1a) Not well-formed, violation of grammar rule [14]: the symbol "<" is not allowed
   Inside of CharData
1b) Not well-formed, violation of grammar rule [5]: the symbols "<" and ">" may not appear
   inside of tag name.
1c) Not well-formed, violation of grammar rule [39] (there are four b-Start-tags but
   only three b-End-tags)
1d) Not well-formed, violation of grammar rule [10]: the symbol "<" may not appear inside of
   an attribute value.
1e) Well-formed.
1f) Well-formed.
1g) Well-formed.

2) 
n=root;
   repeat { 
      while(lastChild(n)!=NIL) 
      { 
        n=lastChild(n):
        If(nodeType(n)==TEXT_NODE) print(nodeValue(n)):
      } 
      while(previousSibling(n)=NIL) 
      { 
        n=parent(n); 
        n=nextSibling(n); 
        if(nodeType(n)==TEXT_NODE) print(nodeValue(n)):
      } 
   }

3) 
id=1
   while (lab(id)!=""")
   { 
      if (lab(id)=="a") count[id]=1 else count[id]=0;
      for each child in dag(id) do 
      { 
        count[id] = count[id] + count[child]
      }
      id = id + 1
   }

4) When computing the minimal DAG, we need to determine whether a given subtree has
   occurred already. If we keep a table of pointers to subtrees that have already occurred,
   then to check for a given subtree if it is in the table takes worst case time
   (# of trees in table) * (# nodes in the subtree)
   Which in the worst case is quadratic in the size of the input tree!
With hashing, we only need
(#trees in the hash bucket) * (#nodes in the subtree).

For the example, take  hash(tree) = 1 if tree is a leaf and
hash(tree) = 2 if not a leaf and contains no "f"
hash(tree) = 3 in all other cases.

Then
hash( c ) = bucket 1
hash( b(c, c) ) = bucket 2
hash( f ) = bucket 1
hash( b(f,c) ) = bucket 3
Etc.
Without hashing: check up to 6 nodes each time.
With hash: check only up to 3 nodes each time!

5) Descendants(Node p){
   for(i=1; i<size(p); i++) print( p + i )
}
Children(Node p){
   c = p+1;
   while( c < p+size(p) ) { print( c ); c = c+size(c) }
}
Parent(Node p){
   for(i=1; i<p; i++) if p is in Children(i) then print(i)
}
Following-Siblings(Node p){
   f = p + size(p);
   while( f < Parent(p) + p ) { print( f ); f = f+size(f) }
}
Preceding(Node p){
   for(i=1; i<p; i++) if(p not in Descendants (i)) then print(i)
}

6a) The string "a" is accepted; the string "c" is not accepted.
It is not deterministic (the initial state has two outgoing a-edges)

6b) c*(a+b)(a+b+c)*

6c) Not 1-unambiguous: Glushkov automaton is non-deterministic.
6d)

(b*(ab)*)*