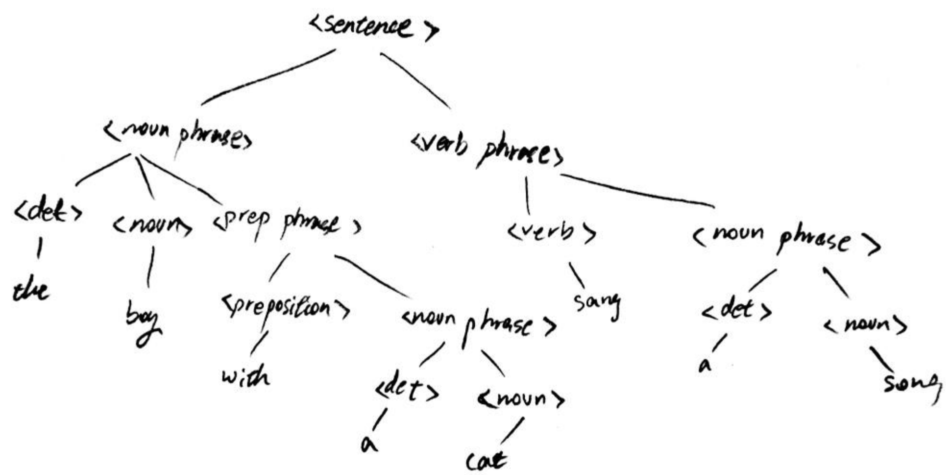
Page 8

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.

دو برداشت متفاوت می‌توان کرد:

۱) پسری که گربه داشت یک موسیقی خواند

۲) پسر با یک گربه یک موسیقی خواند



3. Look up the following terms in a dictionary:
Linguistics = the scientific study of language and its structure
Semiotics = the study of signs and symbols and their use or interpretation
Grammar = the whole system and structure of a language or of languages in general
Syntax = the arrangement of words and phrases to create well-formed sentences in a language
Semantics = the branch of linguistics and logic concerned with meaning
Pragmatics = the branch of linguistics dealing with language in use and the contexts in which it is used

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

<sentence> ::= a<thing>bc
a<thing>bc -> ab<thing>c
ab<thing>c -> ab<other>bcc
ab<other>bcc -> a<other>bbcc
a<other>bbcc -> aa<thing>bbcc
aa<thing>bbcc -> aab<thing>bcc
aab<thing>bcc -> aabb<thing>cc
aabb<thing>cc -> aabb<other>bccc
aabb<other>bccc -> aab<other>bbccc
aab<other>bbccc -> aa<other>bbbccc
aa<other>bbbccc -> aaabbbccc

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

گرامر a ambiguous است. () را در نظر بگیرید؛ دو اشتقاق برای آن وجود دارد:

<string> -> <string><string>

-> (<string>)<string>

-> (ε)<string>

-> ()ε

-> ()

<string> -> <string><string>

-> <string>(<string>)

-> <string>(ε)

-> ε()

-> ()

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

گرامر b ambiguous نیست. چرا که در مرحله فقط از چپ اشتقاق پیدا می‌کند؛ پس تنها یک درخت اشتقاق خواهیم‌داشت.

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

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

$\{a^n b^n\}$

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

{ w ∈ {a, b}* | w = wR and |w| is even }

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

<A> ::= a | a <string> | b <A> <A>

<B> ::= b | b <string> | a <B> <B>

{ w ∈ {a, b}* | numberOf(a) == numberOf(b) }

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> ::= a1 a2 a3 …an <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.

تمام قوانین regular هستند یا می‌توانند به‌صورت regular بازنویسی شوند.

مفروضات:

<noun> ::= boy

<determiner> ::= a

<verb> ::= saw

<preposition> ::= by

بازنویسی:

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

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

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

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

در این قواعد، تنها در سمت راست non-terminal می‌بینیم، پس regular هستند.

اثبات اینکه concatenation دو گرامر regular، regular خواهدبود:

دو زبان L1 و L2 را در نظر بگیرید؛ L(M) = L1 * L2. NFA معادل برای هر زبان را در نظر بگیرید؛ با اتصال آن دو به هم NFA معادل L(M) را خواهیم داشت که regular خواهدبود.

Page 16

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

با توجه به مفروضات، داریم:

$first\ derivation: < expr > => < expr > < weak\ op > < expr >$
$< element > < weak\ op > < expr >$
$=> < element > < weak\ op > < expr > < weak\ op > < expr >$
$=> < element > < weak\ op > < element > < weak\ op > < expr >$
$=> < element > < weak\ op > < element > < weak\ op > < element >$
$=> < variable > < weak\ op > < element > < weak\ op > < element >$
$=> < variable > < weak\ op > < variable > < weak\ op > < element >$
$=> < variable > < weak\ op > < variable > < weak\ op > < variable >$
$=> < variable > - < variable > < weak\ op > < variable >$
$=> < variable > - < varibale > - < variable >$
$=> a - < varibale > - < variable >$
$=> a - b - < variable >$
$=> a - b - c$
$second\ derivation: < expr > => < expr > < weak\ op > < expr >$
$=> < expr > < weak\ op > < element >$
$=> < expr > < weak\ op > < expr > < weak\ op > < element >$
$=> < element > < weak\ op > < element > < weak\ op > < element >$
$=> < element > < weak\ op > < element > < weak\ op > < element >$
$=> < variable > < weak\ op > < element > < weak\ op > < element >$
$=> < variable > < weak\ op > < variable > < weak\ op > < element >$
$=> < variable > < weak\ op > < variable > < weak\ op > < variable >$

=>< *variable* > -< *variable* >< *weak op* >< *variable* >
=>< *variable* > -< *varibale* > -< *variable* >
=> *a*-< *varibale* > -< *variable* >
=> *a* - *b*-< *variable* >
=> *a* - *b* - *c*

در زبان Wren، <expr> به <integer expr> یا <boolean expr> می‌رود که در صورت رفتن به <integer expr>، به جای رفتن به دوتا <integer expr> با یک <weak op> بینشان، به یک <term> و یا یک <integer expr> با یک <weak op> بینشان می‌رود که باعث می‌شود عبارت همواره از یک سمت مشتق شود و ابهامی به‌وجود نیاید.

wren specification:

<expr> ::= <integer expr> | <boolean expr>

<integer expr> ::= <TERM> | <integer expr> <weak op> <TERM>

<term> ::= <element> | <TERM> <strong op> <element>

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

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

1)(line 1) was: context-free error

2)(line 3) redeclaration of variable b: context-sensitive error

3)(line 3) ‘;’ instead of ‘:’: context-free error

4)(line 6) no endif: context-free error

5)(line 6) p := (a+1), p is Boolean and a is an integer (type mismatch): semantic error

6)(line 7) write q: context sensitive [All identifiers that appear in a block must be declared in that block.]

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.

ضرب، جمع = تفریق

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

برای ضرب و تفریق: راست به چپ اجرا می‌شود.

برای جمع: چپ به راست اجرا می‌شود.

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

برای حفظ اولویت‌ها از پرانتز استفاده می‌کنیم.

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

$< roman\ numeral >::=< tens >< units >$
$< tens >::=< \text{low tens} > |XL|L < low\ tens > |XC$
$< units >::=< \text{low units} > |IV|V < low\ units > |IX$
$< low\ tens >::= \text{Ɛ|X} < \text{low tens} >$
$< low\ units >::= \text{Ɛ|I} < \text{low units} >$

$deriving\ XLVII$:

$< roman\ numeral > \Rightarrow < tens >< units >$

$\Rightarrow XL < units >$
$\Rightarrow XLV < low\ units >$
$\Rightarrow XLVI < low\ units >$
$\Rightarrow XLVII < low\ units >$
$\Rightarrow XLVII$

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

فرض کنید می‌خواهیم عبارت a+b*c را به‌دست بیاوریم:

$first\ derivation$:

$< expr > \Rightarrow < term >$

$=> < expr > + < term >$
$=> < expr > + < factor >$
$=> < expr > + < expr > * < factor >$
$=> < expr > + < expr > * < ident >$
$=> < expr > + < expr > * c$
$=> < factor > + < expr > * c$
$=> < factor > + < factor > * c$
$=> a + < factor > * c$
$=> a + b * c$

$second\ derivation$:

$< expr > => < factor >$

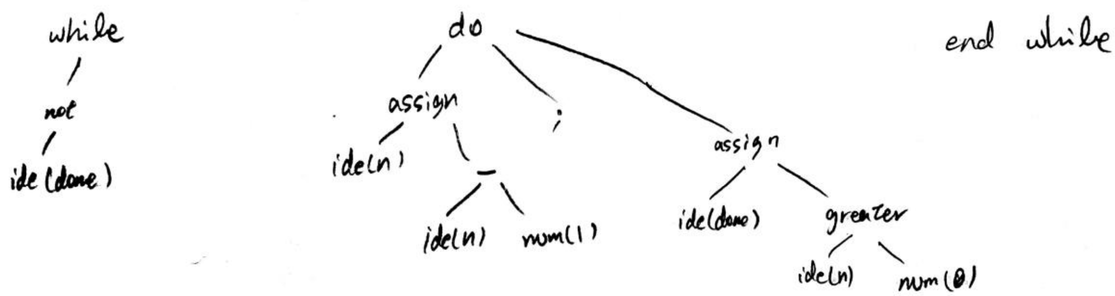
$=> < expr > * < factor >$
$=> < term > * < factor >$
$=> < expr > + < term > * < factor >$
$=> < expr > + < term > * < ident >$
$=> < expr > + < term > * c$
$=> < expr > + < factor > * c$
$=> < expr > + < ident > * c$
$=> < expr > + b * c$
$=> < factor > + b * c$
$=> < ident > + b * c$
$=> a + b * c$

برای رفع ابهام می‌توان از گرامر زیر استفاده کرد:

$< expr > ::= <$ term $> \mid <$ expr $><$ strong op $><$ term $>$
$< term > ::= < factor > \mid < term >< weak\ op >< factor >$
$< factor > ::= < ident > \mid (< expr >)$
$< ident > ::= a \mid b \mid c$
$< strong\ op > ::= * \mid /$
$< weak\ op > ::= + \mid -$

Page 29

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** ;

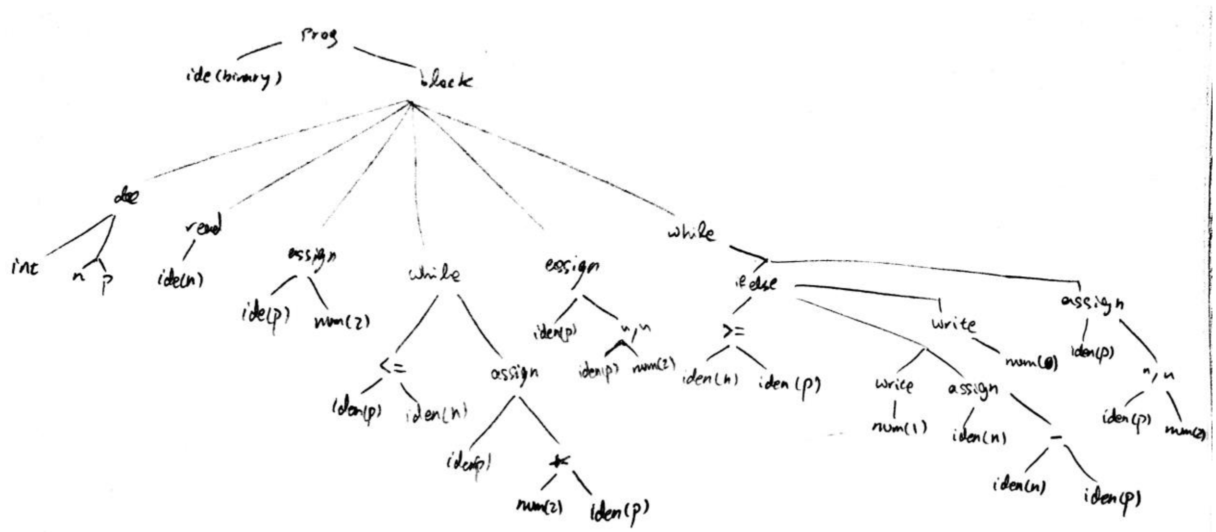
begin

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



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>.

گرامر:

$< string\ literal >::=< numeral > H < string >$
$condition: Len(< numeral >) = Len(< string >)$

$< numeral >::=< digit >$
$Len(< numeral >) \longleftarrow 1$
$| < digit >< numeral >_2$
$Len(< numeral >) \longleftarrow Len(< numeral >_2) + 1$
$< string >::=< letter >$
$Len(< string >)1 \longleftarrow 1$

$| < letter >< string >2$
$Len(< string >) \longleftarrow Len(< string >_2) + 1$

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.

دو val attribute و num را اضافه می‌کنیم:

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

$condition: Num(< hundreds >) \leq 3$ && $Num(< tens >) \leq 3$ && $Num(< units >) \leq 3$
$< hundreds >::=< low\ hundreds >, Num(< hundreds >) \longleftarrow Num(< low\ hundreds >)$
$|CD, Num(< hundreds >) \longleftarrow 1$
$|D < low\ hundreds >, Num(< hundreds >) \longleftarrow Num(< low\ hundreds >)$
$|CM, Num(< hundreds >) \longleftarrow 1$
$< low\ hundreds >::= \varepsilon$
$Num(< low\ hundreds >) \longleftarrow 0$
$| < low\ hundreds >_2 C$
$Num(< low\ hundreds >) \longleftarrow Num(< low\ hundreds >_2) + 1$
$< tens >::=< low\ tens >, Num(< tens >) \longleftarrow Num(< low\ tens >)$

$, Val(< tens >) \longleftarrow Val(< low\ tens >)$
$|XL, Num(< tens >) \longleftarrow 1$
$, Val(< tens >) \longleftarrow 40$
$|L < low\ tens >, Num(< tens >) \longleftarrow Num(< low\ tens >)$
$, Val(< tens >) \longleftarrow 50 + Val(< low\ tens >)$
$|XC, Num(< tens >) \longleftarrow 1$
$, Val(< tens >) \longleftarrow 90$
$< low\ tens >::= \varepsilon$
$Num(< low\ tens >) \longleftarrow 0$
$, Val(< low\ tens >) \longleftarrow 0$
$| < \text{low tens} >_2 \text{X}$
$Num(< low\ tens >) \longleftarrow Num(< \text{low tens} >_2) + 1$
$, Val(< low\ tens >) \longleftarrow 10 + Val(< \text{low tens} >_2)$

$< units >::=< low\ units >, Num(< units >) \longleftarrow Num(< low\ units >)$
$|IV, Num(< units >) \longleftarrow 1$
$|V < low\ units >, Num(< units >) \longleftarrow Num(< low\ units >)$
$|IX, Num(< units >) \longleftarrow 1$
$< low\ units >::= \varepsilon$
$Num(< low\ units >) \longleftarrow 0$
$| < \text{low units} >_2 \text{I}$
$Num(< low\ units >) \longleftarrow Num(< \text{low units} >_2) + 1$

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).

$< binary\ numeral >::=< binary\ digits >. < fractions\ digits >$
$Val(< binary\ numeral >) \longleftarrow Val(< binary\ digits >) + Val(< fraction\ digits >)$
$| < binary\ digits >$
$Val(< binary\ numeral >) \longleftarrow Val(< binary\ digits >)$
$| < binary\ digits >.$
$Val(< binary\ numeral >) \longleftarrow Val(< binary\ digits >)$

$|. < fraction\ digits >$

$Val(< binary\ numeral >) \longleftarrow Val(< fraction\ digits >)$

$< binary\ digits > ::= < binary\ digits >_2 < bit >$

$Val(< binary\ digits >) \longleftarrow 2 * Val(< binary\ digits >_2) + Val(< bit >)$

$| < bit >, Val(< binary\ digits >) \longleftarrow Val(< bit >)$

$< fraction\ digits > ::= < fraction\ digits >_2 < bit >$

$Len(fraction\ digits) \longleftarrow Len(< fraction\ digits >_2) + 1$

$Val(< fraction\ digits >)$

$\longleftarrow Val(< fraction\ digits >_2)$

$+ Val(< bit >) * 2^{-(Len(< fraction\ digits >_2)+1)}$

$| < bit >, Len(fraction\ digits) \longleftarrow 1$

$Val(< fraction\ digits >) \longleftarrow 2^{-1} * Val(< bit >)$

$< bit > ::= 0, Val(< bit >) \longleftarrow 0$

$|1, Val(< bit >) \longleftarrow 1$

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.

$< expr >::=< term >$
$ParantheseNum(< expr >) \longleftarrow ParantheseNum(< term >)$
$OperatorNum(< expr >) \longleftarrow OperatorNum(< term >)$
$MulNum(< expr >) \longleftarrow MulNum(< term >)$
$DivNum(< expr >) \longleftarrow DivNum(< term >)$
$SubNum(< expr >) \longleftarrow SubNum(< term >)$
$| < term >< strongest\ op >< expr >2$
$ParantheseNum(< expr >)$
$\longleftarrow ParantheseNum(< term >) + ParantheseNum(< expr >2)$
$OperatorNum(< expr >)$
$\longleftarrow OperatorNum(< term >) + OperatorNum(< expr >2) + 1$
$MulNum(< expr >) \longleftarrow MulNum(< term >) + MulNum(< expr >2)$
$DivNum(< expr >) \longleftarrow DivNum(< term >) + DivNum(< expr >2)$
$SubNum(< expr >) \longleftarrow SubNum(< term >) + SubNum(< expr >2)$

$condition: ParantheseNum(< expr >) \leq 3$
$condition: OperatorNum(< expr >) \leq 8$
$condition: (DivNum(< expr >) \leq MulNum(< expr >))||(SubNum(< expr >) = 0)$

$< term >::=< phrase >$
$ParantheseNum(< term >) \longleftarrow ParantheseNum(< phrase >)$
$OperatorNum(< term >) \longleftarrow OperatorNum(< phrase >)$
$MulNum(< term >) \longleftarrow MulNum(< phrase >)$
$DivNum(< term >) \longleftarrow DivNum(< phrase >)$
$SubNum(< term >) \longleftarrow SubNum(< phrase >)$ | $< term >2*< phrase >$
$ParantheseNum(< term >) \longleftarrow ParantheseNum(< term >2)$
$+ ParantheseNum(< phrase >)$
$OperatorNum(< term >) \longleftarrow OperatorNum(< term >2) + OperatorNum(< phrase >) + 1$
$MulNum(< term >) \longleftarrow MulNum(< term >2) + MulNum(< phrase >) + 1$
$DivNum(< term >) \longleftarrow DivNum(< term >2) + DivNum(< phrase >)$
$SubNum(< term >) \longleftarrow SubNum(< term >2) + SubNum(< phrase >)$
| $< term >2/< phrase >$
$ParantheseNum(< term >) \longleftarrow ParantheseNum(< term >2)$
$+ ParantheseNum(< phrase >)$
$OperatorNum(< term >) \longleftarrow OperatorNum(< term >2) + OperatorNum(< phrase >) + 1$

$MulNum(< term >) \longleftarrow MulNum(< term >2) + MulNum(< phrase >)$
$DivNum(< term >) \longleftarrow DivNum(< term >2) + DivNum(< phrase >) + 1$
$SubNum(< term >) \longleftarrow SubNum(< term >2) + SubNum(< phrase >)$

$< phrase >::=< factor >$
$ParantheseNum(< phrase >) \longleftarrow ParantheseNum(< factor >)$
$OperatorNum(< phrase >) \longleftarrow OperatorNum(< factor >)$
$MulNum(< phrase >) \longleftarrow MulNum(< factor >)$
$DivNum(< phrase >) \longleftarrow DivNum(< factor >)$
$SubNum(< phrase >) \longleftarrow SubNum(< factor >)$

| $< phrase >2 +< factor >$
$ParantheseNum(< phrase >) \longleftarrow ParantheseNum(< phrase >2)$
$+ ParantheseNum(< factor >)$
$OperatorNum(< phrase >) \longleftarrow OperatorNum(< phrase >2) + OperatorNum(< factor >) + 1$

$MulNum(<phrase>) \longleftarrow MulNum(<phrase>2) + MulNum(<factor>)$
$DivNum(<phrase>) \longleftarrow DivNum(<phrase>2) + DivNum(<factor>)$
$SubNum(<phrase>) \longleftarrow SubNum(<phrase>2) + SubNum(<factor>)$
$| <phrase>2 - <factor>$
$ParantheseNum(<phrase>)$
$\longleftarrow ParantheseNum(<phrase>2)$
$+ ParantheseNum(<factor>)$
$OperatorNum(<phrase>)$
$\longleftarrow OperatorNum(<phrase>2) + OperatorNum(<factor>) + 1$
$MulNum(<phrase>) \longleftarrow MulNum(<phrase>2) + MulNum(<factor>)$
$DivNum(<phrase>) \longleftarrow DivNum(<phrase>2) + DivNum(<factor>)$
$SubNum(<phrase>)$
$\longleftarrow SubNum(<phrase>2) + SubNum(<factor>) + 1$
$<factor> ::= <variable>, ParantheseNum(<factor>) \longleftarrow 1$
$OperatorNum(<factor>) \longleftarrow 0$
$MulNum(<term>) \longleftarrow 0$
$DivNum(<term>) \longleftarrow 0$
$SubNum(<term>) \longleftarrow 0$
$|(<expr>)$
$ParantheseNum(<factor>) \longleftarrow ParantheseNum(<expr>) + 1$
$OperatorNum(<factor>) \longleftarrow OperatorNum(<expr>)$
$MulNum(<factor>) \longleftarrow MulNum(<expr>)$
$DivNum(<factor>) \longleftarrow DivNum(<expr>)$
$SubNum(<factor>) \longleftarrow SubNum(<expr>)$
$<variable> ::= a|b|c$
$<strongest\ op> ::= \uparrow$
$<strong\ op> ::= * \ |/$
$<weak\ op> ::= +| -$

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.

$< binary\ tree >::= nil$
$LeftSubTreeHeight(< binary\ tree >) \longleftarrow nil$
$RightSubTreeHeight(< binary\ tree >) \longleftarrow nil$
$Height(< binary\ tree >) \longleftarrow 0$
$Value(< binary\ tree >) \longleftarrow nil$
$LeftChildValue(< binary\ tree >) \longleftarrow nil$
$RightChildValue(< binary\ tree >) \longleftarrow nil$
$|(< binary\ tree >1< value >< binary\ tree >2)$
$LeftSubTreeHeight(< binary\ tree >) \longleftarrow Height(< binary\ tree >1)$
$RightSubTreeHeight(< binary\ tree >) \longleftarrow Height(< binary\ tree >2)$
$Height(< binary\ tree >)$
$\longleftarrow Max(LeftSubTreeHeight(< binary\ tree$
$>), RightSubTreeHeight(< binary\ tree >) + 1)$
$Value(< binary\ tree >) \longleftarrow Value(< value >)$

$LeftChildValue(< binary\ tree >) \longleftarrow Value(< binary\ tree >1)$
$RightChildValue(< binary\ tree >) \longleftarrow Value(< binary\ tree >2)$
$condition: abs(LeftSubTreeHeight(< binary\ tree >)$
$- RightSubTreeHeight(< binary\ tree >)) \leq 1$
$condition: LeftChildValue(< binary\ tree >) < Value(< binary\ tree >)$
$\leq RightChildValue(< binary\ tree >))$

$< value > ::= < digit >$
$Value(< value >) \longleftarrow Value(< digit >)$
$| < value > 2 < digit >$
$Value(< value >) \longleftarrow 10 * Value(< value > 2) + Value(< digit >)$
$< digit > ::= 0, Value(digit) \longleftarrow 0$
$|1, Value(digit) \longleftarrow 1$
$|2, Value(digit) \longleftarrow 2$
$|3, Value(digit) \longleftarrow 3$
$|4, Value(digit) \longleftarrow 4$
$|5, Value(digit) \longleftarrow 5$
$|6, Value(digit) \longleftarrow 6$
$|7, Value(digit) \longleftarrow 7$
$|8, Value(digit) \longleftarrow 8$
$|9, Value(digit) \longleftarrow 9$