= + | - | or
<class 4 binary op> ::= < | > | <= | >= | = | <> | in

<exp0> ::= <constant> | <variable> | ( <exp4> )
<exp1> ::= <exp0> |        <class 1 prefix op> <exp0>
<exp2> ::= <exp1> | <exp2> <class 2 binary op> <exp1>
<exp3> ::= <exp2> | <exp3> <class 3 binary op> <exp2>
<exp4> ::= <exp3> | <exp4> <class 4 binary op> <exp3>
```

<exp4> is the starting nonterminal.

(b) Put unary - into class 3, the same precedence class as binary + and -. Thus we add the production

```
<class 3 prefix op> ::= -
```

and change the "<exp3> ::=" productions to:

```
<exp3> ::=   <exp2>
           | <exp3> <class 3 binary op> <exp2>
           |        <class 3 prefix op> <exp2>
```

Note: The solution to (a) is based on the problem as it is stated, rather than on the syntax of Standard Pascal. In Standard Pascal, at most one relational operator may appear at the top level of an expression. However, the problem does not mention this restriction. In Standard Pascal, expressions of the form "not not e" are legal. However, the problem says that NOT is left-associative, which might be interpreted as implying that such expressions are illegal--see, e.g., the last paragraph of: https://stackoverflow.com/a/14085309
If at most one relational operator is to be allowed at the top level, then the productions for <exp4> should be changed to:

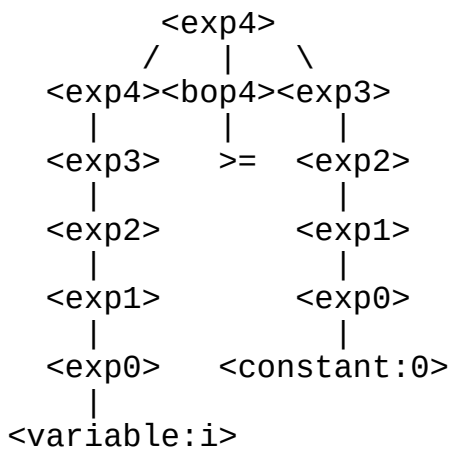
```
<exp4> ::= <exp3> | <exp3> <class 4 binary op> <exp3>
```

To allow expressions of the form "not not e", the productions for <exp1> should be changed to:

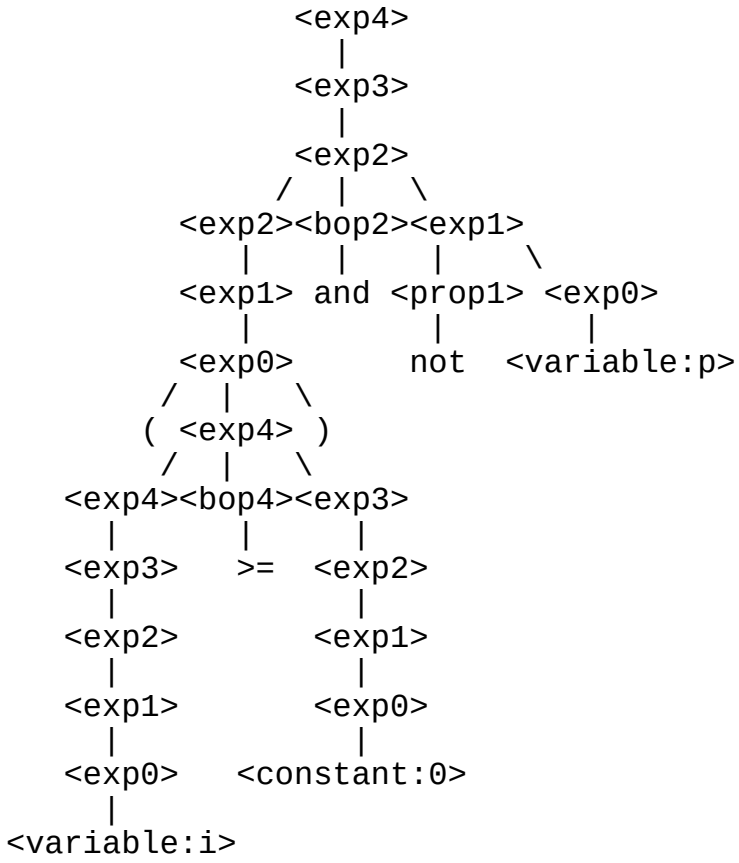
```
<exp1> ::= <exp0> | <class 1 prefix op> <exp1>
```

2.11 We write <bop4>, <bop2> and <prop1> for <class 4 binary op>, <class 2 binary op> and <class 1 prefix op>, respectively.

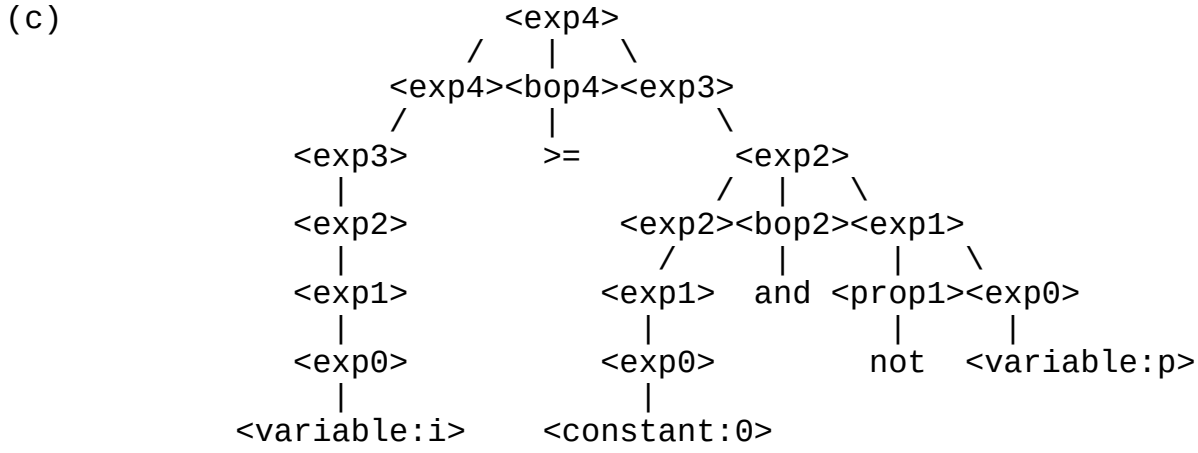
(a)



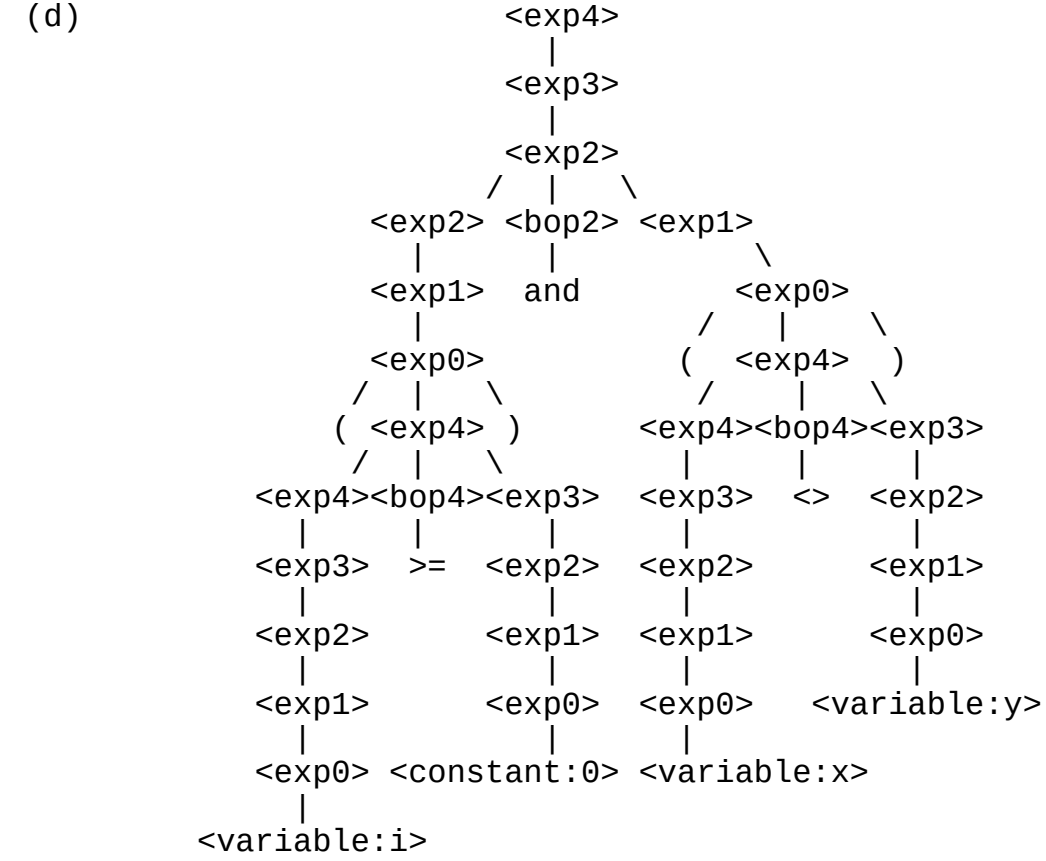
(b)



(c)


<exp4>
<exp4><bop4><exp3>
<exp3> >= <exp2>
<exp2> <exp2><bop2><exp1>
<exp1> <exp1> and <prop1><exp0>
<exp0> <exp0> not <variable:p>
<variable:i> <constant:0>

(d)


<exp4>
<exp3>
<exp2>
<exp2> <bop2> <exp1>
<exp1> and <exp0>
<exp0> (<exp4>)
(<exp4>) <exp4><bop4><exp3>
<exp4><bop4><exp3> <exp3> <> <exp2>
<exp3> >= <exp2> <exp2> <exp1>
<exp2> <exp1> <exp1> <exp0>
<exp1> <exp0> <exp0> <variable:y>
<exp0> <constant:0> <variable:x>
<variable:i>

2.12 It is clear from problem 2.13 that the ! operator should be included in the table; in C/C++ ! has higher precedence than all other operators in the table, and it is right-associative.

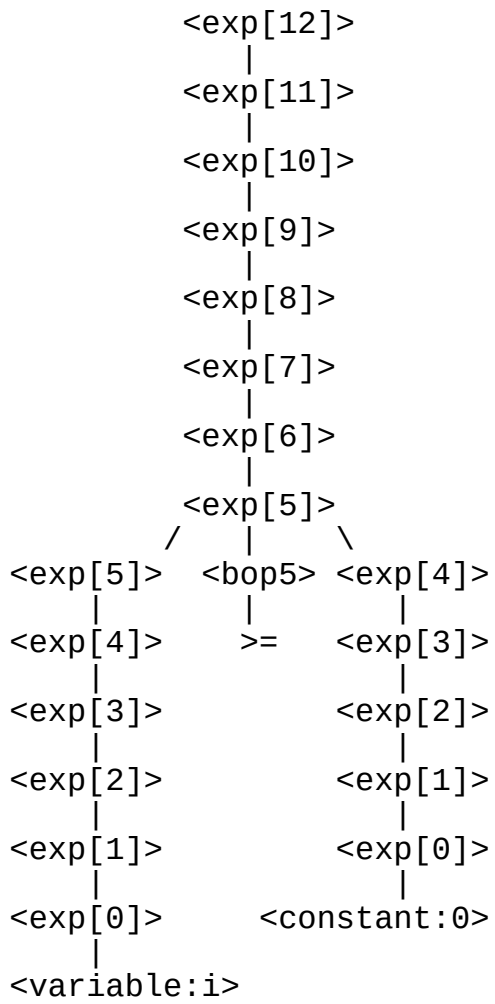
```
<class 1 prefix op> ::= !
<class 2 binary op> ::= * | / | %
<class 3 binary op> ::= + | -
<class 4 binary op> ::= << | >>
<class 5 binary op> ::= < | > | <= | >=
<class 6 binary op> ::= == | !=
<class 7 binary op> ::= &
<class 8 binary op> ::= ^
<class 9 binary op> ::= |
<class 10 binary op> ::= &&
<class 11 binary op> ::= ||
<class 12 binary op> ::= =

<exp[0]> ::= <variable> | <constant> | ( <exp[12]> )
<exp[1]> ::= <exp[0]> | <class 1 prefix op> <exp[1]>
For 2 <= i <= 11,
  <exp[i]> ::= <exp[i-1]> | <exp[i]> <class i binary op> <exp[i-1]>
<exp[12]> ::= <exp[11]> | <exp[11]> <class 12 binary op> <exp[12]>

<exp[12]> is the starting nonterminal.
```

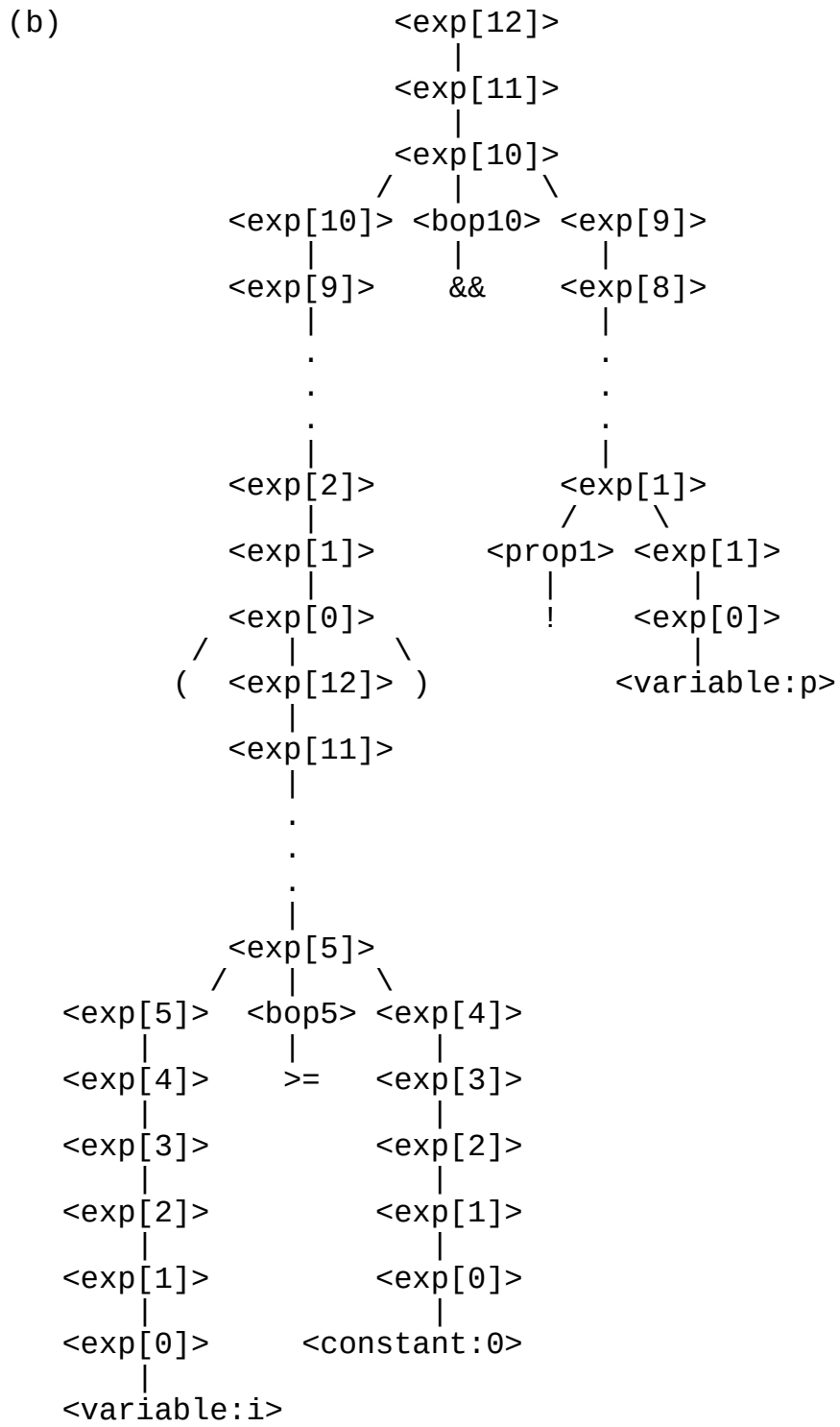
The question does not say this, but in fact the left operand of "=" in C/C++ must be associated with a location. If we are only allowed to use the given operators, then the left side has to be a variable. If we take this into account then the productions for <exp[12]> should be:

```
<exp[12]> ::= <exp[11]> | <variable> = <exp[12]>
```

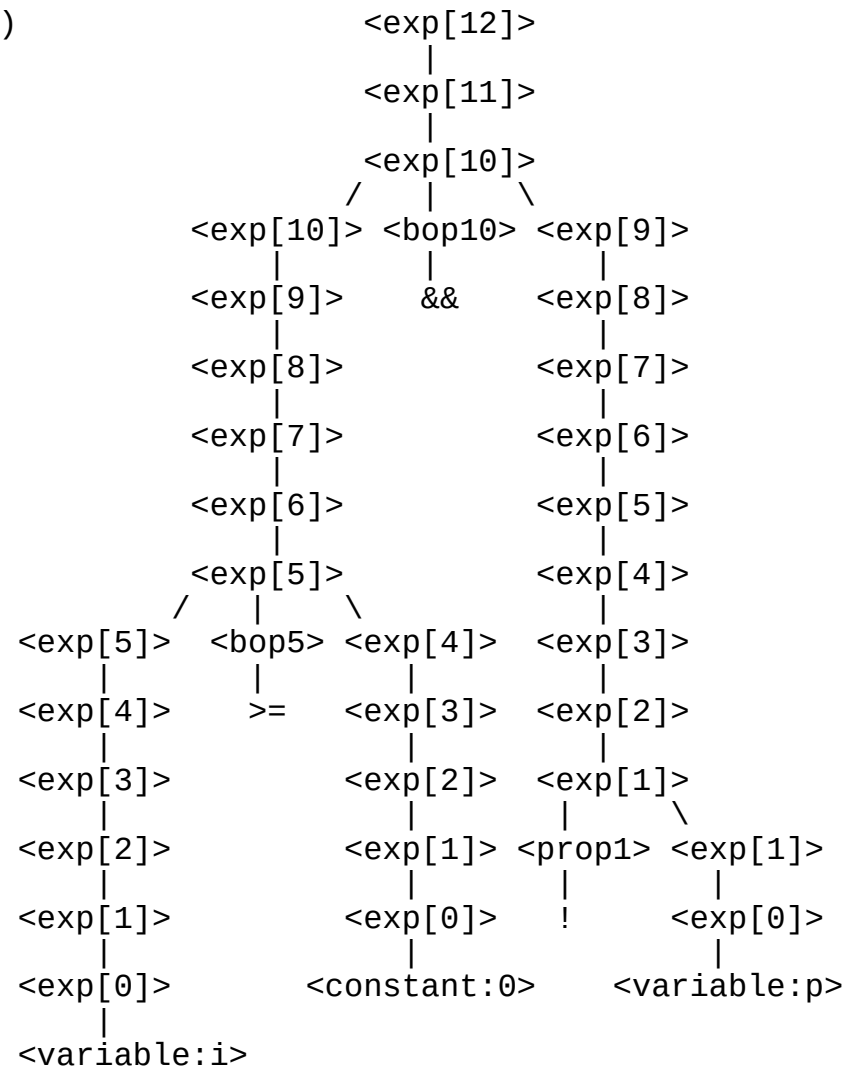
2.13 We write <bop10>, <bop6>, <bop5> and <prop1> for <class 10 binary op>, <class 6 binary op>, <class 5 binary op> and <class 1 prefix op>.

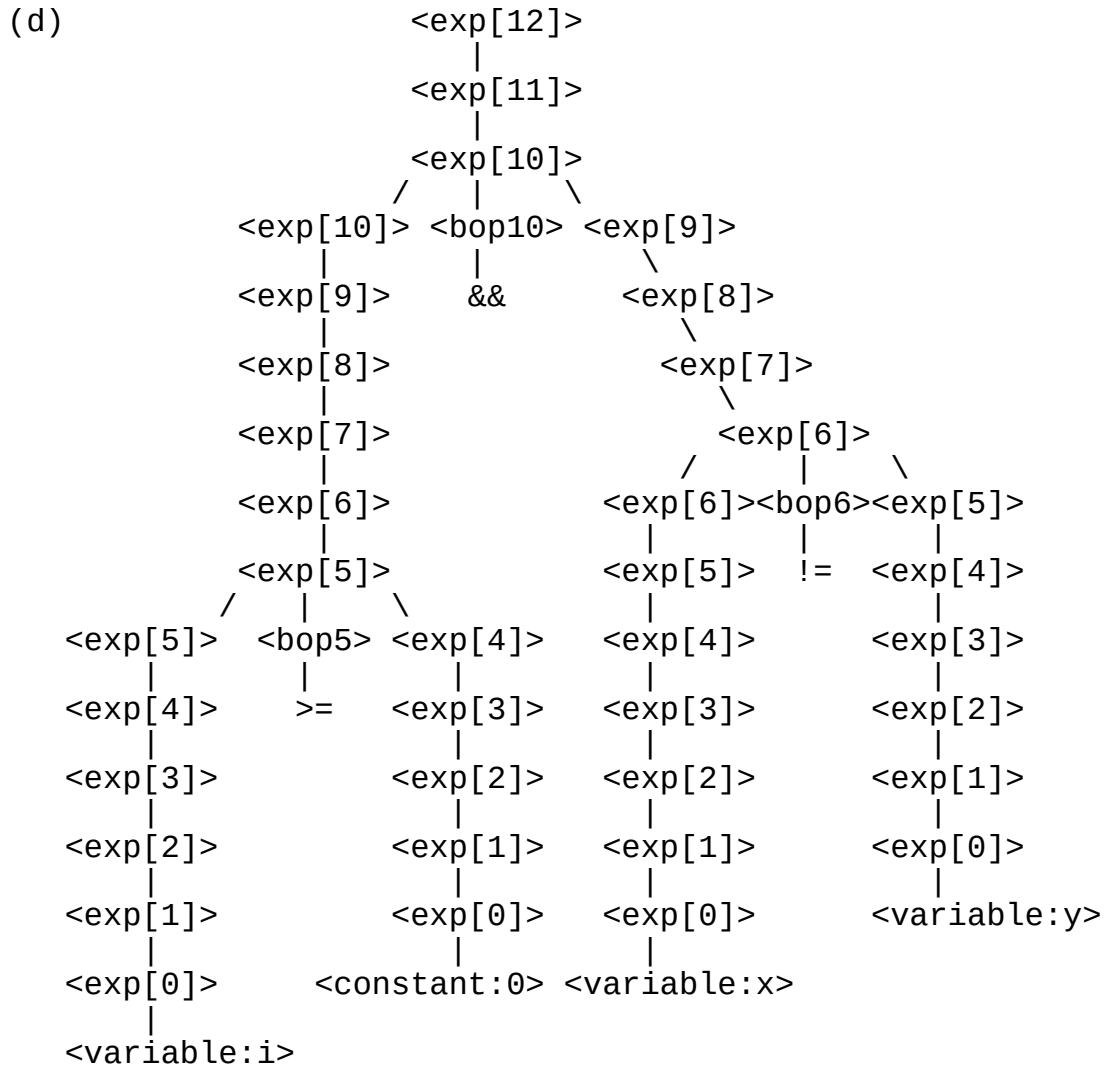
(a)




(b)
<exp[12]>
<exp[11]>
<exp[10]>
<exp[10]> <bop10> <exp[9]>
<exp[9]> && <exp[8]>
<exp[2]> <exp[1]>
<exp[1]> <prop1> <exp[1]>
<exp[0]> ! <exp[0]>
(<exp[12]>) <variable:p>
<exp[11]>
<exp[5]>
<exp[5]> <bop5> <exp[4]>
<exp[4]> >= <exp[3]>
<exp[3]> <exp[2]>
<exp[2]> <exp[1]>
<exp[1]> <exp[0]>
<exp[0]> <constant:0>
<variable:i>

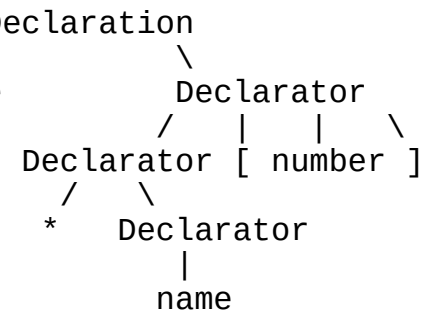
(c)

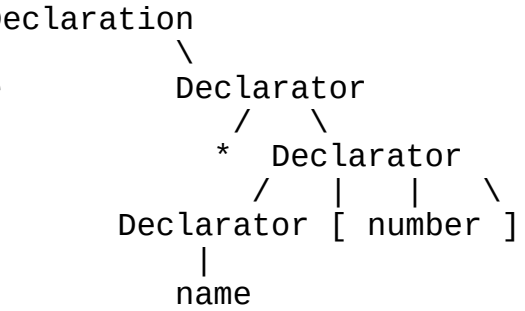

<exp[12]>
<exp[11]>
<exp[10]>
<exp[10]> <bop10> <exp[9]>
<exp[9]> && <exp[8]>
<exp[8]> <exp[7]>
<exp[7]> <exp[6]>
<exp[6]> <exp[5]>
<exp[5]> <exp[4]>
<exp[5]> <bop5> <exp[4]> <exp[3]>
<exp[4]> >= <exp[3]> <exp[2]>
<exp[3]> <exp[2]> <exp[1]>
<exp[2]> <exp[1]> <prop1> <exp[1]>
<exp[1]> <exp[0]> ! <exp[0]>
<exp[0]> <constant:0> <variable:p>
<variable:i>


(d)
<exp[12]>
<exp[11]>
<exp[10]>
<exp[10]> <bop10> <exp[9]>
<exp[9]> && <exp[8]>
<exp[8]> <exp[7]>
<exp[7]> <exp[6]>
<exp[6]> <exp[6]><bop6><exp[5]>
<exp[5]> <exp[5]> != <exp[4]>
<exp[5]> <bop5> <exp[4]> <exp[4]> <exp[3]>
<exp[4]> >= <exp[3]> <exp[3]> <exp[2]>
<exp[3]> <exp[2]> <exp[2]> <exp[1]>
<exp[2]> <exp[1]> <exp[1]> <exp[0]>
<exp[1]> <exp[0]> <exp[0]> <variable:y>
<exp[0]> <constant:0> <variable:x>
<variable:i>

2.15 (a) Any Declaration that involves * Declarator (Type)
or * Declarator [number] has more than one parse tree.

For example, int * name [number] has two parse trees:





2.15 (b) Decl0 ::= name | (Decl2)
Decl1 ::= Decl0 | Decl1 <class 1 postfix op>
Decl2 ::= Decl1 | <class 2 prefix op> Decl2

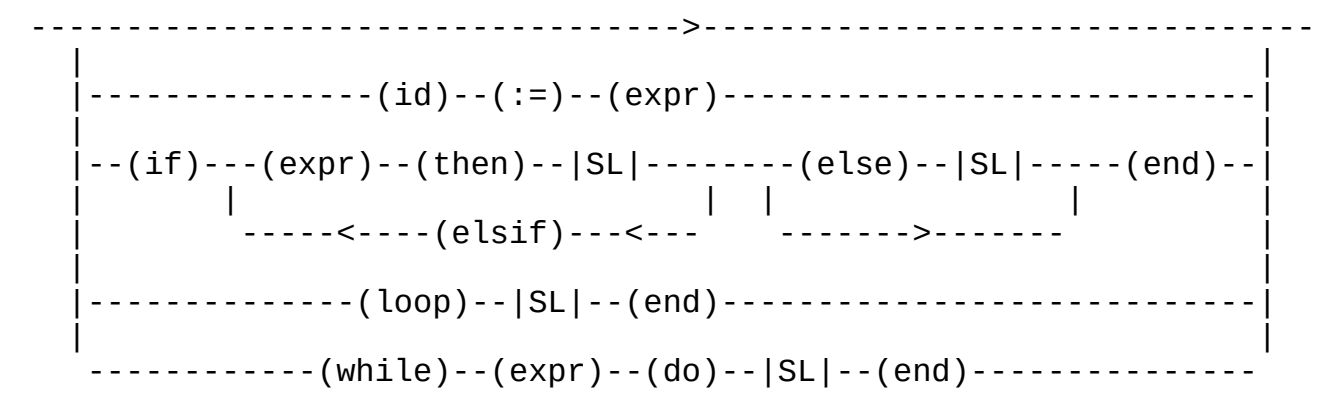
<class 1 postfix op> ::= [number] | (Type)
<class 2 prefix op> ::= *

Decl2 is the starting nonterminal.

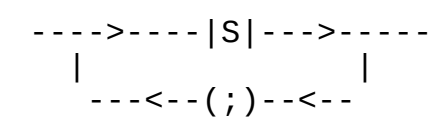
2.16 (b) NOTATION: (...) denotes an oval node.
|...| denotes a rectangular node.

In the given grammar expr is a terminal, even though it would be a non-terminal in a larger grammar that specifies the syntax of an entire programming language. That is why expr nodes in the syntax charts below are oval.

S


(id)--(:=)--(expr)
(if)---(expr)--(then)--|SL|--------(else)--|SL|-----(end)--
(elsif)---<---
------->-------
(loop)--|SL|--(end)
(while)--(expr)--(do)--|SL|--(end)

SL


|S|
(;)