2. Give two different derivations of the sentence "**the boy with a cat sang a song.**", but show that the derivations produce the same derivation tree.

<sentence> ::= <noun phrase> <verb phrase> .
<noun phrase> ::= <determiner> <noun>
| <determiner> <noun> <prepositional phrase>
<verb phrase> ::= <verb> | <verb> <noun phrase>
| <verb> <noun phrase> <prepositional phrase>
<prepositional phrase> ::= <preposition> <noun phrase>
<noun> ::= **boy** | **girl** | **cat** | **telescope** | **song** | **feather**
<determiner> ::= **a** | **the**
<verb> ::= **saw** | **touched** | **surprised** | **sang**
<preposition> ::= **by** | **with**

<sentence> ⇒ <noun phrase> <verb phrase>.

⇒ <determiner> <noun> <prepositional phrase> <verb phrase>.

⇒ the <noun> <prepositional phrase> <verb phrase>.

⇒ the boy <prepositional phrase> <verb phrase>.

⇒ the boy <preposition> <noun phrase> <verb phrase>.

⇒ the boy with <noun phrase> <verb phrase>.

⇒ the boy with <determiner> <noun> <verb phrase>.

⇒ the boy with a <noun> <verb phrase>.

⇒ the boy with a cat <verb phrase>.

$\Rightarrow$ the boy with a cat <verb> <noun phrase>.

$\Rightarrow$ the boy with a cat sang <noun phrase>.

$\Rightarrow$ the boy with a cat sang <determiner> <noun>.

$\Rightarrow$ the boy with a cat sang a <noun>.

$\Rightarrow$ the boy with a cat sang a song.

Right-most derivation:

<sentence> $\Rightarrow$ <noun phrase> <verb phrase>.

$\Rightarrow$ <noun phrase> <verb> <noun phrase>.

$\Rightarrow$ <noun phrase> <verb> <determiner> <noun>.

$\Rightarrow$ <noun phrase> <verb> <determiner> song.

$\Rightarrow$ <noun phrase> <verb> a song.

$\Rightarrow$ <noun phrase> sang a song.

$\Rightarrow$ <determiner> <noun> <prepositional phrase> sang a song.

$\Rightarrow$ <determiner> <noun> <preposition> <noun phrase> sang a song.

$\Rightarrow$ <determiner> <noun> <preposition> <determiner> <noun> sang a song.

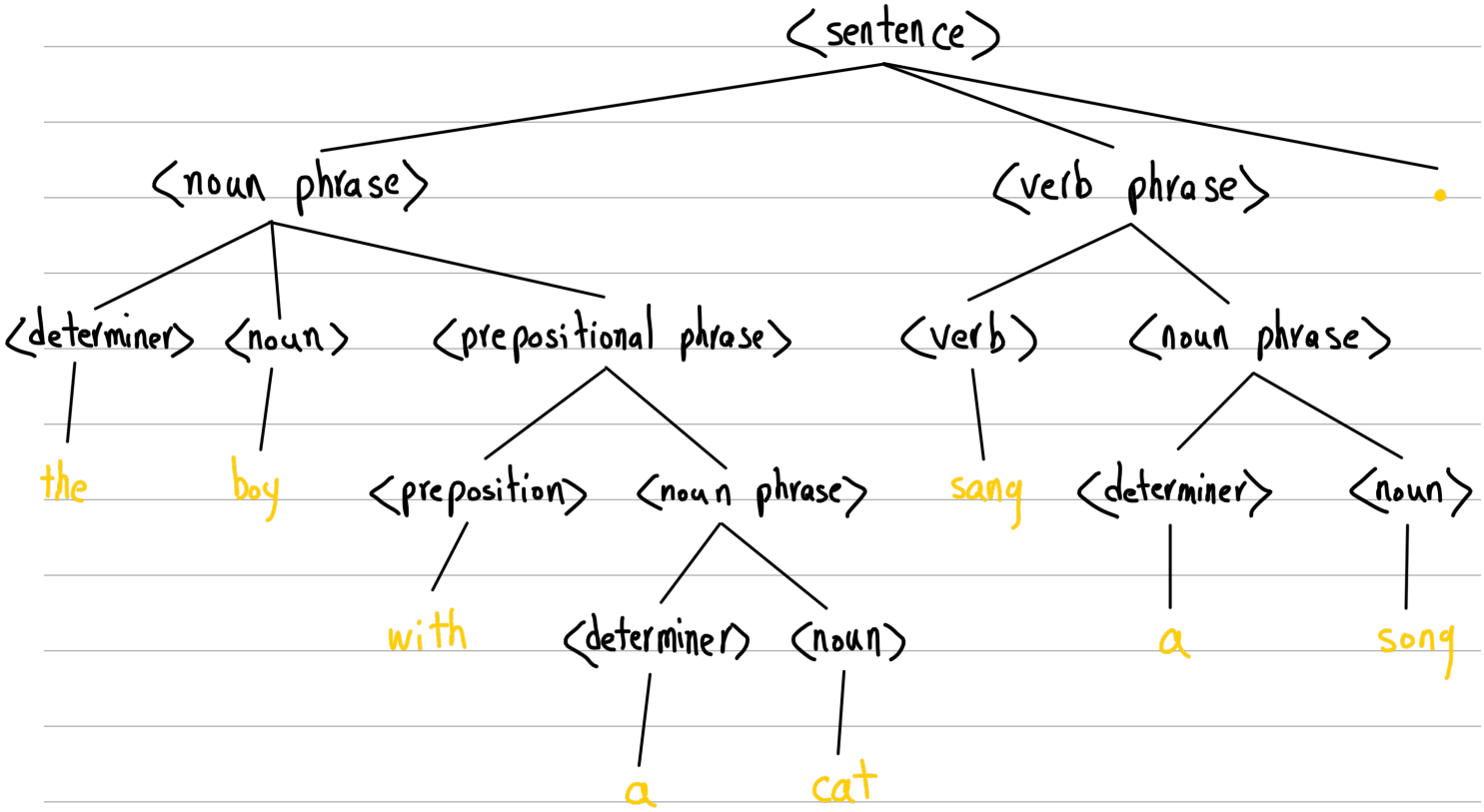
$\Rightarrow$ <determiner> <noun> <preposition> <determiner> cat sang a song.

$\Rightarrow$ <determiner> <noun> <preposition> a cat sang a song.

$\Rightarrow$ <determiner> <noun> with a cat sang a song.

$\Rightarrow$ <determiner> boy with a cat sang a song.

$\Rightarrow$ the boy with a cat sang a song.



There is no difference in derivation tree of both approaches.

3. Look up the following terms in a dictionary: *linguistics*, *semiotics*, *grammar*, *syntax*, *semantics*, and *pragmatics*.

Linguistics: the study of language in general and of particular languages, their structure, grammar, and history

Semiotics: the way in which people communicate through signs and images, or the study of this

Grammer: the rules by which words change their forms and are combined into sentences, or the study or use of these rules

Syntax: 1 the way words are arranged to form sentences or phrases, or the rules of grammar which control this
2 the rules that describe how words and phrases are used in a computer language

Semantics: 1 the study of the meaning of words and phrases
2 the meaning of a word or expression

Pragmatics: the study of how words and phrases are used with special meanings in particular situations

5. Using the grammar in Figure 1.6, derive the <sentence> **aaabbbccc** .

<sentence> ::= **abc** | **a**<thing>**bc**
<thing>**b** ::= **b**<thing>
<thing>**c** ::= <other>**bcc**
a<other> ::= **aa** | **aa**<thing>
b<other> ::= <other>**b**

Figure 1.6: A Context-Sensitive Grammar

<sentence> => a <thing> bc

=> a b <thing> c

=> ab <other> bcc

=> a <other> b bcc

=> aa <thing> bb cc

=> aa b <thing> b c c

=> aa bb <thing> cc

=> aa bb <other> b ccc

=> aa b <other> bb ccc

=> aa <other> bbb ccc

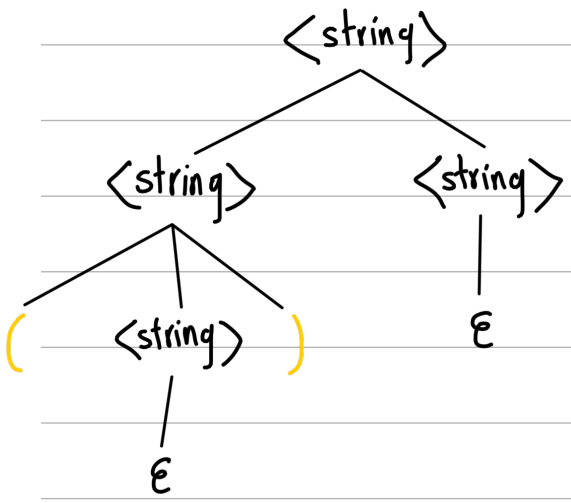
=> aaa bbb ccc

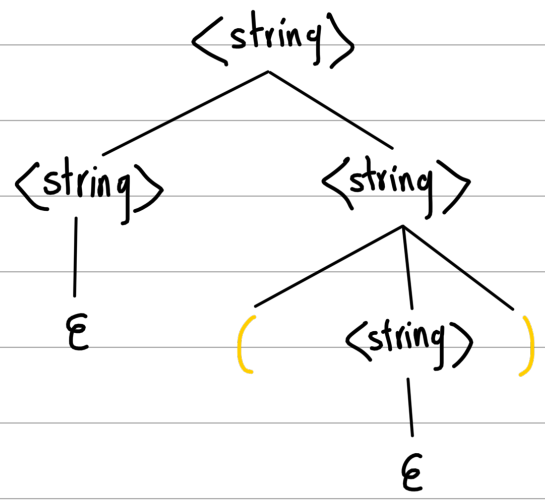
6. Consider the following two grammars, each of which generates strings of correctly balanced parentheses and brackets. Determine if either or both is ambiguous. The Greek letter ε represents an empty string.

   a) <string> ::= <string> <string> | (<string>) |[<string>] | ε

   b) <string> ::= (<string>) <string> | [<string>] <string> | ε

(a) derivation trees for ( )





2 different derivation for a sentence → the grammer is ambiguous

(b) At each step, if you want to add brackets or parentheses, you only have one choice for production rules so there is no ambiguity.

7. Describe the languages over the terminal set { **a**, **b** } defined by each of the following grammars:

   a) <string> ::= **a** <string> **b** | **ab**

   b) <string> ::= **a** <string> **a** | **b** <string> **b** | ε

   c) <string>::= **a** <B> | **b** <A>
   <A> ::= **a** | **a** <string> | **b** <A> <A>
   <B> ::= **b** | **b** <string> | **a** <B> <B>

(a) $L(G) = \{ a^m b^m \mid m \geq 1 \}$

(b) $L(G) = \{ ww^R \mid w \in \{a,b\}^* \}$

(c) $L(G) = \{ w \mid n_w(a) = n_w(b) \geq 1 \}$

9. Identify which productions in the English grammar of Figure 1.1 can be reformulated as type 3 productions. It can be proved that productions of the form <A> ::= a_1 a_2 a_3 ...a_n <B> are also allowable in regular grammars. Given this fact, prove the English grammar is regular—that is, it can be defined by a type 3 grammar. Reduce the size of the language by limiting the terminal vocabulary to **boy**, **a**, **saw**, and **by** and omit the period. This exercise requires showing that the concatenation of two regular grammars is regular.

<sentence> ::= <noun phrase> <verb phrase>
<noun phrase> ::= <determiner> <noun>
| <determiner> <noun> <prepositional phrase>
<verb phrase> ::= <verb> | <verb> <noun phrase>
| <verb> <noun phrase> <prepositional phrase>
<prepositional phrase> ::= <preposition> <noun phrase>
<noun> ::= **boy**
<determiner> ::= **a**
<verb> ::= **saw**
<preposition> ::= **by**

<sentence> ::= <noun phrase> <verb phrase>

<verb phrase> ::= saw | saw <noun phrase> | saw <noun phrase> <prepositional phrase>

<noun phrase> ::= a boy | a boy <prepositional phrase>

<prepositional phrase> ::= by <noun phrase>

<var1> ::= <noun phrase> <verb phrase> <var2> ::= <noun phrase> <prepositional phrase>

باز کردن بست راست این 2 متغیر بدید :

<var1> ::= a boy <verb phrase> | a boy <prepositional phrase> <verb phrase>

::= a boy <verb phrase> | a boy by <noun phrase> <verb phrase>

::= a boy <verb phrase> | a boy by <var1>

<var2> ::= a boy <prepositional phrase> |

a boy <prepositional phrase> <prepositional phrase>

::= a boy <prepositional phrase> | a boy by <noun phrase> <prepositional phrase>

::= a boy <prepositional phrase> | a boy by <var2>

حال گرامر اولیه به گرامر زیر تبدیل می شود:

<sentence> ::= <var1>

<verb phrase> ::= saw | saw <noun phrase> | saw <var2>

<noun phrase> ::= a boy | a boy <prepositional phrase>

<prepositional phrase> ::= by <noun phrase>

<var1> ::= a boy <verb phrase> | a boy by <var1>

<var2> ::= a boy <prepositional phrase> | a boy by <var2>

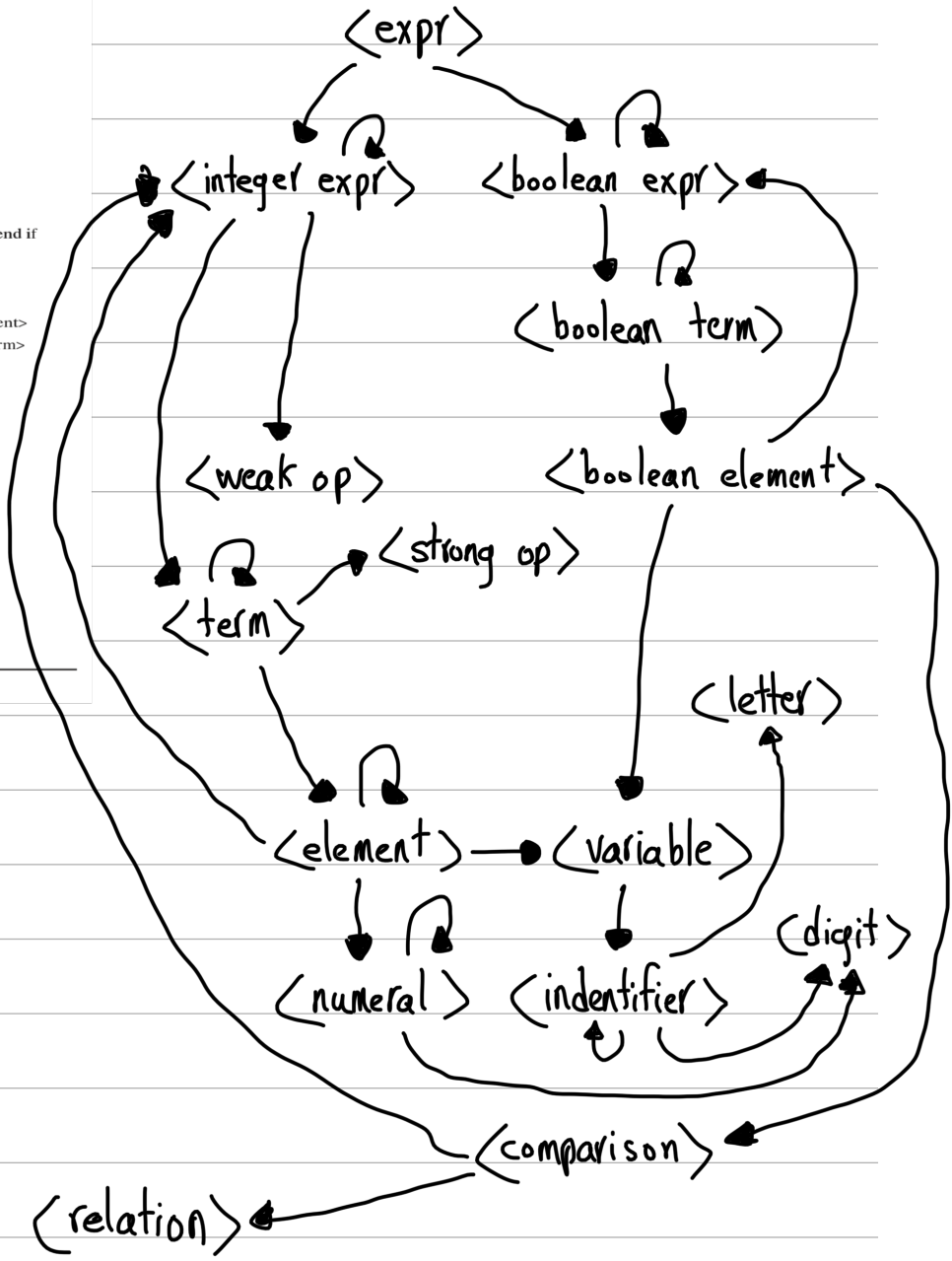
با توجه به فرض مطرح شده در صورت سوال، گرامر بالا متضاد است.

page 16

1. Draw a dependency graph for the nonterminal <expr> in the BNF definition of Wren.

```
<program> ::= program <identifier> is <block>
<block> ::= <declaration seq> begin <command seq> end
<declaration seq> ::= ε | <declaration> <declaration seq>
<declaration> ::= var <variable list> : <type> ;
<type> ::= integer | boolean
<variable list> ::= <variable> | <variable> , <variable list>
<command seq> ::= <command> | <command> ; <command seq>
<command> ::= <variable> := <expr> | skip
    | read <variable> | write <integer expr>
    | while <boolean expr> do <command seq> end while
    | if <boolean expr> then <command seq> end if
    | if <boolean expr> then <command seq> else <command seq> end if
<expr> ::= <integer expr> | <boolean expr>
<integer expr> ::= <term> | <integer expr> <weak op> <term>
<term> ::= <element> | <term> <strong op> <element>
<element> ::= <numeral> | <variable> | ( <integer expr> ) | – <element>
<boolean expr> ::= <boolean term> | <boolean expr> or <boolean term>
<boolean term> ::= <boolean element>
                 | <boolean term> and <boolean element>
<boolean element> ::= true | false | <variable> | <comparison>
                    | not ( <boolean expr> ) | ( <boolean expr> )
<comparison> ::= <integer expr> <relation> <integer expr>
<variable> ::= <identifier>
<relation> ::= <= | < | = | > | >= | <>
<weak op> ::= + | –
<strong op> ::= * | /
<identifier> ::= <letter> | <identifier> <letter> | <identifier> <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
<numeral> ::= <digit> | <digit> <numeral>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

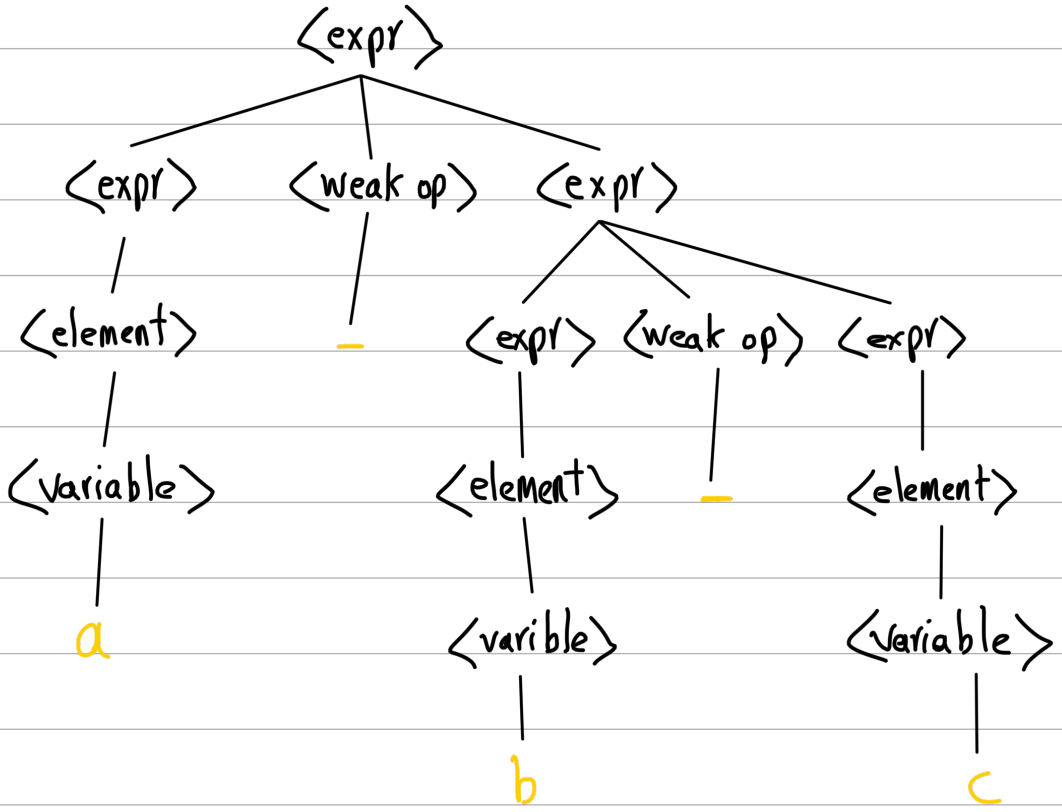
Figure 1.8: BNF for Wren

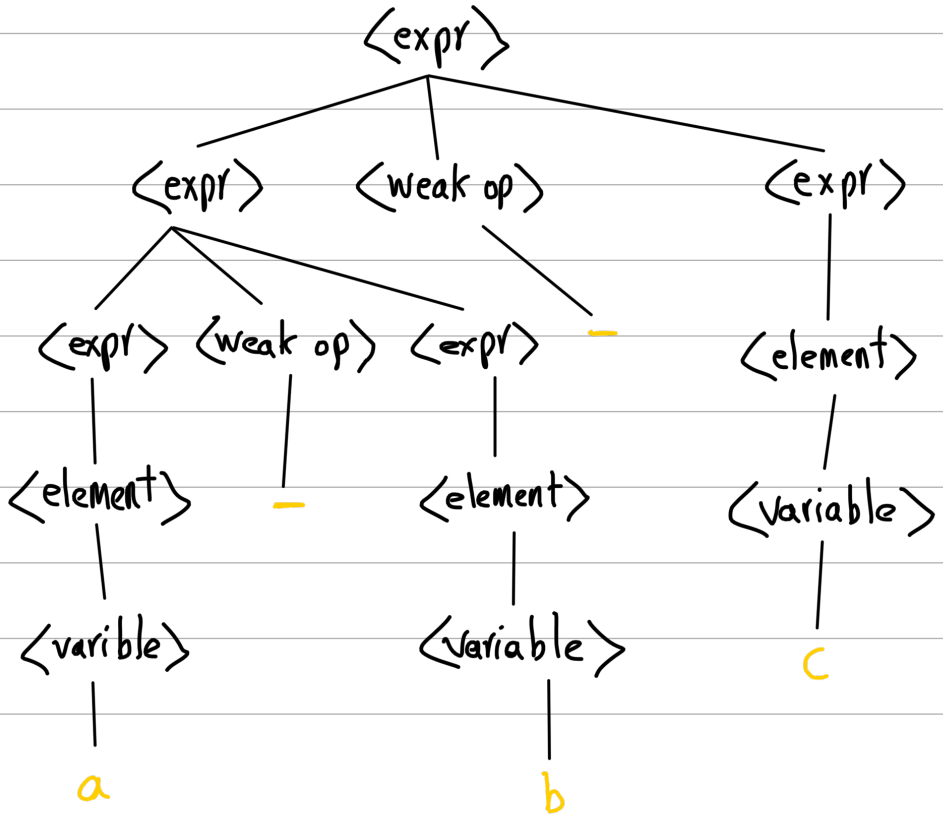


2. Consider the following specification of expressions:

<expr> ::= <element> | <expr> <weak op> <expr>
<element> ::= <numeral> | <variable>
<weak op> ::= + | –

Demonstrate its ambiguity by displaying two derivation trees for the expression "**a–b–c**". Explain how the Wren specification avoids this problem.





Wren avoids this problem by limiting recursion from the right side.

```
<expr> ::= <integer expr> | <boolean expr>
<integer expr> ::= <term> | <integer expr> <weak op> <term>
<term> ::= <element> | <term> <strong op> <element>
<element> ::= <numeral> | <variable> | ( <integer expr> ) | – <element>
```

3. This Wren program has a number of errors. Classify them as context-free, context-sensitive, or semantic.

b is declared as integer and boolean.

```
program errors was
        var a,b : integer ;
        var p,b ; boolean ;
    begin
        a := 34;
        if b≠0 then p := true else p := (a+1);
        write p; write q
    end
```

was → is

; → :

p can not be assigned by an integer

not includes in relation

end if

q is not declared

write is used for integer expr

<program> ::= **program** <identifier> **is** <block>
<block> ::= <declaration seq> **begin** <command seq> **end**
<declaration seq> ::= ε | <declaration> <declaration seq>
<declaration> ::= **var** <variable list> : <type> ;
<type> ::= **integer** | **boolean**
<variable list> ::= <variable> | <variable> , <variable list>
<command seq> ::= <command> | <command> ; <command seq>
<command> ::= <variable> := <expr> | **skip**
| **read** <variable> | **write** <integer expr>
| **while** <boolean expr> **do** <command seq> **end while**
| **if** <boolean expr> **then** <command seq> **end if**
| **if** <boolean expr> **then** <command seq> **else** <command seq> **end if**
<expr> ::= <integer expr> | <boolean expr>
<integer expr> ::= <term> | <integer expr> <weak op> <term>
<term> ::= <element> | <term> <strong op> <element>
<element> ::= <numeral> | <variable> | (<integer expr>) | – <element>
<boolean expr> ::= <boolean term> | <boolean expr> **or** <boolean term>
<boolean term> ::= <boolean element>
| <boolean term> **and** <boolean element>
<boolean element> ::= **true** | **false** | <variable> | <comparison>
| **not** (<boolean expr>) | (<boolean expr>)
<comparison> ::= <integer expr> <relation> <integer expr>
<variable> ::= <identifier>
<relation> ::= **<=** | **<** | **=** | **>** | **>=** | **<>**
<weak op> ::= **+** | **–**
<strong op> ::= ***** | **/**
<identifier> ::= <letter> | <identifier> <letter> | <identifier> <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**
<numeral> ::= <digit> | <digit> <numeral>
<digit> ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

Figure 1.8: BNF for Wren

Since we don't exactly know for what purpose this program was written, we can't surely indentify the semantic errors.

5. This BNF grammar defines expressions with three operations, *, -, and +, and the variables "a", "b", "c", and "d".

<expr> ::= <thing> | <thing> * <expr>

<object> ::= <element> | <element> – <object>

<thing> ::= <object> | <thing> + <object>

<element> ::= **a** | **b** | **c** | **d** | (<object>)

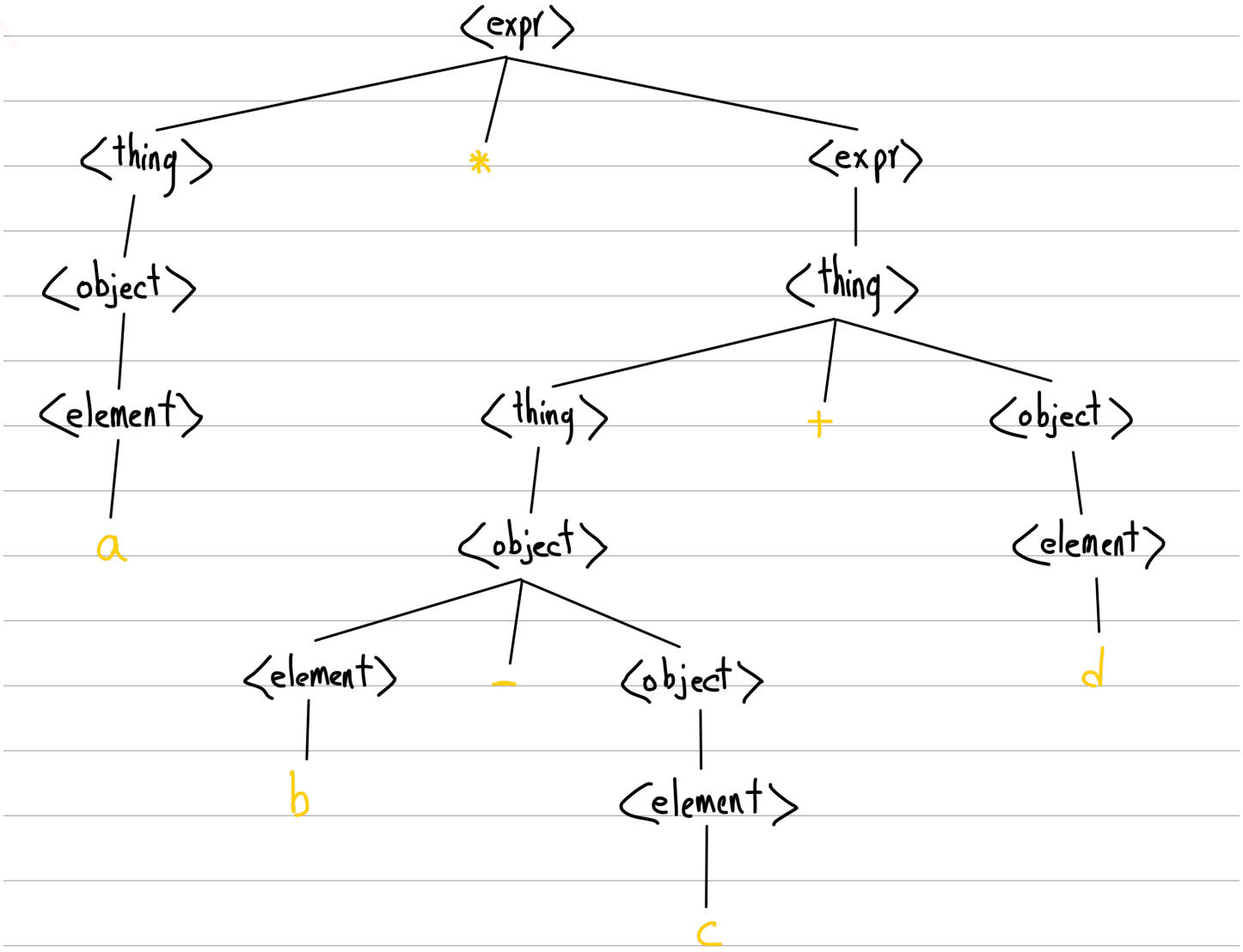
a) Give the order of precedence among the three operations.

b) Give the order (left-to-right or right-to-left) of execution for each operation.

c) Explain how the parentheses defined for the nonterminal <element> may be used in these expressions. Describe their limitations.

(a) As deep as an operation goes in a derivation tree, it has higher precedence.

Derivation tree of a * b - c + d: (next page)



order of precedence : - , + , *

(b) - => right to left + => left to right * => right to left

(c) The parentheses are used for clarifying precedence of - operations only and it can't help us with + and * orders of precedence.

8. Write a BNF specification of the syntax of the Roman numerals less than 100. Use this grammar to derive the string "XLVII".

S ::= B A

B ::= ten | twenty | thirty | fourty | fifty | sixty | seventy | eigdty | ninety | ε

A ::= one | two | three | four | five | six | seven | eight | nine | ε

ten ::= X twenty ::= XX thirty ::= XXX fourty ::= ten fifty fifty ::= L

sixty ::= fifty ten seventy ::= fifty twenty eighty ::= fifty thirty

ninety ::= ten hundred hundred ::= C

one ::= I two ::= II three ::= III five ::= V

four ::= one five six ::= five one seven ::= five two eight ::= five three

nine ::= one ten

10. Show that the following grammar for expressions is ambiguous and provide an alternative unambiguous grammar that defines the same set of expressions.

<expr> ::= <term> | <factor>
<term> ::= <factor> | <expr> + <term>
<factor> ::= <ident> | (<expr>) | <expr> * <factor>
<ident> ::= a | b | c

Since we have 2 derivations for <factor>, the grammer is ambiguous:

<expr> ::= <factor>

and

<expr> ::= <term> ::= <factor>

Alternative:

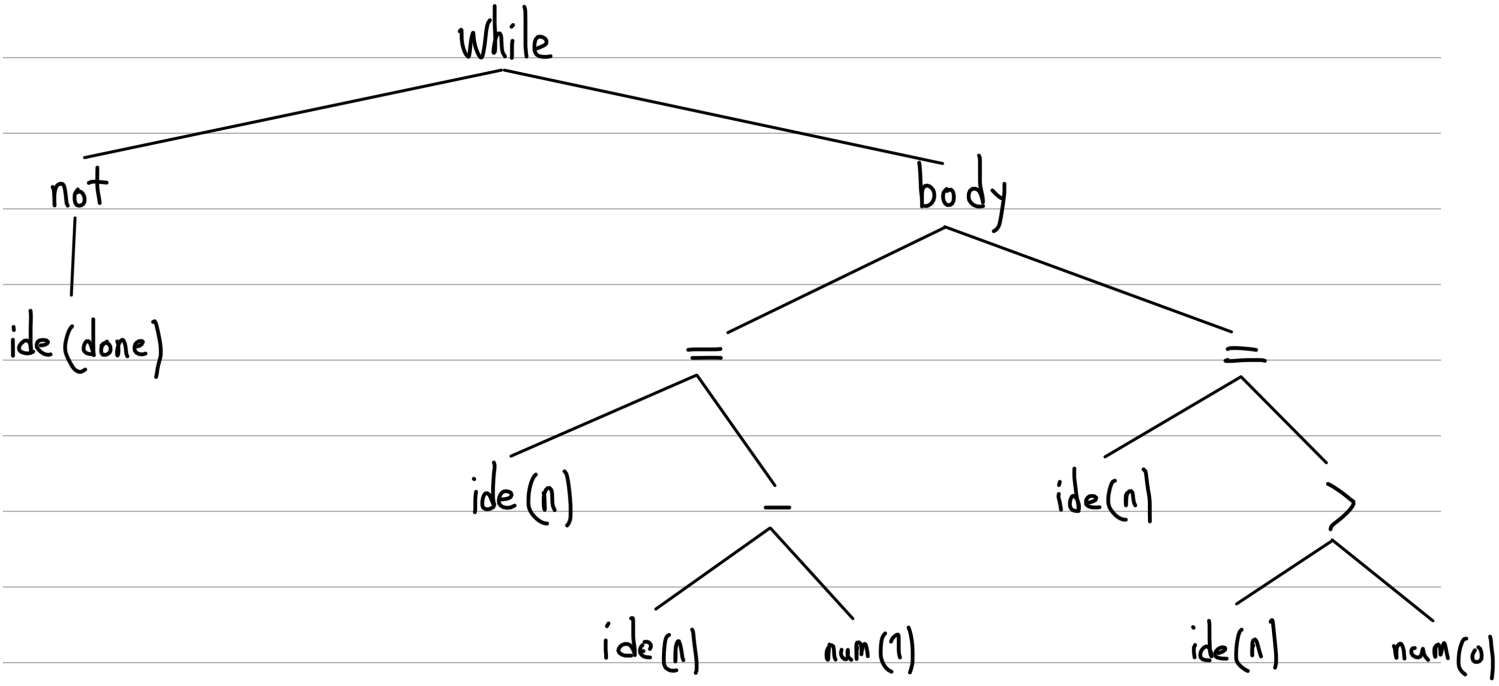
<expr> ::= <term>

<term> ::= <factor> | <factor> + <term>

<factor> ::= <ident> | <ident> * <factor>

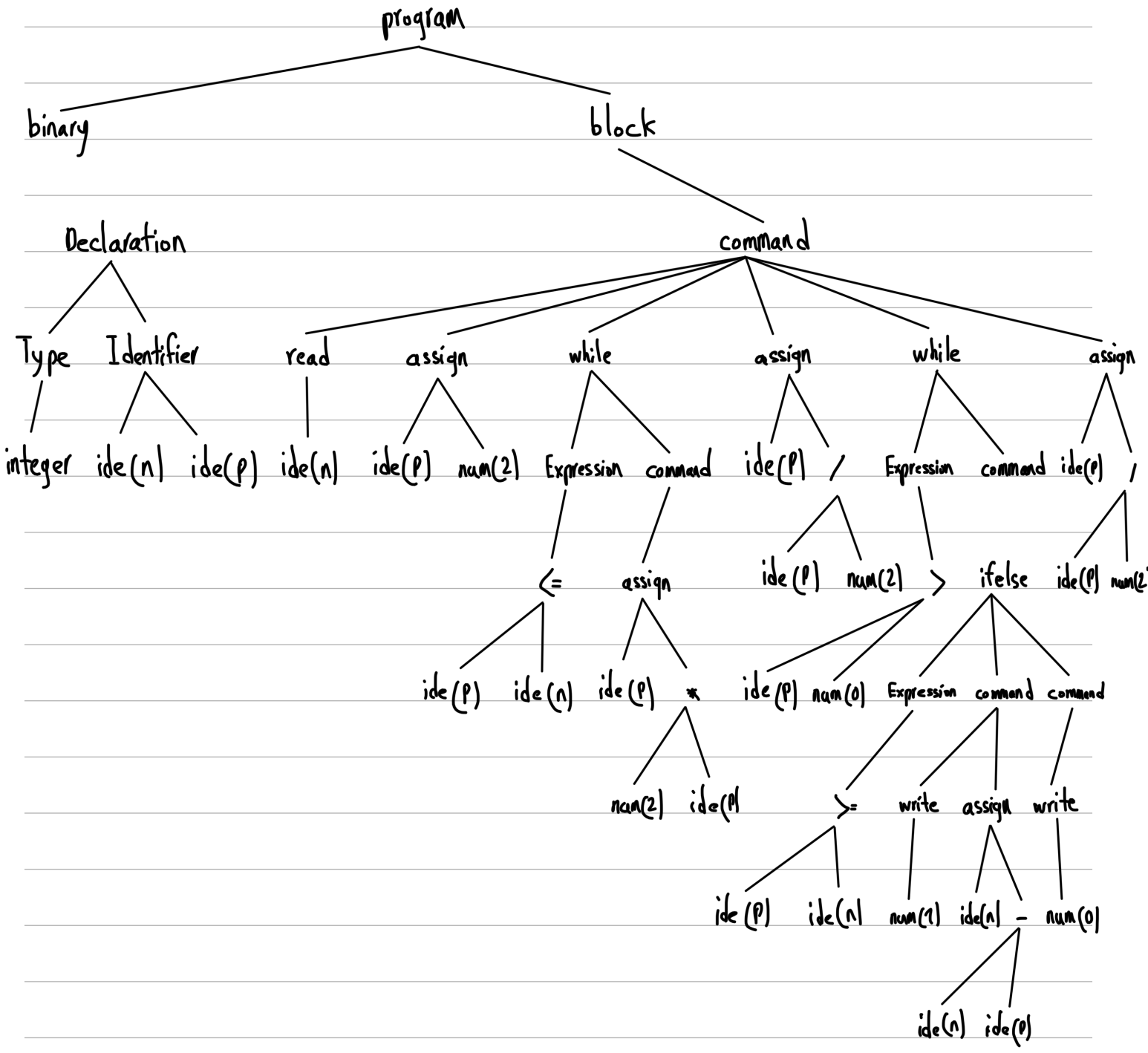
<ident> ::= a | b | c | (<expr>)

2. Parse the following token list to produce an abstract syntax tree:
[while, not, lparen, ide(done), rparen, do, ide(n), assign,
ide(n), minus, num(1), semicolon, ide(done), assign,
ide(n), greater, num(0), end, while]



3. Draw an abstract syntax tree for the following Wren program:

```
program binary is
    var n,p : integer ;
begin
    read n; p := 2;
    while p<=n do p := 2*p end while ;
    p := p/2;
    while p>0 do
        if n>= p  then write  1; n := n–p else write  0 end if ;
        p := p/2
    end while
end
```



1 4 5 10 11

Page 71

1. In old versions of Fortran that did not have the character data type, character strings were expressed in the following format:

<string literal> ::= <numeral> H <string>

where the <numeral> is a base-ten integer (≥ 1), H is a keyword (named after Herman Hollerith), and <string> is a sequence of characters. The semantics of this string literal is correct if the numeric value of the base-ten numeral matches the length of the string. Write an attribute grammar using only synthesized attributes for the nonterminals in the definition of <string literal>.

<string literal> ::= <numeral> H <string>

Condition: Val(<numeral>) == Len(<string>)

$\langle string_1 \rangle ::= \langle string_2 \rangle \langle char \rangle$

$Len(\langle string_1 \rangle) = Len(\langle string_2 \rangle) + 1$

<string> ::= <char>

Len <string> = 1

$\langle numeral_1 \rangle ::= \langle numeral_2 \rangle \langle digit \rangle$

$Val(\langle numeral_1 \rangle) = Val(\langle numeral_2 \rangle) \times 10 + Val(\langle digit \rangle)$

<numeral> ::= <digit>

Val(<numeral>) = Val(<digit>)

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Val(<digit>) = <digit>

4. The following BNF specification defines the language of Roman numerals less than 1000:

<roman> ::= <hundreds> <tens> <units>

<hundreds> ::= <low hundreds> | **CD** | **D** <low hundreds> | **CM**

<low hundreds> ::= ε | <low hundreds> **C**

<tens> ::= <low tens> | **XL** | **L** <low tens> | **XC**

<low tens> ::= ε | <low tens> **X**

<units> ::= <low units> | **IV** | **V** <low units> | **IX**

<low units> ::= ε | <low units> **I**

Define attributes for this grammar to carry out two tasks:

a) Restrict the number of X's in <low tens>, the I's in <low units>, and the C's in <low hundreds> to no more than three.

b) Provide an attribute for <roman> that gives the decimal value of the Roman numeral being defined.

Define any other attributes needed for these tasks, but do not change the BNF grammar.

<roman> :: <hundreds> <tens> <units>

Val(<roman>) = Val(<hundreds>) + Val(<tens>) + Val(<units>)

<hundreds> ::= <low hundreds>

Val(<hundreds>) = Val(<low hundreds>)

| CD

Val(<hundreds>) = 400

| D <low hundreds>

Val(<hundreds>) = 500 + Val(<low hundreds>)

| CM

Val(<hundreds>) = 900

<low hundreds> ::= ε

Val(<low hundreds>) = 0

| <low hundreds$_1$> C

Val(<low hundreds>) = Val(<low hundreds$_1$>) + 100

<tens> ::= <low tens>

Val(<tens>) = Val(<low tens>)

| XL

Val(<tens>) = 40

| L <low tens>

Val(<tens>) = 50 + Val(<low tens>)

| XC

Val(<tens>) = 90

<low tens> ::= ε

Val(<low tens>) = 0

| <low tens> X

Val(<low tens>$_1$) = Val(<low tens>$_2$) + 10

<units> ::= <low units>

Val(<units>) = Val(<low units>)

| IV

Val(<units>) = 4

| V <low units>

Val(<units>) = 5 + Val(<low units>)

| IX

Val(<units>) = 9

<low units> ::= ε

Freq(<low units>) = 0

Val(<low units>) = 0

| <low units> I

$$\text{Freq}(\langle \text{low units}_1 \rangle) = \text{Freq}(\langle \text{low units}_2 \rangle) + 1$$

$$\text{Val}(\langle \text{low units}_1 \rangle) = \text{Val}(\langle \text{low units}_2 \rangle) + 1$$

Condition: $\text{Freq}(\langle \text{low units} \rangle) \leq 3$

5. Expand the binary numeral attribute grammar (either version) to allow for binary numerals with no binary point (1101), binary fractions with no fraction part (101.), and binary fractions with no whole number part (.101).

<binary numerals> ::= <binary digit>

Val(<binary numerals>) = Val(<binary digit>)

Pos(<binary digit>) = 0

| <binary digit> .

Val(<binary numerals>) = Val(<binary digit>)

Pos(<binary digit>) = 0

| . <fraction digit>

Val(<binary numeral>) = Val(<fraction digit>)

10. Consider a language of expressions with only the variables a, b, and c and formed using the binary infix operators

+, –, *, /, and ↑ (for exponentiation)

where ↑ has the highest precedence, * and / have the same next lower precedence, and + and – have the lowest precedence. ↑ is to be right associative and the other operations are to be left associative. Parentheses may be used to override these rules. Provide a BNF specification of this language of expressions. Add attributes to your BNF specification so that the following (unusual) conditions are satisfied by every valid expression accepted by the attribute grammar:

a) The maximum depth of parenthesis nesting is three.

b) No valid expression has more than eight applications of operators.

c) If an expression has more divisions than multiplications, then subtractions are forbidden.

<expr> ::= <integer expr>

ParCount(<expr>) = ParCount(<integer expr>)

OPNum(<expr>) = OPNum(<integer expr>)

MulNum(<expr>) = MulNum(<integer expr>)

DivNum(<expr>) = DivNum(<integer expr>)

SubNum(<expr>) = SubNum(<integer expr>)

Condition: ParCount(<expr>) ≤ 3

Condition: OPNum(<expr>) ≤ 8

Condition: MulNum(<expr>) ≥ DivNum(<expr>) and SubNum(<expr>) = 0

<integer expr> ::= <term1>

ParCount(<integer expr>) = ParCount(<term1>)

OPNum (<integer expr>) = OPNum (<term1>)

MulNum (<integer expr>) = MulNum <term1>)

DivNum (<integer expr>) = DivNum (<term1>)

SubNum (<integer expr>) = SubNum (<term1>)

<term1> ::= <term1> <weak op> <term2>

ParCount (<term1>) = ParCount (<term2>)

OPNum (<term1>) = OPNum (<term2>) + 1

MulNum <term1>) = MulNum <term2>)

DivNum (<term1>) = DivNum (<term2>)

SubNum (<term1>) = SubNum (<term2>) + SubNum (<weak op>

| <term2>

ParCount (<term1>) = ParCount (<term2>)

OPNum (<term1>) = OPNum (<term2>)

MulNum <term1>) = MulNum <term2>)

DivNum(<term1>) = DivNum(<term2>)

SubNum(<term1>) = SubNum(<term2>)

<term2> ::= <term2> <strong op> <term3>

ParCount(<term2>) = ParCount(<term3>)

OPNum(<term2>) = OPNum(<term3>) +1

MulNum <term2>) = MulNum(<term3>) + MulNum(<strong op>

DivNum(<term2>) = DivNum(<term3>) + DivName(<strong op>

SubNum(<term2>) = SubNum(<term3>)

| term3

ParCount(<term2>) = ParCount(<term3>)

OPNum(<term2>) = OPNum(<term3>)

MulNum <term2>) = MulNum(<term3>)

DivNum(<term2>) = DivNum(<term3>)

SubNum(<term2>) = SubNum(<term3>)

<term3> ::= <term 4> <exponent op> <term3>

ParCount(<term3>) = ParCount(<term4>)

OPNum(<term3>) = OPNum(<term4>) + 1

MulNum(<term3>) = MulNum (<term4>)

DivNum(<term3>) = DivNum(<term 4>)

SubNum(<term3>) = SubNum(<term4>)

| term 4

ParCount(<term3>) = ParCount(<term4>)

OPNum(<term3>) = OPNum(<term4>)

MulNum(<term3>) = MulNum (<term4>)

DivNum(<term3>) = DivNum(<term 4>)

SubNum(<term3>) = SubNum(<term4>)

<term 4> ::= (<integer expr>)

ParCount(<term4>) = ParCount(<integer expr>) + 1

OPNum (<term4>) = OPNum (<integer expr>)

MulNum (<term4>) = MulNum (<integer expr>)

DivNum (<term4>) = DivNum (<integer expr>)

SubNum (<term4>) = SubNum (<integer expr>)

| var

ParCount (<term4>) = 0

OPNum (<term4>) = 0

MulNum (<term4>) = 0

DivNum (<term4>) = 0

SubNum (<term4>) = 0

<var> ::= a | b | c

<weak op> ::= +

SubNum (<weak op>) = 0

| -

SubNum(<weak op>) = 1

<strong op> ::= *

MulNum(<strong op>) = 1

Div Num(<strong op>) = 0

||

MulNum(<strong op>) = 0

Div Num(<strong op>) = 1

<exponent op> ::= ↑

11. A binary tree consists of a root containing a value that is an integer, a (possibly empty) left subtree, and a (possibly empty) right subtree. Such a binary tree can be represented by a triple (Left subtree, Root, Right subtree). Let the symbol nil denote an empty tree. Examples of binary trees include:

(nil,13,nil)
represents a tree with one node labeled with the value 13.

((nil,3,nil),8,nil)
represents a tree with 8 at the root, an empty right subtree, and a nonempty left subtree with root labeled by 3 and empty subtrees.

The following BNF specification describes this representation of binary trees.

<binary tree> ::= nil | (<binary tree> <value> <binary tree>)

<value> ::= <digit> | <value> <digit>

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Augment this grammar with attributes that carry out the following tasks:

a) A binary tree is balanced if the heights of the subtrees at each interior node are within one of each other. Accept only balanced binary trees.

b) A binary search tree is a binary tree with the property that all the values in the left subtree of any node N are less than the value at N, and all the value in the right subtree of N are greater than or equal to the value at node N. Accept only binary search trees.

<binary tree> ::= nil

depth(<binary tree>) = 0

maxValue(<binary tree>) = $-\infty$

MinValue(<binary tree>) = $+\infty$

| (<binary tree$_2$>, <value>, <binary tree$_3$>)

depth(<binary tree$_1$>) = max(Depth(<binary tree$_2$>), Depth(<binary tree$_3$>)) +1

maxValue(<binary tree$_1$>) = maxValue(<binary tree$_2$>)

MinValue(<binary tree$_1$>) = MinValue(<binary tree$_3$>)

$\langle value_1 \rangle ::= \langle digit \rangle$

$Val(\langle value \rangle) = val(\langle digit \rangle)$

$\mid \langle value_2 \rangle \langle digit \rangle$

$Val(\langle value_1 \rangle) = Val(\langle value_2 \rangle) \times 10 + Val(\langle digit \rangle)$

$\langle digit \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

$Val(\langle digit \rangle) = \langle digit \rangle$