Programming Languages Hw1

Arya Banaeizadeh 9431029

Page 8

4. Remove the syntactic category <prepositional phrase> and all related productions from the grammar in Figure 1.1. Show that the resulting grammar defines a finite language by counting all the sentences in it.

The anticipated grammar would be like this:

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

<noun phrase> ::= <determiner> <noun>

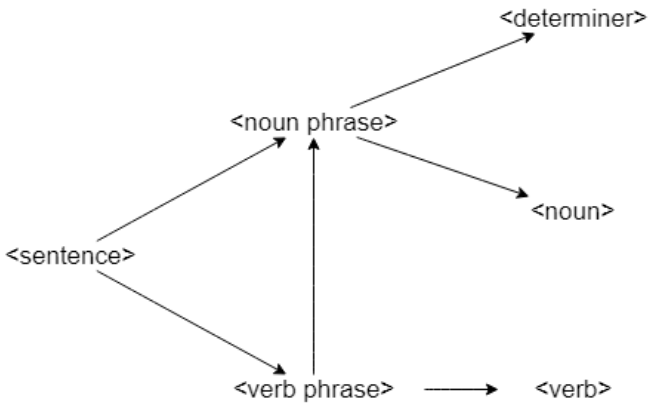
<verb phrase> ::= <verb> | <verb> <noun phrase>

<noun> ::= boy | girl | cat | telescope | song | feather

<determiner> ::= a | the

<verb> ::= saw | touched | surprised | sang

The Dependency Graph has no cycles therefore the grammar is finite:



A sentence consists of noun phrase and verb phrase.

Noun phrase consists of determiner and noun, each have 2 and 6 words in our specified language. Thus total amount of noun phrases are limited to **12**.

Verb can either be a verb or verb and noun phrase, we have 4 verbs, thus sentences in the form of **<sentence> ::= <noun phrase> <verb>.** are in the total amount of 12 times 4 which is **48**. And sentences in the form of **<sentence> ::= <noun phrase> <verb> <noun phrase>.** are in the total amount of 12 times (4 times 12) which is **576**.

In conclusion, **we have 624 sentences**.

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

Figure 1.6:

<sentence> ::= abc | a<thing>bc

<thing>b ::= b<thing>

<thing>c ::= <other>bcc

a<other> ::= aa | aa<thing>

b<other> ::= <other>b

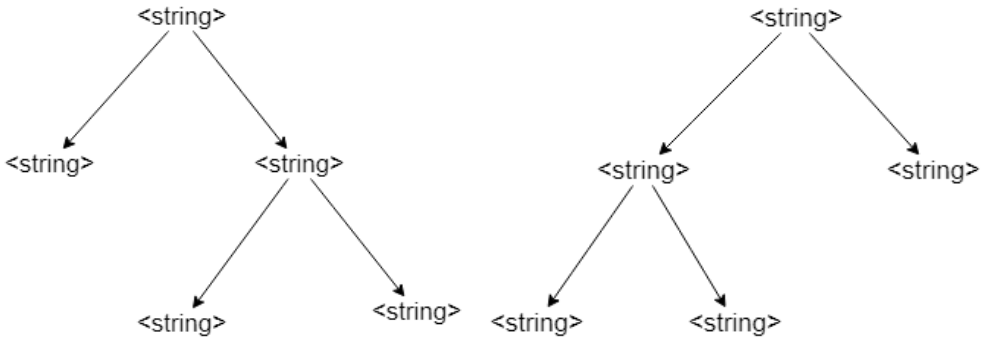
Derivation:

<sentence> ::= a<thing>bc	**=>**	**a<thing>bc**
<thing>b ::= b<thing>	**=>**	**ab<thing>c**
<thing>c ::= <other>bcc	**=>**	**ab<other>bcc**
b<other> ::= <other>b	**=>**	**a<other>bbcc**
a<other> ::= aa | aa<thing>	**=>**	**aa<thing>bbcc**
<thing>b ::= b<thing>	**=>**	**aab<thing>bcc**
<thing>b ::= b<thing>	**=>**	**aabb<thing>cc**
<thing>c ::= <other>bcc	**=>**	**aabb<other>bccc**
b<other> ::= <other>b	**=>**	**aab<other>bbccc**
b<other> ::= <other>b	**=>**	**aa<other>bbbccc**
a<other> ::= aa | aa<thing>	**=>**	**aaabbbccc** **desired outcome.**

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

this form is ambiguous. Consider **<string><string><string>** that may be achieved by these 2 derivations:



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

this grammar is not ambiguous and each sentence derived from this grammar has a unique derivation tree. (This can be practically used to prioritize strings, like mathematical formulas, etc.)

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

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

It produces ab, aabb, aaabbb , … .

In general $a_{(n)}b_{(n)}$ where n is equal to or more than 1. -> type 3

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

It produces ε , aa, bb, aaaa, bbbb, abba, baab, ababbaba , … .

In other words all the palindrome sets of strings with even numbers of a and b including zero. This grammar cannot be regularized. Type 2

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

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

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

It produces ba, baba, baab, bbaa, ab, abab, abba, aabb, aaaaaaaaaabb, … .

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.

First, we show that concatenation of two regular grammars is regular (R1 and R2 for instance). We know that we can represent a regular grammar using regular expressions. Now if L and M are considered to be languages of R1 and R2 then R1 + R2 is too a regular expression considering L U M is its language.

Here is the modified grammar of figure 1.1:

<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

Now if we prove noun phrase and verb phrase have a regular grammar separately we can deduce our desired outcome.

Since words in each section are limited to one we can simplify our grammer.

Noun phrase:

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

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

These two rules fit into our premise (<A> ::= a1 a2 a3 …an <B>) therefore noun phrase is a regular grammar.

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>

Thus,

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

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

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

We can remove **saw a boy <prepositional phrase><prepositional phrase>** since it can be produced using **saw <noun phrase> <prepositional phrase>.**

We proved that these two grammars are regular thus concluding sentence grammar is regular.

Page 16

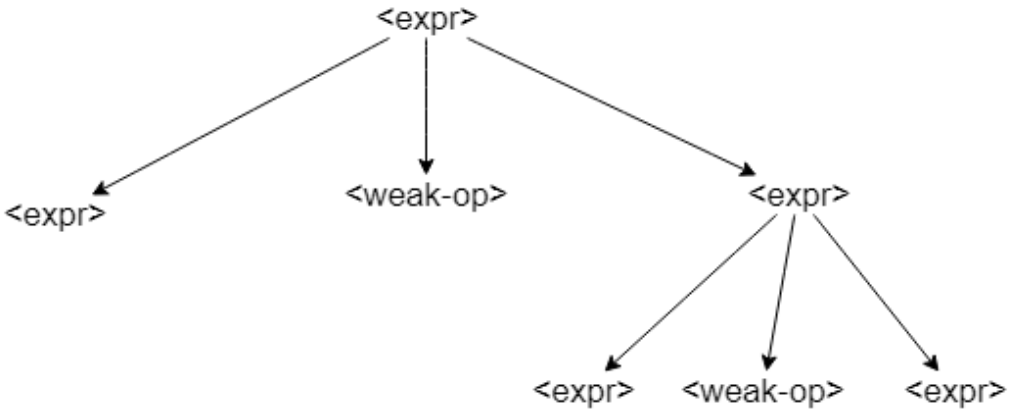
2. Consider the following specification of expressions:

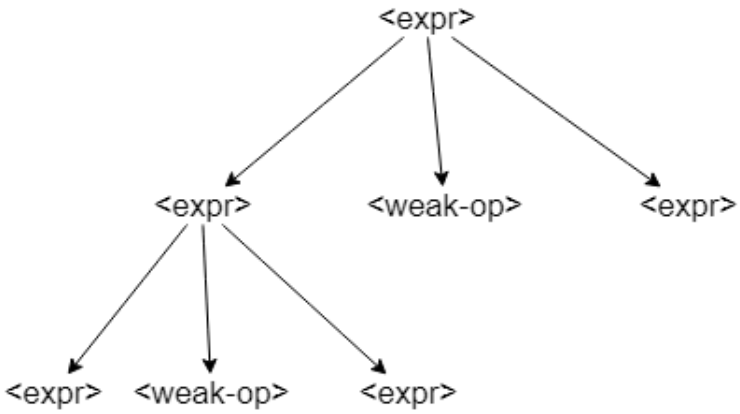
<expr> ::= <element> | <expr> <weak op> <expr>

<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 distinguishes left and right side of the operator by different non terminals which prioritizes operations accordingly.

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
```

Errors:

1.	was is used instead of is.	Context-free
2.	b is declared twice.	Context-sensitive
3.	Use of ; instead of : in var p,b ; boolean ;.	Context-free
4.	a is not defined.	Semantic
5.	a := 34; wrong use of semicolon.	Context-free
6.	Unknown sign ≠ Invalid expression.	Context-free
7.	Integer assigned to Boolean (p).	Context-sensitive
8.	p := (a+1); wrong use of semicolon.	Context-free
9.	no **end if.**	Context-free
10.	write is defined for integer expr.	Context-sensitive

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.

Operations given in order: **– , + , ***

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

– : LtR (object can be expanded)

+: RtL (thing can be expanded)

*: LtR (expr can be expanded)

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

An object in parentheses can be used as an element for example it may be used in this way:

(a – b – c – d – (a – b))

However it cannot be used with any other operations except –.

7. Explain the relation between left or right recursion in definition of expressions and terms, and the associativity of the binary operations (left-to-right or right-to-left).

If the expression is defined in the following manner it would be left-to-right since the recursion will happen on the right side.

<expr> ::= <term> | <term> or <expr>

This type however represents a right-to-left expression:

<expr> ::= <term> | <expr> or <term>

For binary operations these two expressions act in a similar way since we have :

(a or b) or c is equal to **a or (b or c)**

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

According to roman numbers, we have:

<roman number> ::= <ones> | <fives> | <tens> | <fifties> | <hundreds>

<ones> ::= I | <ones>I

<fives> ::= V | V<ones> | IV

<tens> ::= X | X<ones> | X<fives> | IX | X<tens>

<fifties> ::= L | XL | <fifties><ones> | <fifties><fives> | <fifties><tens>

<hundreds> ::= C | XC | <hundreds><ones>|<hundreds><fives>

Thus, XLVII can be excluded:

- ⇨ <roman number>
- ⇨ <fifties>
- ⇨ <fifties><fives>
- ⇨ XL<fives>
- ⇨ XLV<ones>
- ⇨ XLV<ones>I
- ⇨ XLVII

We use <hundreds> for 90-100. It shall not be used for numbers greater than a hundred.

9. Consider a language of expressions over lists of integers. List constants have the form: [3,-6,1], [86], [ ]. General list expressions may be formed using the binary infix operators +, –, *, and @ (for concatenation), where * has the highest precedence, + and - have the same next lower precedence, and @ has 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.

Example: [1,2,3] + [2,2,3] * [5,-1,0] @ [8,21] evaluates to [11,0,3,8,21].

Write a BNF specification for this language of list expressions. Assume that <integer> has already been defined. The conformity of lists for the arithmetic operations is not handled by the BNF grammar since it is a context-sensitive issue

<list> ::= [<integers>] | [] | (<expr>)

<integers> ::= <integer> | <integer> , <integers>

<mid op> ::= + | -

<obj> ::= <obj> @ <list> | <list>

<term> ::= <obj> <mid op> <term> | <obj>

<expr> ::= <term> * <expr> | <term>

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

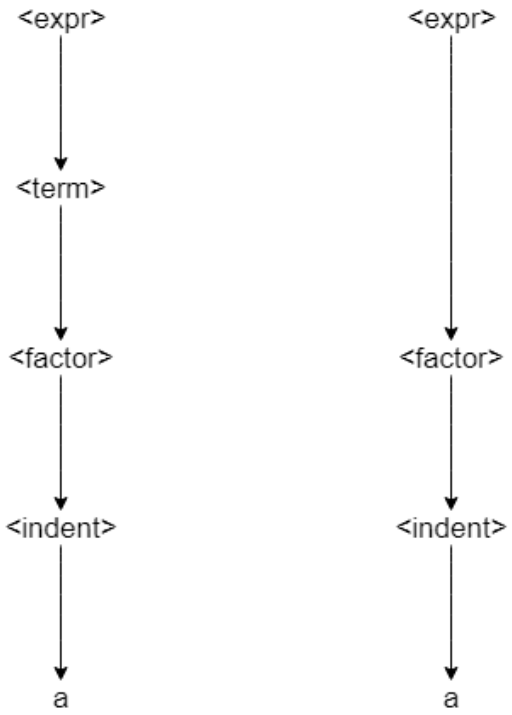
<expr> ::= <term> | <factor>

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

<factor> ::= <ident> | (<expr>) | <expr> * <factor>

<ident> ::= a | b | c

In this grammar + has higher precedence than *. Also we can use parentheses to override the rules. The ambiguity rises from showing a single ident.



This is the unambiguous form:

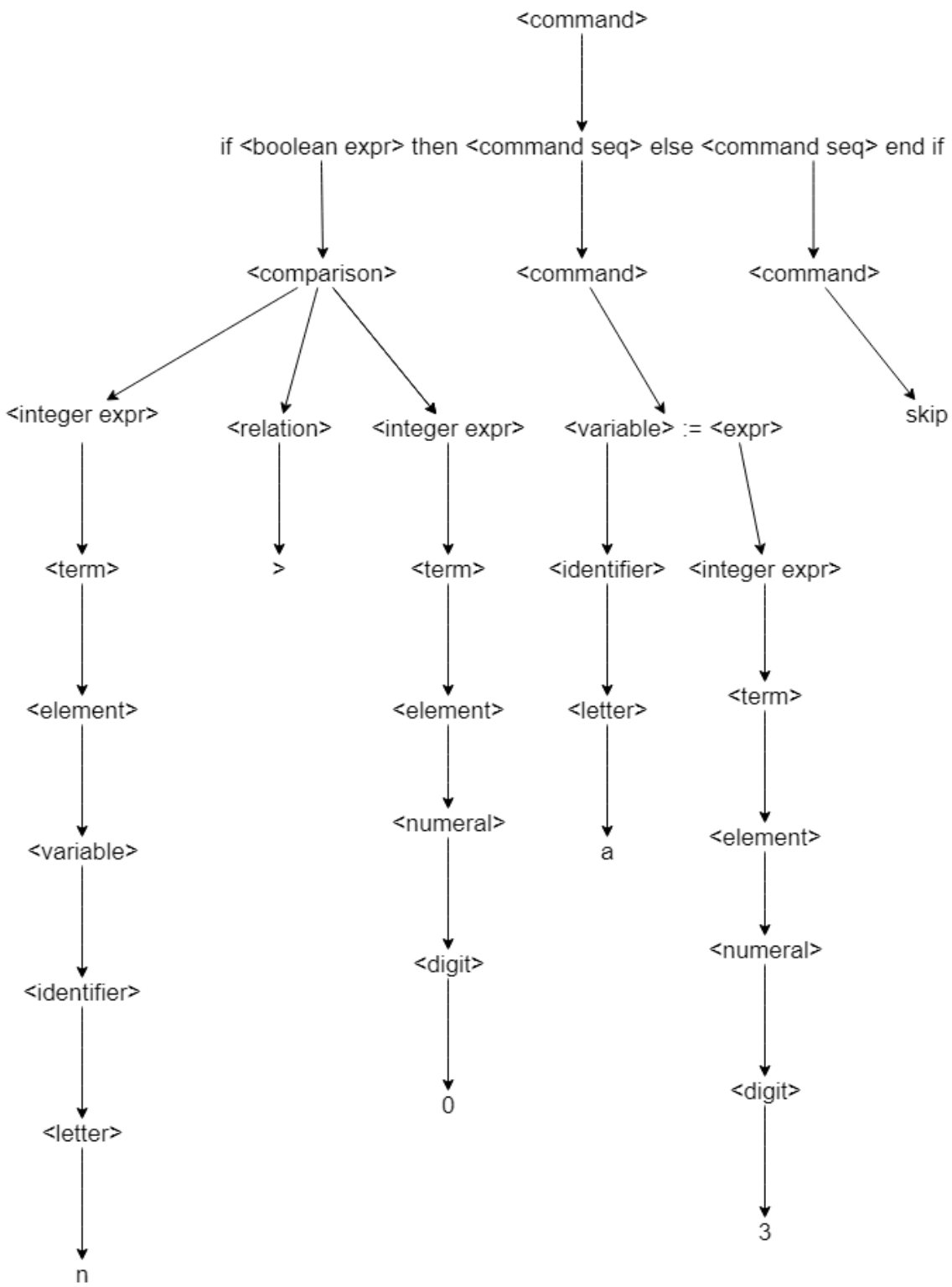
<expr> ::= <term>

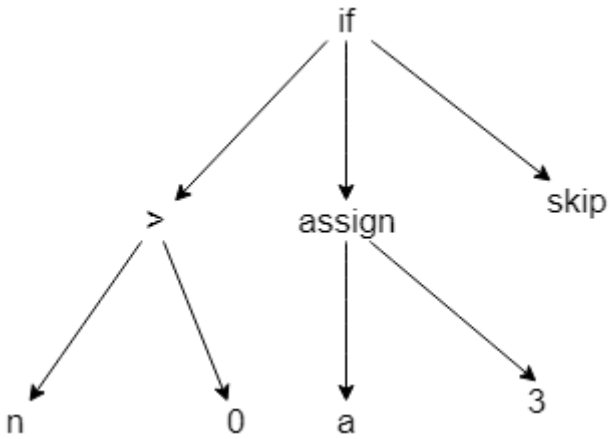
<term> ::= <factor> | <expr> + <term>

<factor> ::= <ident> | (<expr>) | <expr> * <factor>

<ident> ::= a | b | c

1. Construct a derivation tree and an abstract syntax tree for the Wren command “if n>0 then a := 3 else skip end if ”.Also write the abstract tree as a Prolog structure.

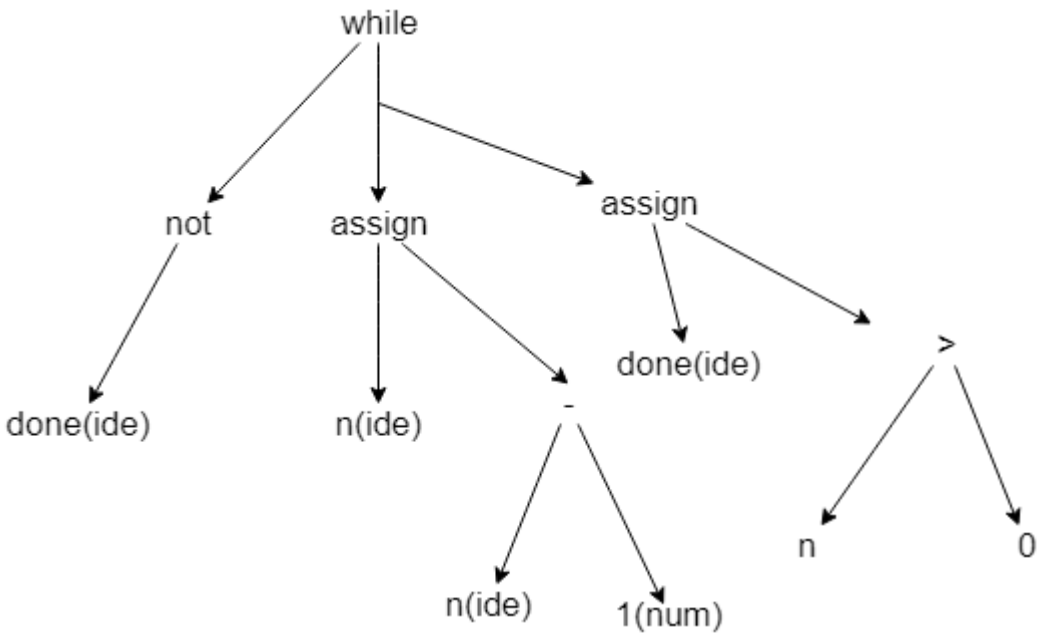




Command(Boolean expr(>,ide(n),num(0)),[assign(ide(a),num(3)],[skip])

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]



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>.

Condition:

size(<numeral>) == size<string>

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

<numeral> ::= <digit> size(<numeral>) ← size(<digit>)

| <numeral><digit> size(<numeral>) ← size($<numeral_2>$) * 10 + size(<digit>)

<string> ::= ε size(<string>) ← 0| <string><character> size(<string>) ← size($<string_2>$) + 1

2. Repeat exercise 1, using a synthesized attribute for <numeral> and an inherited attribute for <string>.

<string> ::= ε **condition: inhSize(string) = 0**

| $<string_2><character>$ Inhsize($<string_2>$) ← Inhsize(<string>) - 1

<numeral> ::= <digit> size(<numeral>) ← size(<digit>)

| $<numeral><digit>$ size(<numeral>) ← size($<numeral_2>$) * 10 + size(<digit>)

3. Repeat exercise 1, using an inherited attribute for <numeral> and a synthesized attribute for <string>.

<string> ::= ε size(<string>) ← 0| $<string><character>$ size(<string>) ← size($<string_2>$) + 1

<numeral> ::= <digit> **condition: inhsize(<numeral>) = 0**

| $<numeral><digit>$ Inhsize($<numeral_2>$) ← [Inhsize(<numeral>) / 10]

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.

Condition:

Size(<low units>) <= 3 and Size(<low tens>) <= 3 and Size(<low hundreds>) <= 3

<low units> ::= ε	Size(<low units>) ← 0
| <low units> C	Size(<low units>) ← Size(<low units$_2$>) +1
<low tens> ::= ε	Size(<low tens>) ← 0
| <low tens> C	Size(<low tens>) ← Size(<low tens$_2$>) +1
<low hundreds> ::= ε	Size(<low hundreds>) ← 0
| <low hundreds> C	Size(<low hundreds>) ← Size(<low hundreds$_2$>) +1

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

<roman> ::= <hundreds> <tens> <units> val(<roman>) ← val(<hundreds>) + val(<tens>) + val(<units>)

<hundreds> ::= <low hundreds> val(<hundreds>) ← size(<low hundreds>)

| CD val(<hundreds>) ← 400

| D <low hundreds> val(<hundreds>) ← 500 + size(<low hundreds>)

| CM val(<hundreds>) ← 900

<tens> ::= <low tens> val(<tens>) ← size(<low tens>)

| XL val(<tens>) ← 40

| L <low tens> val(<tens>) ← 50 + size(<low tens>)

| XC val(<tens>) ← 90

<units> ::= <low units> val(<units>) ← size(<low units>)

| IV val(<units>) ← 4

| V <low units> val(<units>) ← 5 + size(<low units>)

| IX val(<units>) ← 9

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 digit> val (<binary numeral>) ←val (<binary digit>)

|<binary digit>. val (<binary numeral>) ← val (<binary digit>)

| .<binary digit> val (<binary numeral>) ←val (<binary digit>) / $2^{\text{len (<binary digit>)}}$

Rest of the grammar remains unchanged:

<binary digits> ::= <binary digits>2 <bit> Val (<binary digits>) ← 2 • Val (<binary digits>2) + Val (<bit>)

Len (<binary digits>) ← Len (<binary digits>2) + 1

| <bit> Val (<binary digits>) ← Val (<bit>) Len (<binary digits>) ← 1

6. Develop an attribute grammar for integers that allows a leading sign character (+ or -) and that ensures that the value of the integer does not exceed the capacity of the machine. Assume a two's complement representation; if the word-size is n bits, the values range from -2^{n-1} to $2^{n-1}-1$.

Condition:

Val(<integers>) < 2^n

<integers> ::== <sign><digits> **val(integers) ← val(digits) + 2^n * val(sign)**

<digits> ::= <digit> | <digits><digit> val(digits) ← digits.getValue

<sign> ::= + val(sign) ← 0 | - val(sign) ← 1

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

Note: getValue returns the numeral value of digits.

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

condition:

val(paren) <= 3 and val(a) <= 8 and val(b) <=8 and val(c) <= 8 and val(sub) * (val(mul) – val(div)) >= 0

<identifiers> ::= a **val(a) ← val(a) + 1**

| b val(b) ← **val(b) + 1**

| c val(c) ← **val(c) + 1**

| (expr) **val(paren) ← val(paren) + 1**

<expr> ::= <expr> ↑ <term> | <term>

<term> ::= <obj> <strong op> <term> | <obj>

<obj> ::= <identifier> <strong op> <obj> | <identifier>

<strong op> ::= * **val(mul) ← val(mul) + 1| / val(div) ← val(div) + 1**

<weak op> ::= + | - **val(sub) ← val(sub) + 1**

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.

Height property:

Condition:

Abs(height(<binary tree$_1$>) – height(<binary tree$_2$>)) <= 1,

Val(<binary tree$_1$>) <= Val(<binary tree>),

Val(<binary tree>) <= Val(<binary tree$_2$>)

<binary tree> ::= nil height ← 0

| (<binary tree$_1$> <value> <binary tree$_2$>)

Val ← int(<value>)

height ← max(height(<binary tree$_1$>) , height(<binary tree$_2$>)) + 1

lTree ← val(<binary tree$_1$>)

rTree ← val(<binary tree$_2$>)

We assume that int(<value>) returns the numeral value of <value>.